Christian Müller-Schloer
Wolfgang Karl
Sami Yehia (Eds.)

LNCS 5974

Architecture of Computing Systems – ARCS 2010

23rd International Conference
Hannover, Germany, February 2010
Proceedings

Springer

Christian Müller-Schloer
Wolfgang Karl Sami Yehia (Eds.)

Architecture of Computing Systems - ARCS 2010

23rd International Conference
Hannover, Germany, February 22-25, 2010
Proceedings

Springer

Volume Editors

Christian Müller-Schloer
Leibniz University Hannover
Appelstraße 4, 30167 Hannover, Germany
E-mail: cms@sra.uni-hannover.de

Wolfgang Karl
Karlsruhe Institute of Technology (KIT)
Kaiserstraße 12, 76131 Karlsruhe, Germany
E-mail: karl@kit.edu

Sami Yehia
Thales Research and Technology, Campus Polytechnique
1 Avenue Augustin Fresnel, 91767 Palaiseau Cedex, France
E-mail: sami.yehia@thalesgroup.com

Library of Congress Control Number: 2010921317

CR Subject Classification (1998): C.2, C.5.3, D.4, D.2.11, H.3.5, H.4, H.5.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-11949-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-11949-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180 5 4 3 2 1 0

Preface

The ARCS series of conferences has over 30 years of tradition reporting top-notch results in computer architecture and operating systems research. It is organized by the special interest group on "Computer and System Architecture" of the GI (Gesellschaft für Informatik e.V.) and ITG (Informationstechnische Gesellschaft im VDE Information Technology Society). In 2010, ARCS was hosted by Leibniz University Hannover.

This year's special focus was on heterogeneous systems. The conference's topics comprised design aspects of multi-cores and memory systems, adaptive system architectures such as reconfigurable systems in hardware and software, customization and application-specific accelerators in heterogeneous architectures, organic and autonomic computing, energy-awareness, system aspects of ubiquitous and pervasive computing, and embedded systems.

The call for papers attracted about 55 submissions from all around the world. Each submission was assigned to at least three members of the Program Committee for review. The Program Committee decided to accept 20 papers, which were arranged in seven sessions. The accepted papers are from Belgium, China, France, Germany, Italy, Spain, Turkey, and the UK. Two keynotes on heterogeneous systems complemented the strong technical program.

We would like to thank all those who contributed to the success of this conference, in particular the members of the Program Committee (and the additional reviewers) for carefully reviewing the contributions and selecting a high-quality program. The Workshops and Tutorials were organized and coordinated perfectly by Michael Beigl and Francisco J. Cazorla Almeida. Our special thanks go to the members of the Organizing Committee for their numerous contributions: Rainer Buchty set up the conference software. Thomas B. Preußler designed and maintained the website. David Kramer and Martin Schindewolf took over the tremendous task of preparing this volume. We especially would like to thank Yvonne Bernard, Jörg Hähner, Björn Hurling, and Jürgen Brehm for taking care of the local arrangements and the many other aspects of preparing the conference.

February 2010

Christian Müller-Schloer
Wolfgang Karl
Sami Yehia

Organization

General Chair

Christian Müller-Schloer	Leibniz Universität Hannover, Germany

Program Chairs

Wolfgang Karl	Karlsruhe Institute of Technology (KIT), Germany
Sami Yehia	THALES Research and Technology, France

Workshop and Tutorial Chair

Michael Beigl	TU Braunschweig, Germany
Francisco J.C. Almeida	Barcelona Supercomputing Center (BSC), Spain

Proceedings

David Kramer	Karlsruhe Institute of Technology (KIT), Germany
Martin Schindewolf	Karlsruhe Institute of Technology (KIT), Germany

Local Organization

Jörg Hähner	Leibniz Universität Hannover, Germany
Yvonne Bernard	Leibniz Universität Hannover, Germany

Local Organization and Finance

Jürgen Brehm	Leibniz Universität Hannover, Germany

Program Committee

Frank Bellosa	Karlsruhe Institute of Technology (KIT), Germany
Mladen Berekovic	TU Braunschweig, Germany
Arndt Bode	Technische Universität München, Germany
Plamenka Borovska	TU Sofia, Bulgaria

Hartmut Schmeck	Karlsruhe Institute of Technology (KIT), Germany
Karsten Schwan	Georgia Tech, USA
Olaf Spinczyk	University of Dortmund, Germany
Martin Schulz	LLNL, USA
Cristina Silvano	Polimi, Italy
Leonel Sousa	TU Lisbon, Portugal
Rainer G. Spallek	TU Dresden, Germany
Jarmo Takala	Tampere University of Technology, Finland
Djamshjd Tavangarian	University of Rostock, Germany
Jürgen Teich	Universität Erlangen, Germany
Olivier Temam	INRIA, France
Lothar Thiele	ETH Zurich, Switzerland
Pedro Trancoso	University of Cyprus, Cyprus
Theo Ungerer	University of Augsburg, Germany
Mateo Valero	UPC, Spain
Stephane Vialle	Supelec, France
Lucian Vintan	Lucian Blaga University of Sibiu, Romania
Klaus Waldschmidt	University of Frankfurt, Germany
Stephan Wong	Delft University of Technology, The Netherlands

Reviewers

Fakhar Anjam
Ramon Beivide
Frank Bellosa
Mladen Berekovic
Arndt Bode
Plamenka Borovska
Koen De Bosschere
Jürgen Branke
Uwe Brinkschulte
Philip Brisk
Jiannong Cao
João M.P. Cardoso
Luigi Carro
Fran Cazorla
Nate Clark
Nikitas Dimopoulos
Oliver Diessel
Marc Duranton
Babak Falsafi
Paolo Faraboschi
Fabrizio Ferrandi
Alois Ferscha
Björn Franke
Jörg Henkel
Andreas Herkersdorf
Mike Hinchey
Christian Hochberger
Murali Jayapala
Gert Jervan
Chris Jesshope
Ben Juurlink
Kamil Kedzierski
Andreas Koch
Krzysztof Kuchcinski
Paul Lukowicz
Erik Maehle
Ahmed El Mahdy
Dragomir Milojevic
Faisal Nadeem
Dimitrios Nikoplopoulos
Alex Orailoglu
Emre Ozer

Table of Contents

HyVM - Hybrid Virtual Machines - Efficient Use of Future Heterogeneous Chip Multiprocessors

Karsten Schwan, Ada Gavrilovska, and Sudha Yalamanchili

Georgia Institute of Technology

Abstract. The HyVM project is developing system support for future heterogeneous chip multiprocessors. Such hybrid hardware platforms offer opportunities in terms of improved power/performance properties, but pose challenges to systems technologies due to heterogeneous processing cores, non-uniform memory access, and complex software stacks. The HyVM project is creating new hypervisor- and system-level abstractions in support of providing a uniform program execution model for future hybrid computing platforms. Rather than treating accelerators as external devices, the model anticipates future integrated systems by providing sets of virtual processing units for use by both accelerator and commodity programs, offering the resource management support needed to efficiently execute such parallel multi-core applications, and supplying the tool chains needed, at hypervisor level, to permit applications to freely use arbitrary combinations of accelerator and commodity cores. The talk will overview the HyVM project, review results that range from efficient methods for virtualizing accelerators, to online techniques for managing heterogenous system resources, to JIT binary translation for dealing with diverse accelerator targets. The effort is driven by both commercial and high performance applications targeting future hybrid machines.

How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT

Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer

Institute of Computer Science
University of Augsburg
86159 Augsburg, Germany
{mische,guliashvili,uhrig,ungerer}@informatik.uni-augsburg.de

Abstract. This paper describes how a superscalar in-order processor must be modified to support Simultaneous Multithreading (SMT) such that time-predictability is preserved for hard real-time applications. For superscalar in-order architectures the calculation of the Worst Case Execution Time (WCET) is much easier and tighter than for out-of-order architectures. By a careful enhancement that completely isolates the threads, this capability can be perpetuated to an in-order SMT architecture. Our design goal is to minimise the WCET of the highest priority thread, while releasing as many resources as possible for the execution of concurrent non critical threads. The resultant processor executes hard real-time threads at the same speed as its singlethreaded ancestor, but idle issue slots are dynamically used by non critical threads. The modifications to enable SMT are demonstrated by CarCore, a multithreaded embedded processor that implements the Infineon Tricore instruction set.

1 Introduction

The common way to construct a simultaneous multithreaded (SMT) processor is to take a superscalar out-of-order processor and allow it to fetch from multiple threads [1]. This procedure is simple, the fetch stage must be modified and the number of registers should be enhanced, but there are only minor modifications at the internal logic of the pipeline. Despite its simplicity, this combination greatly improves processor throughput [1]. But there are two drawbacks of out-of-order SMT processors: they consume a lot of chip area and energy and it is hard to predict the Worst Case Execution Time (WCET) because of the dynamic allocation of processor resources, making this kind of SMT improper for hard real-time applications.

We eliminate these drawbacks by taking an in-order superscalar processor as base architecture for an alternative implementation of SMT, called *In-Order Simultaneous Multithreading*. The Intel Atom processor [2] is a well-known representative of this class of SMT processors, although it only benefits of a smaller transistor count and lower energy consumption than comparable out-of-order SMT processors. The second advantage, the deterministic behaviour that allows for tight WCET analyses is addressed in this paper. By adding strictly prioritised multithreading capabilities that completely isolate threads from each other, the tight WCETs of superscalar in-order processors can be preserved, while the utilisation and energy-efficiency of the processor is increased by concurrent threads.

The contributions of this paper are:

– an in-order SMT processor that isolates the highest priority thread (HPT),
– the execution of the HPT as if the underlying processor was a singlethreaded superscalar processor to keep the WCET analysis tight,
– a detailed description how the pipeline must be modified to enable SMT and
– a prototype of an SMT processor with TriCore instruction set architecture.

The resulting architecture is called CarCore and we already published articles on other aspects of the architecture: [3] describes how lower priority memory accesses are delayed to avoid influence on the HPT and it presents a scheduler that executes multiple hard real-time threads by time slicing the HPT. In [4] a soft real-time scheduler with direct IPC control based on the CarCore architecture is introduced and [5] discussed the integration of scratchpad memories.

The rest of the paper is organised as follows: the next section presents the related work and section 3 explains the TriCore architecture and the differences to the baseline singlethreaded CarCore processor. In section 4 the enhancements to enable SMT are described in detail. Section 5 discusses our evaluation results and section 6 concludes the paper.

2 Related Work

Tullsen [1] defined SMT as multithreading for superscalar pipelines. He did not specify the execution order, but he used an out-of-order processor as base architecture and so did most of the later SMT researchers. Consequently, most of the work on real-time and SMT is also based on out-of-order pipelines [6,7,8,9]. But the unpredictability of out-of-order pipelines does not allow hard real-time execution, only soft real-time scheduling is addressed.

Although Hily [10] already showed in 1999 that in-order SMT increases total throughput, while out-of-order execution only boosts one single thread and is less cost-effective, only few studies focus on designing SMT processors with in-order pipelines [11]. Similar results were published by Moon [12], who discovered, that static partitioning and execution in-order has only little negative effect on the performance while significantly reducing design complexity. Other studies that divide the pipeline into an out-of-order front-end and an in-order back-end [13] or that restrict certain parts of the pipeline to in-order execution [14] approved the advantages of in-order execution.

Zang et al. [11] investigated issue mechanism for in-order SMT processors. Their processor has a 7 stage pipeline and can issue up to 6 instructions from 6 concurrent threads. A well-known commercial processor that has an in-order SMT architecture is the Intel Atom [2] with a two-way in-order pipeline. But none of the mentioned works address hard real-time execution, to our knowledge our project is the first on hard real-time for in-order SMT processors.

The *Real-time Virtual Multiprocessor (RVMP)* [15] issues multiple instructions from multiple threads to multiple pipelines, but it assumes multiple identical pipelines and statically maps threads to pipelines. Therefore multiple hard real-time threads can be executed, but the throughput is not increased, as idle pipeline slots cannot be used dynamically by other threads.

The *Precision Timed (PRET) Architecture* [16] is another example of a hard real-time capable multithreaded processor, but again the schedule is very static: there are 6 threads and they are executed in fixed order, hence every thread gets exactly one sixth of the execution time. If a thread is stalled, the cycle cannot be used by another thread, as the PRET architecture supports only precisely timed hard real-time threads, no other threads with softer timing demands can be executed to increase throughput.

3 Baseline

Exemplary we use a TriCore compatible processor to present the SMT enhancements, but they can easily be transferred to other superscalar in-order architectures. TriCore-specific parts are explicitly marked.

3.1 TriCore Architecture

The Infineon TriCore [17] is a microcontroller that is commonly used in safety-critical applications of the automotive industry. It combines a real-time capable load-store microcontroller architecture with DSP instructions. The instruction set comprises more than 700 instructions. Besides the common arithmetic, logic, branch and load-store instructions it provides instructions for sophisticated logic, context saving, load-modify-store, packed arithmetic, saturated math and multiply-accumulate. The processor consists of a three-way superscalar in-order pipeline with four stages. If an address, an integer, and a loop instruction appear in this order in the instruction stream, they are issued within one cycle, even if they are data-dependent.

3.2 Simplifications for Single-Threaded CarCore

As baseline for the SMT enhancement we implemented a cycle-accurate SystemC model and a synthesisable VHDL model of a Tricore-compatible processor. It differs from the original Infineon TriCore in the following aspects:

Instruction Subset. Special DSP instructions and addressing modes which are never generated by the Hightec [18] compiler are not supported. This reduces the number of instructions to 433, but there is no impact on the execution time, as only pure C code without assembler code snippets is used.

Later Address Calculation. CarCore calculates branch target and memory access addresses in the execute stage, one stage later than TriCore. Hence the branch and memory delay slots are increased by one, but the critical path of the very complex and slow decode stage is shortened resulting in a higher overall clock rate.

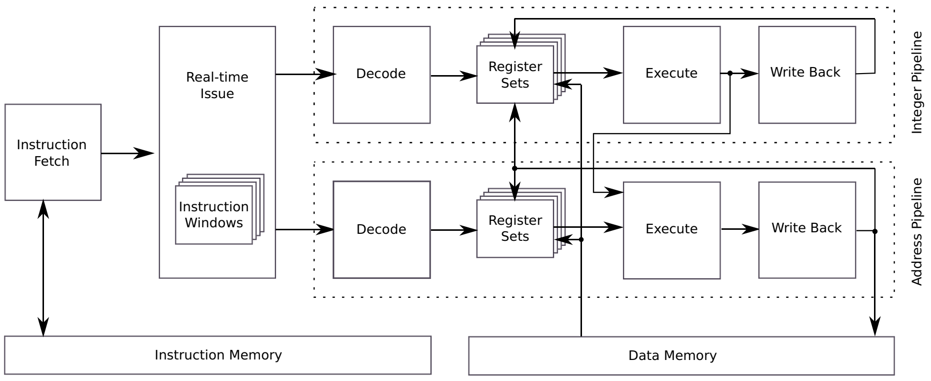


Fig. 1. Block diagram of the CarCore architecture

No Branch Prediction. The HPT does not benefit of a branch prediction, because it only decreases the average, not the worst case execution time. For lower priority threads the benefit is likewise low, as between the branch instruction and the first target instruction usually HPT instructions must be executed anyway. Omitting the speculative execution replaces wasted mis-predicted instructions by instructions of lower priority threads that increase the overall throughput.

No Loop Pipeline. The TriCore loop pipeline that speeds up special loops is not implemented, as the compiler can use it only in a few special cases and with multithreading the latencies can be used to execute concurrent threads.

No Dedicated Context Save Memory. According to the TriCore instruction set architecture, at a subroutine call, 16 registers have to be saved. To speed this up, the Tricore has a special memory area with a very wide bus for context saving. In our version we use the standard memory bus, therefore a function call is about ten times slower.

4 SMT Enhancements

To enable multihreading, some parts of the processor must be duplicated for every thread: the register set, the program counter and the instruction window. To manage these instruction windows where fetched instructions are buffered and to decide which instruction from which thread should be issued to which pipeline, a further pipeline stage is added, the *Real-Time Issue (RTI)* stage. Special attention demands the fetch stage, which is in charge of preventing the lower priority threads from delaying the hard real-time capable HPT. The same reason applies to the memory controller that should issue memory accesses in the same prioritised order like the RTI issues instructions. The other pipeline stages remain unchanged, besides an additional signal that passes the thread number through the pipeline, in order that the write back stage writes to the appropriate register file. Fig. 1 shows the resulting architecture.

4.1 Instruction Fetch

The execution of an instruction typically occupies one pipeline stage for only one cycle and issuing multiple instructions from one thread is only reasonable, if there are enough instructions available. Hence the number of instructions that is fetched per cycle must be equal or greater than the number of instructions that can be issued concurrently. As the number of concurrent instructions is equal to the number of pipelines, this number must be multiplied with the maximum instruction length to get the required fetch bandwidth.

Assuming a zero cycle memory latency, it takes two cycles from the decision, that a new fetch must be initiated (in the issue stage) until the arrival of the data at the instruction window (again in the issue stage). Therefore the instruction windows (IW) must be large enough to hold at least two times the fetch width. Each additional memory latency further increases the size of the IW by the fetch width. If the instruction width varies and the instructions are not aligned to the borders of the fetch words, the size must be further increased.

A concrete example with the CarCore architecture: There are two pipelines and an instruction can be 16 or 32 bits wide. Accordingly, fetching 64 bits should provide at least enough instructions for one cycle. If in cycle $t+0$ the RTI issues two instructions to the pipelines it removes at most 64 bit from the IW and recognises that it should be refilled and initiates a fetch. During cycle $t+1$ the memory is accessed and the RTI must take the next 64 bits from the IW. In cycle $t+2$ the fetched data arrives at the RTI, so the data can be directly issued to the pipelines. But 128 bits are still not enough, as TriCore instructions must only be aligned to 16 bit boundaries, consequently four instructions could cover three 64 bit words and the minimum IW size is 192 bits.

With the proposed fetch width and instruction windows size optimal execution of the highest priority thread (HPT) can be guaranteed. But what about the other threads? The HPT only fully occupies the fetch stage, if there is code that uses every pipeline in every cycle and if these instructions are of maximum length. As the evaluation shows, this is almost never the case. Whenever the IW of a thread is full, the fetch logic tries to fetch for the thread with the next highest priority. Again, the evaluation shows that empty IWs are only a minor reason for not executing a lower priority thread.

There are two possibilities to optimise the fetching: The first one called `ENOUGH` exactly counts how much instructions are in the IW and how they are mapped to pipelines. If there are enough instructions to cover two cycles, further fetches to this thread are delayed, no matter if the IW is already full or not. The `AHEAD` logic stops fetching when it recognizes a branch somewhere within the IW. This optimisation is only applicable if there is no branch prediction and if there are at least two pipeline stages between fetch and branch decision.

An example for the CarCore architecture with three stages between fetch and branch decision (RTI, decode and execute): If in cycle $t+0$ only the branch is in the IW, the RTI issues it and removes the instruction from the IW. In cycle $t+1$ the `AHEAD` logic recognises that there is no longer a branch in the IW and permits to fetch the next instruction. In cycle $t+2$ the next instruction

Algorithm 1. Policy of the Real Time Issue Stage

```
Input: number of threads $T$, number of pipelines $P$
  $thread_0$ has the highest priority, $thread_{T-1}$ the lowest
Output: assignment of instructions to pipelines in $pipeline_p$
  $pipeline_p \leftarrow \emptyset \quad \forall 0 \leq p < P$
  for $0 \leq t < T$ do
    $instr \leftarrow$ next instruction of $thread_t$
    for $0 \leq p < P$ do
      if (pipeline of $instr = p) \wedge (pipeline_p = \emptyset)$ then
        $pipeline_p \leftarrow instr$
        $instr \leftarrow$ next instruction of $thread_t$
      end if
    end for
  end for
```

word arrives at the RTI and is ready for issuing, while the branch instruction is now in the execute stage, calculating the branch target and checking the branch condition. In cycle $t + 3$ the RTI receives the signal, if the branch is taken or not and can issue the next instruction if it is not taken. The next instruction arrives even one cycle early at the RTI, but this is necessary in the CarCore architecture, as a single instruction could span two 64 bit words and then the cycle is needed for the second half of the instruction.

Both techniques save unused fetches and therefore increase the fetch bandwidth of lower priority threads without influencing the HPT performance. They are effective if the instruction length varies or only part of the pipelines are occupied within one cycle.

4.2 Real-Time Issue

The real-time issue (RTI) stage receives the fetched instructions from the fetch stage and inserts it into the instruction window of the appropriate thread. Then the instructions in the windows are analysed and instructions are assigned to pipelines. To decide, in which pipeline an instruction should be executed, the opcode contains a field, where the number of the appropriate pipeline is stored. Instructions that could be issued in parallel must be located in ascending order within the instruction stream. As long as the value of the pipeline field of the next instruction is higher than the former one, it can be issued concurrently. When the pipeline number of an instruction is lower or equal to the number of the preceding instruction, the latter instruction is the first instruction of the next cycle.

The assignment strictly depends on the priorities of every thread. Starting with the thread with the highest priority, the RTI tries to issue simultaneously as many instructions as possible. Then instructions from the thread with the second highest priority are issued, if the desired pipeline is not occupied yet. Algorithm 1 explains the issue strategy in pseudo code.

Additionally the RTI manages multicycle instructions. They are implemented as microcode sequences that can be interrupted at any position within the sequence. This interruptibility is important, otherwise low priority threads would delay higher priority threads for several cycles, once they were able to start their microcode sequence.

In this paper we assume that the priorities are fixed, but it is also possible to provide a new priority mapping from an external module in each cycle. With this technique, sophisticated scheduling algorithms for multiple hard real-time threads [3] or overlapping IPC controlled threads [4] can be implemented.

4.3 Prioritised Memory Controller

The easiest way to deal with memory accesses is to add a memory stage in the pipeline between execute and write-back stage, as it is implemented in the classical DLX pipeline [19]. But then there might be a read-after-write data dependency after a load instruction that cannot be solved by forwarding. Depending on the instruction set, the check if there is really a dependency can be difficult (for TriCore it is), therefore stalling the thread for one cycle anyway (and using the cycle for a lower priority thread) is an acceptable solution. After a store no bubble cycle must be inserted.

In TriCore the bubble cycle is avoided by calculating the address in the decode stage and accessing memory in the execute stage, but this cannot be applied here, as CarCore calculates the address in the execute stage (to achieve better stage balance, see section 3.2). If a memory access takes multiple cycles, say it has a latency of M, singlethreaded processors stall the complete pipeline until the access is completed, but this cannot be applied here, as this would prevent all other threads from being executed, even the highest priority thread.

A straight enhancement of the memory stage idea would be to add M memory stages (plus the one memory stage mentioned earlier). To avoid data dependencies, the thread must be stalled for $M + 1$ before the next instruction of the same thread might be issued. But there is one more problem: if two memory instructions of different threads are issued in successive cycles the second memory instruction will arrive at the memory controller when it is busy because of the first instruction. Consequently, after the RTI issued a memory operation, no other memory operation from any thread may be issued for M cycles.

To save the enormous hardware costs of multiple memory stages, we applied a technique called *Split Phase Load*. For a memory write the additional memory stages are not needed, as nothing must be written back, only the stalling of the threads is important. Hence only a solution for loads must be found: the load is split into two microinstructions, the address calculation and the register write back. When the RTI recognises a load instruction, it issues the first microinstruction that calculates the address in the execute stage and forwards it to the memory controller. When the memory controller receives the data it notifies the RTI and it issues the second microinstruction that writes the data from the memory to the register set in the write back stage. The notification can even be

threads	ALUTs	regs	MHz
1	14808	3857	27.17
2	21129	5968	26.10
3	27519	8061	24.38
4	31603	10125	17.55
5	39325	12195	11.10
6	45400	14271	8.52
7	49082	16378	7.00

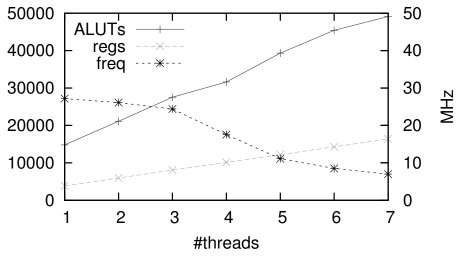


Fig. 2. CarCore hardware characteristics depending on the number of threads

sent some cycles earlier in order that the second microinstruction arrives at the write back stage at the same cycle as the data arrives from memory.

To avoid the restriction of the other threads, not to issue memory operations, *Address Buffers* are added. There is one address buffer per thread located in the memory controller. After a store or the first microinstruction of a load the thread is temporarily suspended from further issuing instructions. When the memory instruction arrives at the memory controller the address is saved in the address buffer of the appropriate thread. Whenever a memory operation is completed, the memory controller looks at the address buffers in priority order and starts a new memory operation if there is a valid entry. At the same cycle the memory controller notifies the RTI to resume issuing instructions of the thread whose data word had just arrived. Depending on the kind of instruction the RTI continues with the second microinstruction of a load or the next instruction after a store.

There is another advantage of the Split Phase Load / Address Buffer technique: the memory latency can vary and there is no upper bound. If the memory access is fast, the second microinstruction of the load is issued earlier, if it takes longer, the second microinstruction (respectively the next instruction after a store) is issued later. So multiple memories with different access times are supported.

5 Evaluation

We started our SMT enhancement with a singlethreaded SystemC model of the CarCore and enhanced it to support multithreading. The final SystemC model was translated to VHDL for FPGA synthesis. There are separate data and instruction memory buses, each 64 bits wide. In the FPGA model the memory latencies are fixed to 0 for the instruction memory (internal on-chip RAM) and 4 cycles to the off-chip data memory.

Fig. 2 shows the size of the CarCore on an Altera Stratix II EP2S180F1020C3 device. The numbers of Adaptive Lookup Tables (ALUTs) and register bits grow nearly linear with the number of threads. Each thread requires about 6000 ALUTs and 2000 registers and the base processor adds about 9000 ALUTs and 2000 registers.

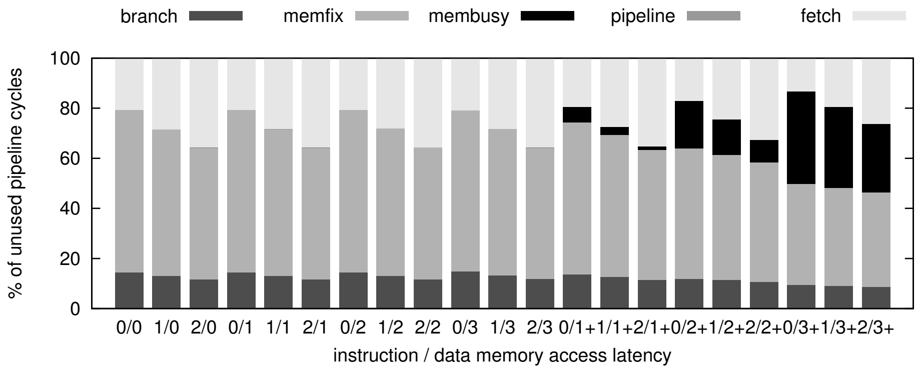


Fig. 3. Reason why the HPT cannot be issued, depending on the memory latencies

When executing multiple threads, the HPT reaches exactly 100% of its singlethreaded speed, hence a WCET analysis for a singlethreaded simplification of our architecture is also valid for the HPT in the multithreaded architecture [3]. The speed of the threads with lower priorities falls exponentially to about 50, 35 and 20 percent of singlethreaded performance (measured in Instructions Per Cycle, IPC), see [3] for a more detailed discussion.

We used the Hightec GNU C/C++ compiler for TriCore [18] to compile benchmark programs from the EEMBC AutoBech 1.1 benchmark suite [20] (`a2time`, `canrdr`, `aifirf`, `rspeed`) and the Mälardalen WCET group [21] (`crc`, `fft1`, `mm`). 1000 task-sets of 8 threads were randomly assembled from these seven benchmark and executed for one million cycles each. The given figures are the average values of these 1000 runs.

5.1 Reasons for Stalling Threads

The reasons why a thread cannot issue any instructions in a certain cycle can be divided into five classes:

branch. Fixed latency of a branch: 2 cycles
memfix. Minimum latency of a memory access: 3 cycles
membusy. Additional stall cycles, when a memory instruction cannot be executed, because the memory is busy with an operation from another thread.
pipeline. The desired pipeline is already occupied by a higher priority thread. (never applies to the HPT)
fetch. The instruction window is empty.

Fig. 3 shows the distribution of the reasons for not issuing instructions of the highest priority thread (HPT), depending on the memory latencies. The x-axis gives the reason for a delay and the numbers on the x-axis indicate the memory latencies: the first number is the latency of the instruction memory, the second one the latency for data memory.

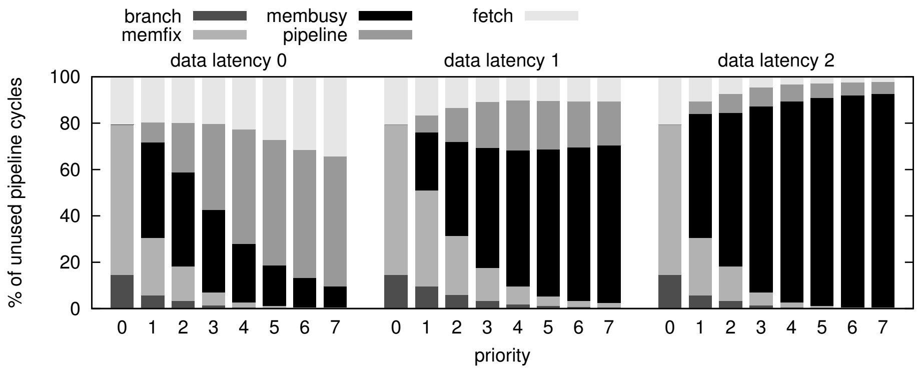


Fig. 4. Reason why a thread cannot issue an instruction, depending on its priority

Even with minimum latencies (0/0), more than 80% of unused cycles are due to memory delays (**fetch** or **memfix**). Not surprisingly the percentage is increased when the latencies are increased. Each additional cycle of instruction memory latency increases the percentage of fetch stalls by 8%, therefore a fast instruction connection (via scratchpad or instruction cache) is inevitable.

The bars marked with plus show additional **membusy** latencies. These appear, when the HPT is executed together with other threads (for the bars without a plus, the HPT was executed without concurrent threads). If a lower priority memory access takes multiple cycles and begins in the cycle preceding the cycle when a HPT memory access should start, the former occupies the memory controller for multiple cycles and therefore delays the HPT thread.

This effect violates the complete isolation (and thus the hard real-time capability of the HPT), but it can be avoided by either modifying the WCET analysis and assuming twice the memory latency or the lower priority memory accesses can be delayed when a HPT memory access is on the way through the pipeline (then the distribution of reasons is the same as in the corresponding case without the plus).

The later technique is called *Dominant Memory Access Announcing (DMAA)*: when a memory access of the HPT is recognized at the beginning of the pipeline, it is announced immediately to the memory controller that consequently delays all memory accesses until the HPT memory access arrives and can be invoked. Therefore all memory accesses between issue stage and memory controller are delayed and if the distance between them is longer than the memory latency, the HPT is never influenced by lower priority memory accesses (for details see [3]).

Applying the DMAA technique, Fig. 4 shows the average distribution of stall reasons against the priority. The fixed latencies (which dominate the stall reasons of the HPT) are less important for lower priority threads. For these threads, the influence of fetch and pipeline conflicts grows significantly. If the memory latency is more than zero (rightmost group of Fig. 4), this stall reason dominates for lower priority threads.

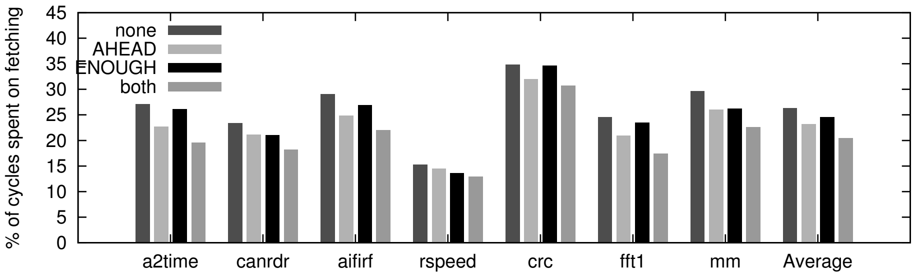


Fig. 5. Percentage of cycles that were required to fetch instructions

5.2 Instruction Fetch Optimisation

To decrease the fetch conflicts and hence increase the performance of lower priority threads, we introduced the `AHEAD` and the `ENOUGH` fetch policies. They reduce the number of fetches of the HPT without affecting its real-time behaviour. The `AHEAD` logic occupies 18 ALUTs per thread slot, `ENOUGH` 44 and both together require 60 ALUTs per slot, an acceptable size compared to 6000 ALUTs for a whole thread slot.

Fig. 5 shows the percentage of fetch cycles executing the benchmarks singlethreaded on the CarCore. The percentages vary significantly depending on the benchmark. Both optimised policies reduce the number of fetches by about 5 to 10 percent and as the average of the benchmarks shows, `ENOUGH` is on average better than `AHEAD`, but not for all. Very interesting is the combination of both policies: their sets of eliminated fetches are nearly distinct, hence it is not surprising that the numbers of eliminated fetches could nearly be added if combining both. But for some benchmarks the savings of the combination is even bigger than the sum of the single savings. Regrettably the explanation of this complicated interaction goes beyond the scope of this paper.

6 Conclusion

We explained how a singlethreaded superscalar TriCor compatible processor can be enhanced to provide SMT while still allowing the execution of one hard real-time thread with several non real-time threads concurrently in the background. The techniques described can easily be transferred to any other superscalar in-order processor. The latency of the memory is the main reason for stalling threads and thus the biggest problem of the architecture. Currently our group is integrating scratchpad memory into the CarCore, to ease this problem. First results are available in [5].

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement n° 216415 and from the Deutsche Forschungsgemeinschaft (DFG).

References

1. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In: ISCA 1995, pp. 392–403 (1995)
2. Gerosa, G., Curtis, S., D'Addeo, M., Jiang, B., Kuttanna, B., Merchant, F., Patel, B., Taufique, M., Samarchi, H.: A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-K Metal Gate CMOS. In: IEEE International Solid-State Circuits Conference (ISSCC 2008), pp. 256–611 (2008)
3. Mische, J., Uhrig, S., Kluge, F., Ungerer, T.: Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads. In: ICCD 2008, pp. 371–376 (2008)
4. Mische, J., Uhrig, S., Kluge, F., Ungerer, T.: IPC Control for Multiple Real-Time Threads on an In-order SMT Processor. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 125–139. Springer, Heidelberg (2009)
5. Metzlaff, S., Uhrig, S., Mische, J., Ungerer, T.: Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In: Proceedings of the 9th Workshop on Memory Performance (MEDEA 2008), pp. 38–45 (2008)
6. Jain, R., Hughes, C.J., Adve, S.V.: Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In: RTSS 2002, pp. 134–145 (2002)
7. Dorai, G.K., Yeung, D., Choi, S.: Optimizing SMT Processors for High Single-Thread Performance. Journal of Instruction-Level Parallelism 5 (April 2003)
8. Cazorla, F.J., Knijnenburg, P.M., Sakellariou, R., Fernndez, E., Ramirez, A., Valero, M.: Predictable Performance in SMT Processors. In: Proceedings of the 1st Conference on Computing Frontiers, pp. 433–443 (2004)
9. Yamasaki, N., Magaki, I., Itou, T.: Prioritized SMT Architecture with IPC Control Method for Real-Time Processing. In: RTAS 2007, pp. 12–21 (2007)
10. Hily, S., Seznec, A.: Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading. In: HPCA-5, pp. 64–67 (1999)
11. Zang, C., Imai, S., Frank, S., Kimura, S.: Issue Mechanism for Embedded Simultaneous Multithreading Processor. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E91-A(4), 1092–1100 (2008)
12. Moon, B.I., Yoon, H., Yun, I., Kang, S.: An In-Order SMT Architecture with Static Resource Partitoning for Consumer Applications. In: Liew, K.-M., Shen, H., See, S., Cai, W. (eds.) PDCAT 2004. LNCS, vol. 3320, pp. 539–544. Springer, Heidelberg (2004)
13. El-Moursy, A., Garg, R., Albonesi, D.H., Dwarkadas, S.: Partitioning Multi-Threaded Processors with a Large Number of Threads. In: IEEE International Symposium on Performance Analysis of Systems and Software, March 2005, pp. 112–123 (2005)
14. Raasch, S.E., Reinhardt, S.K.: The Impact of Resource Partitioning on SMT Processors. In: PACT 2003, pp. 15–25 (2003)
15. El-Haj-Mahmoud, A., AL-Zawawi, A.S., Anantaraman, A., Rotenberg, E.: Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing. In: CASES 2005, pp. 213–224 (2005)

16. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In: CASES 2008, pp. 137–146 (2008)
17. Infineon Technologies AG: TriCore 1 User's Manual. V1.3.8 (January 2008)
18. HighTec EDV-Systeme GmbH: Website, http://www.hightec-rt.com/
19. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach, 4th edn. Morgan Kaufmann Publishers Inc., San Francisco (2007)
20. Embedded Microprocessor Benchmark Consortiu: AutoBench 1.1 software benchmark data book, http://www.eembc.com/techlit/datasheets/autobench_db.pdf
21. Märdalen WCET research group: Worst Case Execution Time Bechmarks, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Complexity-Effective Rename Table Design for Rapid Speculation Recovery

Görkem Aşılıoğlu, Emine Merve Kaya, and Oğuz Ergin

TOBB University of Economics and Technology
Söğütözü Cad. No: 43 Söğütözü
Ankara/TÜRKİYE
gasilioglu@etu.edu.tr, emkaya@cis.jhu.edu, oergin@etu.edu.tr

Abstract. Register renaming is a widely used technique to remove false data dependencies in contemporary superscalar microprocessors. The register rename logic includes a mapping table that holds the physical register identifiers assigned to each architectural register. This mapping table needs to be recovered to its correct state when a branch prediction occurs. In this paper we propose a scalable rename table design that allows fast recovery on branch predictions. A FIFO scheme is applied with a distributed rename table structure that holds a variable number of checkpoints specific to each architectural register. Our results show that although the area of the rename table is increased, it is possible to recover from a branch misprediction in at worst 2 cycles.

Keywords: Register renaming, branch prediction, rename table, speculation recovery.

1 Introduction

Superscalar microprocessors use aggressive techniques like out-of-order execution and dynamic scheduling in order to boost performance. Data and control dependencies between the instructions are the major limiting factor on performance during the processor design process. Some data dependencies are in fact not real but result from the lack of architectural registers at compile time and hence are called the "false data dependencies". In order to remove these false data dependencies and reduce the number of stalls in the pipeline, many of the contemporary microprocessors employ the technique called "register renaming".

In register renaming, a new physical register is assigned for each instruction that produces a result. A register alias table (RAT) is used in the rename stage of the pipeline to keep track of the assignments between the architectural registers and the physical registers. Usually this table (also called the "rename table") is implemented by using an entry for each architectural register. The rename table holds the precise state of the processor and has to be recovered in the case of an unexpected event, such as a branch mispredicton or exception. Since the time required for restoring the contents of the table add to the branch misprediction penalty, it is crucial that the table is recovered as soon as possible. Different schemes exist in the literature to rollback the branch speculation, of which the most used techniques include checkpointing the rename table, walking forward and backwards on the reorder buffer or waiting until the commit time of the mispredicted branch instruction [2][9]. Each of these schemes have various pros and cons in terms of circuit complexity and misprediction penalty.

In this paper we propose a new implementation of the rename table that allows fast recovery on a branch misprediction. We propose to use a FIFO structure for each architectural register and recover the tail pointers when a branch misprediction occurs. Our design is less complex than taking a full checkpoint of the rename table, and has smaller restoration latency when compared to the schemes that walk through the reorder buffer.

2 Register Renaming

Register renaming requires a new physical register to be assigned to each and every result producing instruction. Fig. 1 shows an example of how the register renaming is performed. Note that R1 is overwritten by the second ADD instruction which is in fact not dependent on the first ADD. This false dependency is removed by assigning different physical registers to two ADD instructions.

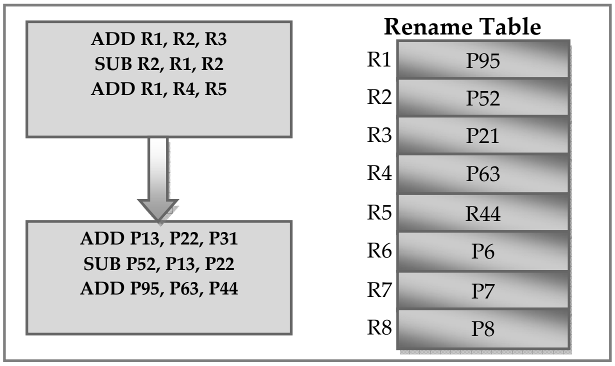


Fig. 1. Example of Register Renaming

The second ADD instruction in Fig. 1 removes its write after write (WAW) and write after read (WAR) dependencies by writing its result to a separate physical register. However, this is not enough to maintain precise execution; any younger instruction that needs to access R1 has to be aware of the fact that the value it needs to read resides in P95. The rename table is used for this purpose; the table includes an entry for each architectural register that holds the location of the most recent instance of the register. Consequently, as the rename table holds the precise state of the processor, any fault on this table will result in a system crash.

Ideally the rename table idea works perfectly. Each instruction that produces a result checks the availability of a physical register and allocates a register. Afterwards, the result producing instruction updates the rename table to inform the consumers that will later check this table to obtain the location of their architectural source registers. In superscalar machines, multiple instructions need to be renamed each cycle, which mandates multiple updates to the rename table in the same cycle. Multiple updates to the same entry of the rename table are avoided by using a series of comparator circuits to check for dependencies.

3 Speculation Recovery in the Rename Table

Although the idea of rename table works fine in regular conditions, some recovery effort is needed when the processor employs some kind of speculation. All of the contemporary microprocessors employ branch prediction to alleviate the negative effects of deep pipelining on processor performance [3][5]. When a branch misprediction occurs, all of the instructions following the branch instruction in the reorder buffer are squashed from the issue queue and the reorder buffer. However this is not enough to recover the precise state as the modifications on the rename table also have to be rolled back to its state just before the faulting branch instruction.

Different schemes exist in the literature and are implemented in real processors to recover the rename table on a branch misprediction. This schemes can be divided into two groups depending on their dependence on a retirement rename table (also called the commit rename table).

3.1 Schemes That Use a Separate Commit Rename Table

Commit rename table is used to hold the locations of the last committed instances of the architectural registers. Its structure is the same with the speculative rename table; contains one entry for each architectural register. Every instruction that leaves the reorder buffer at the commit stage updates this table with its destination physical register tag.

1. ***Wait:*** It is always the safe choice to wait until the faulting branch reaches the top of the reorder buffer. When the branch commits, the commit rename table is copied to the frontend (speculative) rename table. This scheme is not preferred for many branches since a large number of cycles may pass before the branch itself commits.
2. ***Walk forward:*** When the misprediction occurs, start from the head of the reorder buffer and pseudo commit all instructions to construct the commit rename table. When the faulting branch is reached, the commit rename table is precise. Afterwards the commit rename table is copied to the frontend rename table. Usually the number of pseudo committed instructions is equal to the commit width of the machine. If the faulting branch is far away from the head of the reorder buffer, branch misprediction penalty increases.

3.2 Schemes without a Separate Commit Rename Table

3. ***Walk backward:*** When a misprediction occurs, start from the tail of the reorder buffer and undo the changes of each instruction on the speculative rename table. The number of instructions processed each cycle is equal to the commit width of the machine. If the faulting branch is far away from the tail of the reorder buffer, the misprediction penalty is high. This scheme also requires that every instruction

holds the previous mapping of the corresponding architectural register in its reorder buffer entry.

4. ***Checkpointing:*** The circuitry of the rename table is modified to include some shadow copies of the rename table. Whenever a branch instruction passes the rename stage, a copy of the rename table is created. If the branch is mispredicted, the corresponding checkpoint is copied to the speculative rename table. This scheme enables the recovery of the rename table state in a single cycle. However the circuit level complexity limits the number of checkpoints that can be implemented given a clock frequency. Also, the limited number of checkpoints limits the number of branch instructions that can be inside the reorder buffer of the processor since a branch cannot proceed if a free checkpoint is not available when it arrives at the rename stage.

   It is previously shown that walking backwards on the reorder buffer outperforms the other schemes (except for the checkpointing scheme which has its own circuit level difficulties) in terms of instructions per cycle for spec 2000 benchmarks [2].

4 Proposed Rename Table Structure

During a processor design process, it is desirable to keep the number of cycles required to recover the rename table at minimum with minimum circuit level complexity. In other words, a processor designer would want the performance of checkpointing with a very simple circuit. For this purpose, we propose a new rename table structure that is capable of storing the history of physical register assignments for each architectural register.

Fig. 2 shows the proposed rename table structure. A separate circular FIFO queue is used for each architectural register to store the physical register assignments. Whenever a new instruction that targets an architectural register as destination arrives at the rename stage and allocates an available free physical register, the tag of the allocated register is inserted into the corresponding FIFO queue by using the corresponding tail pointer. The tail pointer always shows the most recent instance of the corresponding architectural register. Subsequent dependent instructions always read their source locations from the registers pointed by the tail pointers.

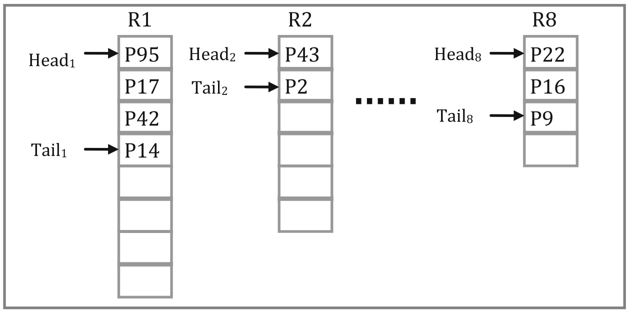


Fig. 2. Proposed Rename Table Structure

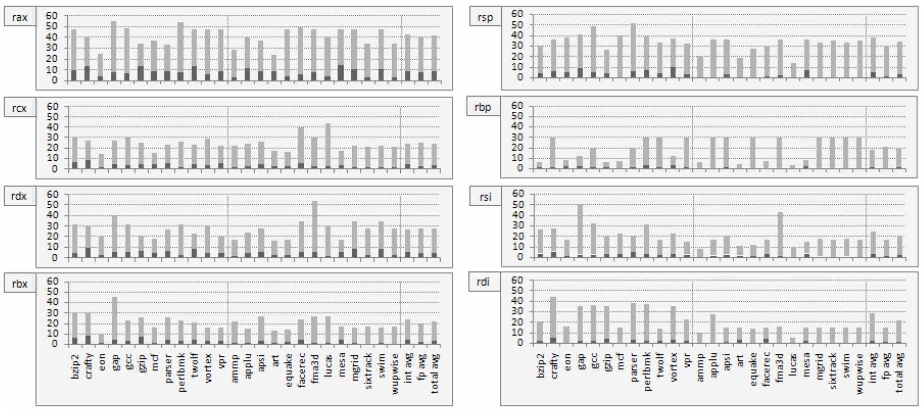


Fig. 3. Amount of register usage in the Spec2000 benchmark suite, light area represents maximum usage, dark area represents average usage

Conditions for vacating entries inside the FIFO queues vary according to the implementation of register renaming inside the processor. In P6 architecture the reorder buffer entries also serve as physical registers and a separate architectural register file exists to hold the architectural state. Therefore the head pointer is updated when an instruction that targets the corresponding architectural register commits. On the other hand, in architectures, which use a unified register file that holds both the physical and architectural state, such as the Intel's Pentium 4 [3], Alpha 21264 [5] and MIPS R10000 [7], a physical register is released only when an instruction that renames the same architectural register commits. In the proposed rename structure, the head pointer of the corresponding architectural register is updated (incremented by one unless the pointer is at the end of the buffer) whenever a physical register that holds an instance of the architectural register is released.

Since each architectural register has its own circular FIFO buffer in the proposed structure, the number of entries in each FIFO queue may vary. The number of entries for each architectural register can be determined by observing the common behavior of programs and compilers. If an instruction targets an architectural register and the corresponding FIFO queue does not contain any available entries, the pipeline stalls and the frontend waits until an entry is available for the instruction. In order to minimize the performance degradation due to the proposed rename structure, the FIFO queues have to be sized appropriately, so that the pipeline does not get stalled frequently. Fig. 3 shows the average and maximum number of renamed instances of each general purpose architectural register in the simulated x86 architecture. As the results reveal, some of the registers are employed more than the others. For example, register rax has the most concurrent instances on average. This result shows that a larger FIFO queue has to be used for rax.

The number of entries in each FIFO queue can be at most equal to (the number of physical registers – the number of architectural registers). This is because of the fact that each architectural register has to have a physical location and the same physical register cannot be assigned to two different architectural registers at the same time.

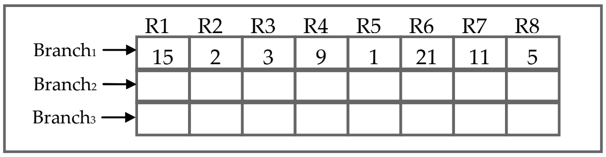
5 Recovering from Branch Mispredictions

The proposed structure allows rapid recovery of the rename table as all of the speculative register assignments are available in direct mapped SRAM bitcell array. In the case of a branch misprediction, fixing the tail pointers is enough to recover the rename table. Different schemes can be employed by using the proposed rename table structure to rollback the speculation on a branch instruction. For a processor that waits until the faulting branch reaches the top of the rename table, there is no need to keep a commit rename table since when the branch reaches the top of the ROB, the head pointers for each architectural register will point to the precise state. In fact, the head pointers in the proposed structure form the commit rename table when they are used together which alleviates the need to store a separate commit rename table as implemented in some modern microprocessors.

Walk backward and walk forward schemes get easier to implement with the proposed design since nothing in the rename table is overwritten during the rename process. During the walk backward operation, the processor just decrements the corresponding tail pointer of the architectural register that is targeted by the squashed instruction. When the faulting branch instruction is reached, the tail pointers of all FIFO queues are restored.

Checkpointing becomes simpler at the circuit level by using the proposed rename table design. Regular checkpointing requires a shadow copy taken at each branch instruction. This is accomplished by implementing each bit of the rename table as a shift register. Since the number of checkpoints limits the number of branch instructions that can reside inside the processor concurrently, having more checkpoints is desirable. However, as the number of checkpoints increases, the logic depth for an individual shift register increases, which results in a higher rename table latency and limits the frequency of the processor. The use of the proposed scheme allows implementing the checkpointing scheme more easily without any need for shift registers. Instead of checkpointing the entire table on each branch, the tail pointers for each architectural register are stored in a table whenever a branch instruction arrives at the rename stage. This table is indexed by the branch identifiers and has to be implemented as a circular FIFO queue since nested branch instructions may require squashing multiple branch instructions in program order.

Fig. 4 shows the structure of the checkpoint table used to store the information for each branch instruction. For each entry, a tail pointer is stored for each architectural register. Since the size of each FIFO queue can be at most equal to the number of physical registers, each stored tail pointer can be represented with $\log_2$(number of physical registers). For a processor with 256 registers, tail pointers are 8-bits long, resulting in 64 bits for each entry in the checkpoint table. Therefore the latency of this structure is similar to the register file itself depending on the number of branches allowed inside the processor. Whenever the outcome of a branch is mispredicted, the processor accesses the checkpoint table with branch index and reads the stored tail pointers. The tail pointer registers of the rename table are overwritten by the tail pointer values read from the checkpoint table in order to recover the rename table to the cycle just before the faulting branch instruction. Depending on the circuit level implementation and the clock frequency of the processor, restoring the rename table may take one or two cycles. While it can be possible to fix the tail pointers in the



	R1	R2	R3	R4	R5	R6	R7	R8
Branch1 →	15	2	3	9	1	21	11	5
Branch2 →								
Branch3 →								

Fig. 4. The Checkpoint Table used for branch instructions

same cycle with the reading of information from the checkpoint table, for a pipelined implementation, the processor may read the tail pointers in the first cycle and restore the rename table in the second cycle.

6 Hardware Implementation

The rename tables, like any regular memory structure used in contemporary microprocessors, are implemented by employing SRAM bitcells. While in the architectures that make use of waiting or walking forward/backward schemes plain SRAM bitcells are enough for implementation, each bitcell is needed to have a shift register capability in order to use checkpoints inside the rename table. If there are 16 checkpoints allowed inside the processor each bit of the rename table needs to be a 16-bit shift register.

Our proposed rename table scheme removes the shifting complexity at bitcell level and implements all of the FIFO queues with plain SRAM bitcells. Each FIFO queue is in fact a small payload RAM that has the same bit-width as a rename table but has a larger number of entries.

In the baseline 4-way architecture the rename table must have 4 write ports (in case 4 instructions rename different architectural registers) and 8 read ports (in case each instruction has a different set of architectural source registers). This structure is accessed after the dependency checking logic detects any possible multiple renames in a single cycle. Our solution does not alter the dependency checking logic but offers a different storage mechanism for the register mappings.

In an architecture with N architectural registers, our proposed scheme mandates the use of N FIFO tables that are made of SRAM bitcells. Each table is a regular payload RAM with regular decoder circuits that allow random access. However the inputs to these decoders are wired to the tail pointer register of the corresponding FIFO instead of taking the values from incoming instructions. The instructions that are renamed only write their tags though the regular bitlines to the entry (or entries) automatically selected by the tail pointer register. Note that although, in a 4-way machine, up to 4 values can be written to a FIFO in the same cycle, only one value is read. Therefore each FIFO structure needs 4 write ports and 1 read port. Also, it should be noted that although the structure can be accessed randomly like a regular register file, the access is not random; at each cycle the value that is pointed by the tail pointer is read and if the corresponding architectural register is renamed up to 4 values are written to the following entries. The tail pointer is updated after the new register assignments are written. All of the FIFO queues are circular buffers. Therefore the tail pointer points to the top of the structure after reaching the bottom.

The head pointers of each FIFO queues are also separate registers that are updated from the commit stage of the pipeline. When an instruction is committed and removed from the reorder buffer it simply increments the head pointer of its destination architectural register. Note that instructions do not access the payload area from the commit stage but they only access the head pointer register.

Processor is stalled at the rename stage if the contents of the head pointer register is just one over the contents of the tail pointer register for an architectural register that needs to be renamed. Since these registers are just incremented unless there is an exception (such as a branch misprediction), they are better be implemented as a counter with parallel data loading capability.

The checkpoint table that is needed to recover the contents of the pointer register in case of an exception or misprediction is a plain payload structure that only holds the values of the registers just at the time a branch reaches the rename stage. That structure is implemented by using SRAM bitcells and it is smaller than a register file which makes it possible to access this structure in less than a cycle.

Complexity is moved to the pointer registers in the proposed scheme since these registers are actually counters but they also need a logic for immediate data loading and comparison of head and tail pointers. There is also a control logic that allows the reading and writing of values by using only the contents of the tail pointer.

7 Results and Discussions

In order to get an accurate idea of the performance impact of the proposed technique we used the PTLsim simulator [8] that is capable of simulating x86 instructions. We ran the spec 2000 benchmarks for 1 billion committed instructions. Table 1 shows the simulated processor configuration.

The simulation was run with checkpointing and walk-backwards recovery mechanisms already included in PTLsim. The FIFO tests were based on a modification of the checkpointing algorithm which imposed penalties to the cycle count of the simulation when the allocated queue was filled, and performed similar to checkpointing when the queues had available space.

Table 1. Simulation Parameters

Parameter	Configuration
Machine width	4-wide fetch, 4-wide issue, 4-wide commit
Window size	32 entry issue queue, 48 entry load queue, 32 entry store queue, 128–entry ROB
Function Units	Integer ALU (2), Load unit (2), FPU (2), Store Unit (2)
Latencies	Integer ALU (1), ALU Multiplication (4), ALU Bit scans(3), ALU Division (32), FPU Arithmetic (6), FPU Vector Arithmetic (1)
L1 I-Cache	32 KB, 4-way set-associative, 64 byte line, 1 cycle hit time
L1 D-Cache	16 KB, 4–way set–associative, 64 byte line, 1 cycle hit time
L2 Unified Cache	256 KB , 16–way set–associative, 64 byte line, 6 cycles hit time
L3 Unified Cache	4 MB, 32–way set–associative, 64 byte line, 16 cycles hit time
Branch Predictor	Meta predictor using bimodal and two-way
Memory	140 cycles hit time

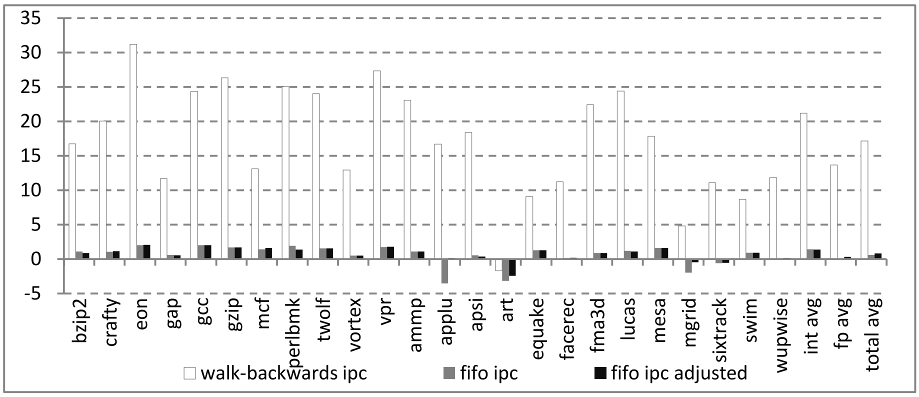


Fig. 5. Performance comparsion of the proposed scheme against regular checkpointing and walk-backwards scheme

Fig. 5 shows the performance comparison of the proposed renaming scheme when compared against regular checkpointing. The results for the walk-back scheme is also shown on the graph. The y-axis of the graph is the percent decrease of IPC. Zero on the y-axis means that the method has the same IPC as checkpointing. For the proposed scheme there are two graphs: one labeled as fifo ipc and one labeled as fifo ipc adjusted. For the first bar, we used a constant FIFO size of 16 for each architectural register, whereas for the adjusted bar we adjusted the size of the FIFO queues of each architectural register according to the average and maximum values shown in Fig. 3. In the latter case, the sizes of the FIFO queue were set to numbers between 16 and 32. We assumed a recovery time of 2 cycles for the proposed scheme although it may be possible to recover in one cycle depending on the circuit level implementation.

Results in Fig. 5 show that checkpointing always outperforms walking through the reorder buffer and the proposed scheme almost meets the performance of checkpointing although it limits the number of instances of the architectural registers that can be present inside the processor. Interestingly increasing the sizes of the FIFO queues did not provide any significant performance benefits.

8 Conclusion and Future Work

In this paper we proposed a FIFO-queue-based rename table design that is scalable and allows faster and simple branch misprediction recovery. Each architectural register is assigned a FIFO queue that holds every speculative physical register assignment. By using another checkpoint table that holds the tail pointers for each queue, it is possible to recover the rename table in a single cycle by just using the regular SRAM bitcells rather than shift registers. Our design simplifies the circuits used for constructing the rename table especially for processors that employ checkpointing.

If implementing the regular checkpointing scheme is feasible in terms of circuit complexity, proposed scheme is not a good alternative; using the regular checkpointing is a better choice. However, the proposed renaming scheme offers the power of checkpointing when the checkpointing is not feasible at the circuit level. This is a result of the fact that our new scheme decouples the rename table recovery from the number of instructions whose modifications on the rename table needs to be undone. The proposed checkpointing scheme also offers the use of the speculative rename table and the commit rename table together in a single structure.

In the future, we plan to remove the rename table all together and apply the proposed scheme to the register file. This will result in a distributed register file structure with one bank of registers for each architectural register. In this future case, there won't be any need for a rename table or a free list as the physical register identifier for a free register request will come only from a single source (the FIFO of the corresponding architectural register) and the instructions that need to read the physical register mapping of their source architectural registers will read the tail pointer of the corresponding FIFO queue.

Acknowledgments

This work was supported by the Scientific and Technological Research Council of Turkey (TUBITAK) through the research grants 107E043 and 109E043.

References

1. Burger, D., Austin, T.M.: The SimpleScalar tool set: Version 2.0., Technical Report, Det. of CS, Univ. of Wisconsin-Madison (June 1997)
2. Akkary, H., et al.: Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In: IEEE/ACM International Symposium on Microarchitecture, MICRO (2003)
3. Hinton, G., et al.: The Microarchitecture of the Pentium 4 Processor. Intel Technology Journal 5(1) (February 2001)
4. Hwu, W.W., Patt, Y.N.: Checkpoint repair for out-of order execution machines. In: 14th International Symposium on Computer Architecture (ISCA), June 1987, pp. 18–26 (1987)
5. Kessler, R.E.: The Alpha 21264 Microprocessor. IEEE Micro 19(2), 24–36 (1999)
6. Sima, D.: The Design Space of Register Renaming Techniques. IEEE Micro 20(5), 70–83 (2000)
7. Yeager, K.: The MIPS R10000 superscalar microprocessor. IEEE Micro 16(2), 28–40 (1996)
8. Yourst, M.T.: Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: IEEE Symposium on Performance Analysis of Systems and Software, ISPASS (2007)
9. Aasarai, K., Moshovos, A.: Toward a Viable, Out-of-Order Soft Core: Copy-Free, Checkpointed Register Renaming. In: IEEE International Conference on Field Programmable Logic (August 2009)

An Embedded GC Module with Support for Multiple Mutators and Weak References

Thomas B. Preußer, Peter Reichel, and Rainer G. Spallek

Institut für Technische Informatik, Technische Universität Dresden, Germany
{thomas.preusser,rainer.spallek}@tu-dresden.de,
peter@peterreichel.info

Abstract. This paper describes the design of a garbage collection (GC) module, which introduces modern GC features to the domain of embedded implementations. It supports weak references and feeds reference queues. Its architecture allows multiple concurrent mutators operating on the shared managed memory. The garbage collection is exact and fully concurrent. It combines a distributed root marking with a centralized heap scan of the managed memory. It features a novel mark-and-copy GC strategy on a segmented memory, thereby overcoming the tremendous space overhead of two-space copying and the compaction race of mark-and-compact approaches. The proposed GC architecture has been practically implemented and proven using the embedded bytecode processor SHAP as a sample testbed. The synthesis results for settings up to three SHAP mutator cores are given and online functional measurements are presented.

1 Introduction

The automatic reclamation of memory space no longer in use is an essential feature of modern software platforms. Widely known as garbage collection (GC), it completely drains a prominent source of programming errors and dramatically increases programmer productivity. Despites its inherent costs in processing time and memory bandwidth, it has even been adopted in the embedded platform domain. Also there, the increasing system complexity calls for higher programmer productivity and increased product confidence.

In addition to general time-to-market and design cost requirements, a few additional design constraints are of prominent importance especially in the embedded platform domain. Embedded systems should be frugal and just offer the needed performance with the smallest feasible silicon and the least possible power consumption. However, the specified performance is also often required strongly to be delivered so that services are guaranteed to meet their deadlines. These aspects motivate the designated and optimized implementation of essential functionalities despites of the additional design effort. The garbage collection is one example for such an optimization target, which has already been approached by several research groups.

While garbage collection automates the management of one essential system resource, the heap memory; many other resources (files, locks, queues, sockets etc.)

remain subject to the careful manual management by the programmer. Implementors of the objects encapsulating such resources often seek to secure the proper release of these resources as by the help of the automated memory management. The drawbacks of the traditional concept of the finalizer method (especially the fact that it can be circumvented by simple method overriding) has helped to establish the concept of reference queues. They provide a notification mechanism relying on proxy objects, which establish a weak reference to their targets that do not prevent their garbage collection.

Besides the advantages over classical finalizers, weak references even without associated reference queues enable new flavors of reachability. These can be used to mark certain associations as *nice to have but not critical.* This allows the caching of larger reconstructible data while still allowing the GC to reclaim the occupied memory whenever needed.

Both finalization and weak references enjoy thorough support in the Java2 Standard Edition (J2SE). Weak references even come in three flavors: soft, weak and phantom, which support caching as well as pre-finalization and post-finalization notification. In the Java2 Micro Edition (J2ME), only the Connected Device Configuration (CDC) requires the same set of features. The Connected Limited Device Configuration (CLDC) for more constrained devices only knows about the single weak flavor of references and adopted it only with version 1.1. Finalization is avoided altogether. Thus, it is not at all suprising that, so far, hardware-assisted GC implementations have not supported any of these enhanced features.

This paper proposes a designated concurrent hardware GC architecture that supports weak and soft references as well as reference enqueuing as known from the J2SE. Without finalization support, the phantom references collapse semantically with the weak ones[1]. This makes them obsolete in a CLDC setting.

The proposed GC implements a copying approach within a segmented memory so as to reduce the 100 percent space overhead of classical copying GC significantly. It further employs designated tap units on the internal mutator components to collect the root reference set without explicit mutator assistance.

In the remainder of this paper, Sec. 2 gives an overview on the previous work undertaken on hardware-assisted GC implementations. Sec. 3 describes the proposed GC architecture. Its practical implementation is evaluated by Sec. 4 based on its integration into SHAP [1]. Sec. 5, finally, concludes this paper.

2 Related Work

Systems with minimal hardware support for their GC implementations were built already in the late 1970s and early 1980s as native LISP machines [2,3],

[1] In contrast to the other flavors of weak references, the Java API particularity requires the *unretrievable* target of a phantom reference to be cleared programmatically in order to become *unreachable* in spite of an alive phantom reference to it. This particularity appears to stem from the avoidance of a software patent (`http://forums.sun.com/thread.jspa?threadID=5230019`). It, in fact, prevents a complete semantic blend of weak and phantom references, which is, however, of no conceptional importance.

which featured type-tagged memory words to identify pointers and hardware-implemented read and write barriers.

The later Garbage Collected Memory Module by Nilsen et. al [4,5] implements an object view upon the memory space. Upon request from the mutator, which needs to provide a list of the root objects, it employs a copying approach for an independent garbage collection. Although well documented, this system has never been prototyped [6].

Srisa-an et. al [7] describe a mark-and-sweep accelarator backed by allocation bit vectors. Additional 3-bit reference counting within similar bit vectors is suggested for the fast reclamation of low-fanin objects. The scalability of this approach is achieved by the caching of partial vectors as directed by an undetailed software component. Also, this architecture was merely simulated as C++ model and has not been included in any synthesized design.

The Komodo [8] architecture and its successor jamuth [9] support a very fine-grained, low-cost thread parallelism on instruction level. This architecture enables the non-intrusive execution of background system threads, one of which can be designated to the garbage collection. Different implementations of mark-and-sweep algorithms based on Dijkstra's tri-color marking are described.

Meyer [10] described a RISC architecture, which features separate register sets and storage areas for pointers and other data. This enables a clear distinction among these data types as required for an exact garbage collection. A micro-programmed co-processor, finally, implements the actual GC with a copying approach [11]. The required read barrier is later backed by designated hardware so as to reduce its latency significantly [6].

Gruian and Salcic [12] describe a hardware GC for the Java Optimized Processor (JOP) [13]. The implemented mark-and-compact algorithm requires the mutator to provide the initial set of root references. As objects may be moved during the compaction phase, read and write accesses to them must be secured by appropriate locks. The mutator may otherwise proceed with its computation concurrently to the garbage collection. Reference and data fields inside heap objects are distinguished by appropriate layout information associated with each object's type. Root references on the stack are determined conservatively only excluding data words that do not resemble valid reference handles.

In summary, several embedded GC implementations built on designated architectural resources and may thus be called hardware-assisted. The controlling algorithms are typically implemented in software albeit regularly on a level as low as microcode. The implementations are frugal and do not provide any features beyond the plain automatic memory management. Thus, these systems are obsolete even for the implementation of the current CLDC 1.1.

3 Garbage Collector Design

3.1 Fundamental Design

The desired GC architecture should provide several functional features. First of all, it is to abstract the memory to a managed object heap with high-level object

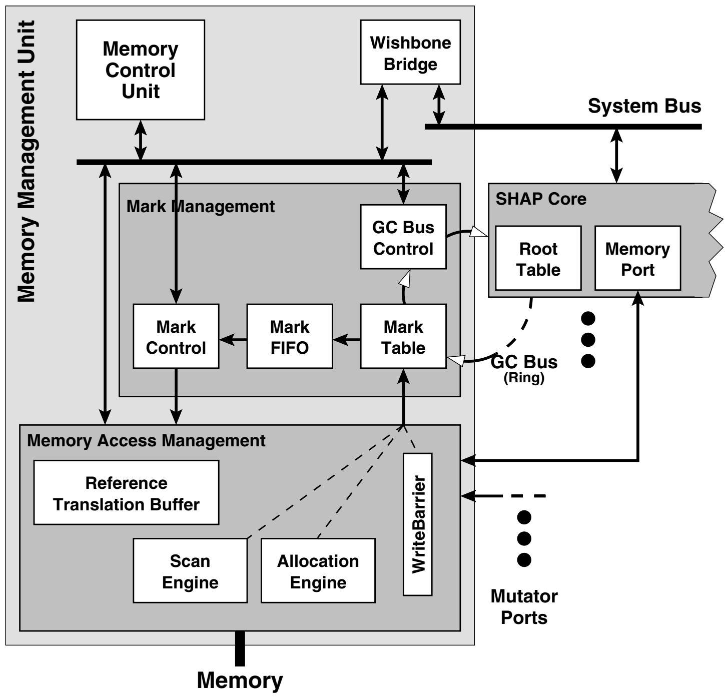


Fig. 1. Memory Management Unit: Structural Overview

creation and field access operations. It is further to support weak references as defined for the CLDC 1.1. Finally, it should be capable to serve multiple mutators.

The GC task obviously grows more complex with the support of weak references. In order to keep its implementation maintainable and extensible, the FSM implementation of SHAP's former memory manager [14] was abolished. Instead, several options for software-programmed solutions were explored. The obvious solution to simply duplicate SHAP itself was soon dismissed as its high-level Java programming greatly relies on the memory object abstraction that first needs to be established by this component. The continuous access to low-level memory would further require a unnatural if not abusive use of the language. Last but not least, the microcode implementation used by SHAP would establish a significant overhead when compared to the still rather compact memory management task. A reduced variant duplicating only the core microcode engine, finally, disqualifies due to the low achieved gain in abstraction level. Thus, the search was narrowed to compact RISC cores with a functional C toolchain, specifically, to the OpenFIRE [15] and the ZPU [16].

Both of these cores are freely available but quite contrary in philosophy. While the OpenFIRE is a full-fledged RISC engine, the ZPU takes a very frugal approach. It turned out that the ZPU with a few replacements of emulated by implemented instructions was well-sufficient for the task of memory management. It further

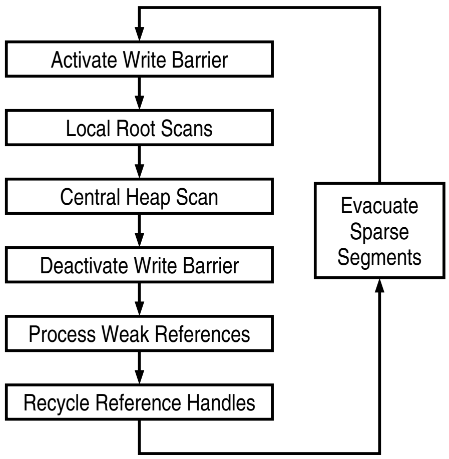


Fig. 2. GC Cycle Overview

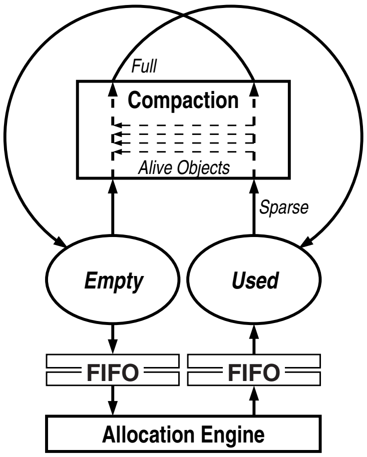


Fig. 3. Segment Life Cycle

provides the strong advantages of a concise design structure, lower resource demand and a high achieved clock frequency. In fact, an OpenFIRE solution would require a second slower clock domain or the reduction of the overall system clock by 40%. Consequently, it was the ZPU microarchitecture, which we chose to be at the heart of our memory management.

As illustrated in Fig. 1, the ZPU assumes the central control of the memory management as memory control unit (MCU). It is surrounded by several special-purpose hardware components implementing time-critical subtasks. The connection to the system Wishbone bus enables the slow administrative communication with the mutator cores. This not only serves the communication of statistics data but is also used for delivering weak reference proxies to the runtime system for their possible enqueuing into reference queues so that our design supports this feature in addition to the CLDC requirements.

The memory access of the mutator cores are prioritized over GC-related accesses. The garbage collector, thus, operates on cycle stealing, a very fine-grained utilization of otherwise idle memory bandwidth.

The overall GC cycle is summarized in Fig. 2. It is initiated and supervised by the MCU. Individual tasks are, however, backed by dedicated hardware components containing small specialized state machines.

3.2 GC Strategy

The organization of the heap memory must enable both the prompt allocation of objects as well as a steady recycling of unused memory. We chose a simple yet efficient bump-pointer allocation scheme. While it guarantees an instant allocation, it also requires the compaction of the used memory as to re-generate the continuous allocation region.

The classical compaction approaches are two-space copying [17] and mark-and-compact, which is used by JOP's hardware-assisted GC [12]. We, instead, propose a novel mark-and-copy approach operating on a segmented memory. This approach avoids both the tremendous space overhead of two-space copying as well as the compaction race of a concurrent mark-and-compact where each allocation shrinks the allocation area while also adding to the compaction work.

The segment life cycle depicted in Fig. 3 is managed by the MCU. Designated hardware modules assist the allocation and the compaction and operate on segments assigned by the MCU. The time-critical allocation is, thus, fully decoupled from the MCU, and even the exchange of the allocation segment is buffered by FIFOs. The compaction process, on the other hand, is mostly controlled by the MCU itself. Merely, the object movement is implemented as hardware service provided by the memory access management. This low-level integration enables transparent mutator access even to objects being copied.

The MCU regularly initiates heap scans to detect dead objects. While their references handles are recycled immediately, their occupied memory is only marked as such. Segment utilization statistics are maintained to identify sparsely used segments whose surviving objects are evacuated into an evacuation segment before the segment is returned to the pool of empty ones. Similar to the allocation segment, the evacuation segment is populated compactly using bump-pointer allocation and is only exchanged upon the first unsuccessful migration.

This proposed algorithm enables continuous allocation and concurrent garbage collection. A race between allocation and collection has been avoided as both are operating in distinct segments. The copying effort is reduced to surviving objects co-residing with garbage in the same segment. Segments with only short-lived operational objects are freed as a whole without any copying work. Segments with accumulated old long-lived objects will remain untouched. In addition to the spontaneous formation of object generations, the collector can accelerate this trend by the use of generational evacuation segments.

The critical parameter of the proposed approach is the segment size. Firstly, it restricts the size of the largest allocatable object. Secondly, small segments increase the management overhead in terms of segment exchanges and state information. On the other hand, large segments force a coarse-grain memory management with a potentially significant space overhead approaching the behavior of a copying collector. Hence, a set of well over 4 segments should be, at least, available.

Having decided for a moving GC, measures must be taken to ensure the stable identification of each object throughout its life cycle even in the possibility of its displacement. This is achieved through fully-transparent handles, which are the only identifications of objects ever known to a mutator. The memory manager internally maps a handle to a state record comprising the current storage location of the referenced object, the sizes of its reference and data areas as well as some GC information.

3.3 Exact Garbage Collection

Targetting the employment in constraint embedded devices, it is essential to provide an *exact* GC. A *conservative* collection is simply no option as its impreciseness would have to be paid for with additional usually spare memory resources.

The exact garbage collection must be capable of a clear distinction between references and primitive data. This may be achieved either by the direct tagging of data words or by administrative metadata. The latter option is the more economical on the heap where only instances of well-defined class layouts reside. Adopting the bidirectional class layout of the SableVM [18] further allows to reduce the required metadata to the sizes of the primitive storage growing with positive offsets and of the reference storage growing with negative offsets from the object's base address.

Our GC architecture features distributed tapping modules to collect the root set of references from the mutators. These must also be able to identify references precisely within the tapped storage elements. In our adaptation for SHAP, we chose the tagging approach for the internal stack and registers as the data contained therein does not follow a only handful of structural blueprints.

3.4 Object Graph Marking

Our architecture builds upon a decentralized collection of the root set. Each mutator core is extended by its own root scan unit, which collects the references contained in the local registers and stack into a mark table. As shown in Fig. 1, all mutator cores are arranged in a ring on a GC bus, which is mastered by the MMU. This unit issues commands onto the ring and receives their acknowledgements as they return on the other end. This ring also serves the gradual merger of the contents of the individual root tables into the global root set. While the MMU transmits an empty table, each core `OR`s the received table with its own table contents before forwarding it along the ring. The final result is entered into the central mark table inside the MMU.

The local root scans operate concurrently to their associated mutator core. Once finished, they are deactivated for the remainder of the GC cycle so that further modifications of the local root sets are not logged. This is safe as the mutators only gain access to other references by loading them from objects – which were, thus, reachable from the original root set – or by creating new ones. Objects allocated during the GC cycle are already implicitly marked by the allocation engine.

After the completion of all root scans, the actual computation of the reachable subgraph is performed by the heap scan. Our optimized variant of the well-established tri-color marking uses the mark table to tell reached from unreached objects. The colors *red* and *green* further distinguish the reachable objects into scanned and unscanned ones. The meaning of these colors alternates and is determined by the color of the active GC cycle. Initially, all references are unscanned and have the color opposite to the current cycle. Upon being scanned, a reference assumes the cycle color. References to newly-allocated objects are immediately assigned the active color as they do not contain valid references and need not be

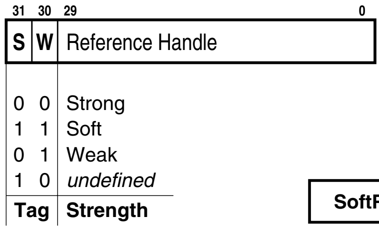


Fig. 4. Reference Strength Tags

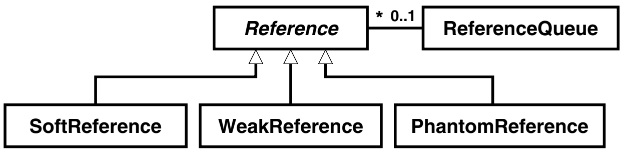


Fig. 5. Hierarchy of Reference Proxy Classes

scanned. The heap scan is completed when all reachable and, thus, marked references have assumed the cycle color. After disposing of the remaining unmarked and, thus, unreachable references, the meaning of the colors is exchanged and the initial condition that all references are of the opposite color is re-established for the next GC cycle.

While the completion of the scan could be detected by another walk through the mark table that does not produce any marked reference without cycle color, we implemented a simple but effective optimization. Each time a previously unmarked reference is marked in the mark table, it is also copied into a mark FIFO. This FIFO is read to determine objects that still have to be scanned during the heap scan. Only when it runs empty, the mark table is re-walked to search for additional work but only if the walk just completed did produce a FIFO overflow. As all work has been safely finished when the FIFO sufficed, a final unproductive walk of the table is no longer needed.

The heap scan must account for conccurent modifications of the object graph by the mutators. In particular, it must be ensured that no mutator ever gets hold of a reference that then remains unmarked. This situation might occur when a reference field of a still unscanned object is read and overwritten by a different value before it is scanned. To keep the effected subgraph safely alive, a write barrier is implemented to intercept reference writes and to enter the overwritten references into the mark table.

In the context of multiple mutators, the write barrier is implemented using an atomic swap operation on the backing memory as to outrule a race condition among possibly concurrent read-write sequences on the same storage location. Although only required for the heap scan, we chose to activate the write barrier even prior to the initiation of the local root scans. This choice relaxes the phase transition from root to heap scan and avoids its system-wide synchronization through an atomic rendezvous.

3.5 Weak Reference Support

In contrast to their regular *strong* counterparts, weak references do not keep their referents alive. An object may become eligible for garbage collection in spite of the existence of paths via weak references to it. If the collector decides to discard the referent of a weak reference, the reference must be deprived of its capability to retrieve the referent, which is usually achieved by clearing it to `null`. Reference queues may establish an additional notification scheme allowing cleared references to be enqueued for processing by an application thread.

The Java 2 Standard Edition knows of several different strengths of weak references, which gives rise to the class hierarchy shown in Fig. 5. While the CLDC 1.1 only requires the `WeakReference`, our architecture additonally supports their soft variant and reference queuing.

Weak references are implemented as proxy objects containing a special reference member initialized from a strong reference provided to the constructor. The garbage collector must be enabled to recognize these special references in order to treat them appropriately in the heap scan and to clear them when it is about to collect their referents.

Instead of providing the GC with information about the hierarchy of the reference classes, we designated two bits of the reference word to tag the supported weak reference types as shown in Fig. 4. While these bits are generally cleared, they are set by the constructors of reference objects. This approach even enables a more flexible use of weak references outside the hierarchy of the reference classes.

During the heap scan, weak references are *not* followed. The treatment of soft references is decided once at the beginning of a GC cycle according to the current memory utilization so that they are followed as long as free storage is available. The objects containing unfollowed references are, however, enlisted during the scan. After its completion, this list of proxies is scanned for references to unreached referents. Any one found will be cleared, and the effected proxy will be communicated to the mutators for its possible enqueuing into a reference queue. For this purpose, the SHAP runtime system forks a designated service thread that maintains the communication with the memory manager via its Wishbone connection.

Special care is to be taken upon the retrieval of a strong reference from a weak proxy. Assume that an object only remains reachable through a weak reference. As long as the garbage collector does not discover this condition, the reference is not cleared but totally valid. The invocation of the `get()` method on the `WeakReference` proxy object will return with a normal strong reference to the referent effectively resurrecting the object from the brink of death. A race condition may, however, arise when such a resurrection interferes with an ongoing heap scan. It must be ensured that the reference to be returned by `get()` is either entered into the mark table prior to the completion of the heap scan or invalidated by returning `null`. Not knowing whether an ongoing heap scan will render the referent strongly-reachable or not, a consistent resurrection with a mark table entry is attempted first. Only if the scan has finished in the meanwhile, the actually determined reachability is evaluated. If necessary, `null` will be returned in conformance to the inevitable clearing of the original reference field.

4 Evaluation in SHAP

The described design was implemented for SHAP and integrated into the runtime system through adapted microcode implementations accessing the port to the

Table 1. Device Utilization of Reference Platform

Cores	**FFs**	**LUTs**	**BRAMs**
1	3206	7813	10
2	4465	11396	17
3	5720	14963	24
n	$\approx 1257n + 1950$	$\approx 3575n + 4240$	$7n + 3$

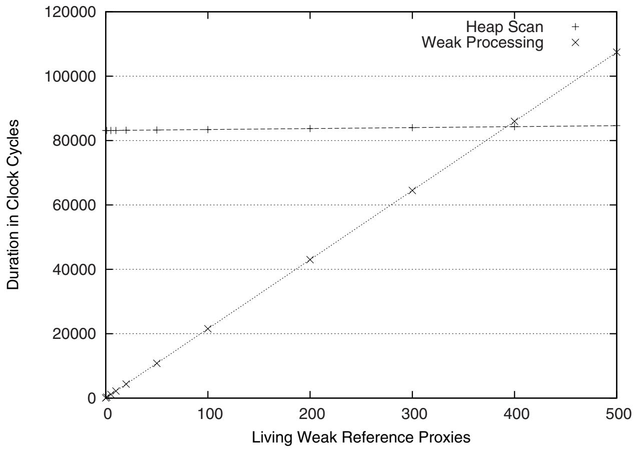


Fig. 6. Overhead of Weak Reference Processing

memory access manager as well as through regular Wishbone I/O for less time-critical operations. The latter also provides fundamental statistical data that we used for this evaluation. The runtime library was extended to enable application access to the new features and to provide implementations for the reference proxy and the reference queue classes. It was also turned into the first client of the weak reference support by the implementation of resource pools, which, for instance, enable the proper interning of strings.

The reference design was implemented on a Xilinx Spartan-3 XC3S1000, which is capable of holding up to three SHAP cores next to the memory management unit. The utilization of this reference platform is summarized in Tab. 1. As indicated, the demand on active resources grows linearly with the number of integrated SHAP cores. While most of the basic core-independent flip-flops and LUTs may be attributed to the central memory management, global Wishbone-attached IO accounts for about a quarter of it. The three Block RAMs outside the cores are used as MCU storage also containing the GC program and the global mark table.

The processing of living weak reference proxies constitutes a processing overhead as they are enlisted during the heap scan to be revisited thereafter in order to verify that their referents have been reached or to clear and enqueue them. This overhead grows linearly with the number of living proxy objects. This is

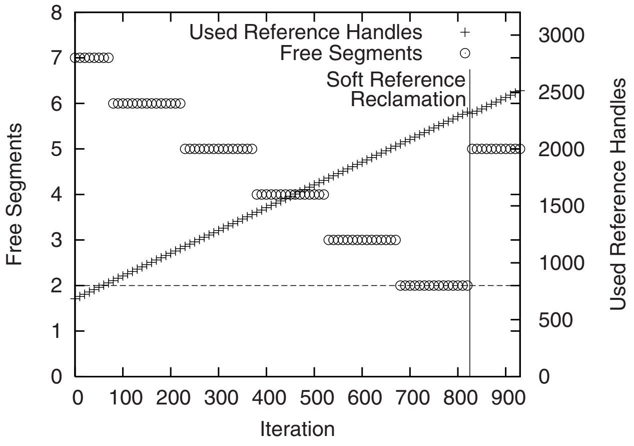


Fig. 7. Soft References and Growing Memory Utilization

shown in Fig. 6 for several scans of a heap with an identical object population, which merely differs in the number of weak reference proxies pointing to living objects. While the heap scan merely slows down marginally, the overhead becomes clearly visible in the succeeding weak reference processing. The lower impact on the heap scan is due to its hardware-assisted implementation inside the memory access management.

The distinguished treatment of soft references according to the current memory utilization is illustrated in Fig. 7. Initially, a set of few but large objects is allocated and made solely soft-reachable. Then, small objects are created continuously and kept reachable so that the memory fills up. While the large objects are initially kept alive, the available empty segments will eventually fall short of the threshold of two segments so that the soft references will be treated like weak ones by the GC. Consequently, the large objects are collected and the memory utilization relaxes.

5 Conclusions

This paper has demonstrated that the integration of advanced GC features is feasible even for small embedded bytecode processors. The presented solution makes thorough use of hardware acceleration wherever applicable while employing a main C-programmed software control for easy maintenance. The presented solution implements a concurrent garbage collector with the support of multiple mutators. Even going beyond the requirements of the CLDC 1.1, it includes the support for soft references and reference enqueuing. The practical implementation of the garbage collector was proven in the SHAP bytecode processor.

References

1. Zabel, M., Preußer, T.B., Reichel, P., Spallek, R.G.: Secure, real-time and multithreaded general-purpose embedded Java microarchitecture. In: 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007), pp. 59–62. IEEE, Los Alamitos (2007)
2. Holloway, J., Steele Jr., G.L., Sussman, G.J., Bell, A.: The SCHEME-79 chip. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Lab. (1980)
3. Moon, D.A.: Garbage collection in a large LISP system. In: LFP 1984: 1984 ACM Symposium on LISP and functional programming, pp. 235–246. ACM, New York (1984)
4. Nilsen, K.D., Schmidt, W.J.: Hardware support for garbage collection of linked objects and arrays in real-time. In: ECOOP/OOPSLA 1990 Workshop on Garbage Collection (1990)
5. Nilsen, K.D., Schmidt, W.J.: Cost-effective object space management for hardware-assisted real-time garbage collection. ACM Lett. Program. Lang. Syst. 1(4), 338–354 (1992)
6. Meyer, M.: A true hardware read barrier. In: ISMM 2006: 5th International Symposium on Memory Management, pp. 3–16. ACM, New York (2006)
7. Srisa-an, W., Dan Lo, C.-T., Chang, J.-e.M.: Active memory processor: A hardware garbage collector for real-time Java embedded devices. IEEE Transactions on Mobile Computing 2(2), 89–101 (2003)
8. Pfeffer, M., Ungerer, T., Fuhrmann, S., Kreuzinger, J., Brinkschulte, U.: Real-time garbage collection for a multithreaded Java microcontroller. Real-Time Syst. 26(1), 89–106 (2004)
9. Uhrig, S., Wiese, J.: Jamuth: an IP processor core for embedded Java real-time systems. In: Bollella, G. (ed.) JTRES 2007. ACM International Conference Proceeding Series, pp. 230–237. ACM, New York (2007)
10. Meyer, M.: A novel processor architecture with exact tag-free pointers. IEEE Micro 24(3), 46–55 (2004)
11. Meyer, M.: An on-chip garbage collection coprocessor for embedded real-time systems. In: RTCSA 2005: 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Washington, DC, USA, pp. 517–524. IEEE Computer Society, Los Alamitos (2005)
12. Gruian, F., Salcic, Z.A.: Designing a concurrent hardware garbage collector for small embedded systems. In: Asia-Pacific Computer Systems Architecture Conference, pp. 281–294 (2005)
13. Schoeberl, M.: JOP: A Java optimized processor. In: Meersman, R., Tari, Z. (eds.) OTM-WS 2003. LNCS, vol. 2889, pp. 346–359. Springer, Heidelberg (2003)
14. Reichel, P.: Entwurf und Implementierung verschiedener Garbage-Collector-Strategien für die Java-Plattform SHAP. Großer beleg, Technische Universität Dresden (2007)
15. Anton, A.: OpenFIRE (2007), `http://www.opencores.org/project,openfire2`
16. Harboe, Ø.: ZPU - the worlds smallest 32 bit CPU with GCC toolchain (2008), `http://www.opencores.org/project,zpu`
17. Henry, G., Baker, J.: List processing in real time on a serial computer. Commun. ACM 21(4), 280–294 (1978)
18. Gagnon, E.M., Hendren, L.J.: SableVM: A research framework for the efficient execution of Java bytecode. In: Java Virtual Machine Research and Technology Symposium, April 2001, pp. 27–40 (2001)

A Hierarchical Distributed Control for Power and Performances Optimization of Embedded Systems

Patrick Bellasi[1], William Fornaciari[1], and David Siorpaes[2]

[1] Politecnico di Milano, P.zza Leonardo da Vinci, 32. 20133 - Milano, Italy
bellasi@elet.polimi.it, fornacia@elet.polimi.it
[2] STMicroelectronics, Via C. Olivetti, 2. 20041 - Agrate Brianza, Italy
david.siorpaes@st.com

Abstract. Power and resource management are key goals for the success of modern battery-supplied multimedia devices. This kind of devices are usually based on SoCs with a wide range of subsystems, that compete in the usage of shared resources, and offer several power saving capabilities, but need an adequate software support to exploit such capabilities.

In this paper we present *Constrained Power Management* (CPM), a cross-layer formal model and framework for power and resource management, targeted to MPSoC-based devices. CPM allows coordination and communication, among applications and device drivers, to reduce energy consumption without compromising QoS. A dynamic and multi-objective optimization strategy is supported, which has been designed to have a negligible overhead on the development process and at run-time.

1 Introduction

New generation of mobile devices are usually based on platforms using a Multi-Processor System-on-Chip (MPSoC) which is composed of many subsystems and surrounded by a broad set of peripherals. Each subsystem is typically characterized by several working modes (WMs), with corresponding different levels of Quality-of-Service (QoS) provided and power consumptions profiles. Modern hardware technologies increased the capability to reduce both dynamic and static power consumptions [1]. However, several mechanisms to save power at circuit level require an adequate software support to be effectively exploited. Indeed, to properly satisfy applications' QoS demands it is required to track system resources availability and usage which directly impact on energy consumptions. Moreover the need for support of heterogeneous usage scenarios makes the management of resources and power saving a challenging design goal.

This work introduce the general idea behind the definition of a system-wide power and performances optimization strategy, to be implemented at OS level. This solution has been designed to be sufficiently portable among different platforms, without compromising too much its accuracy and efficiency, and easily adapt to all possible different device usage scenarios. We prove that a *hierarchical* *distributed control* is particularly suited to meet the goals of both adaptability and portability, without unduly compromising the effectiveness of control and its efficiency. A formal model to tackle the performances vs. power consumption trade-off is sketched. Finally, “Constrained Power Manager” (CPM), a Linux kernel framework implementing this model, is evaluated on a real-world multimedia mobile platform.

The rest of the paper is organized as follow. In the next section we briefly present an overview of the previous works. In Sec. 3 the proposed distributed control model is presented while in Sec. 4 some experimental results are reported. Conclusions and research directions are drawn in Section 5.

2 Related Works

Techniques to reduce power consumption in computing systems range from physical layers design up to higher software abstraction levels [2,3]. Considering a classification based on the abstraction levels, the main techniques proposed in literature fall into five categories: pure hardware, pure OS, cooperative OS, application level and cross-layer adaption approaches.

The ’pure hardware’ approaches mainly address the processor DVFS. These techniques are based on specific hardware support that measure the current CPU load and configure its frequency according to the inferred system utilization. Examples are the Transmeta’s *LongRun* technology, the Intel’s *Wireless Speedstep Power Manager* and the ARM’s *Intelligent Energy Manager* (IEM). All these approaches are completely transparent from the user-space, cannot exploit knowledge on future workloads and disregard specific application needs.

The ’pure OS’ techniques basically try to improve the memoryless hardware approaches in two main directions: by exploiting OS scheduler knowledge (e.g., [4]) and by allowing software system designer to compare different optimization policies [5]. A number of other works has focused techniques for the power optimization of specific subsystems, e.g.,disks, network cards and displays.

In ’cooperative OS’ approaches, the OS tries to exploit some “application hints” to increase the level of knowledge about tasks’ requirements. At this level of abstraction that we find the first comprehensive approaches that try to optimize the system-wide configuration considering all of its components [6,7].

The ’application level’ approaches, rather than a partnership between the OS and the applications, try to exports the entire burden of PM to the user level, resembling the philosophy of the Exokernels. Application adaption techniques trade power consumption with quality or data fidelity (e.g., [8]). Others techniques propose complete optimization frameworks, such as the Odyssey OS [9] or the Chameleon’s application-level PM architecture [10].

The development of holistic approaches, that aggregate data from multiple layers, is nowadays a popular research topic. Indeed, a number of approaches based on ’cross-layer adaptations’ have already been proposed. Unfortunately, available solutions are frequently designed only for the energy optimization of real-time multimedia tasks with fixed periods and deadlines [11], require application modifications to feed some meta-informations to the optimization layer

[12] or are based on complex models to be build off-line and thus not easily portable across different platforms [13]. In this class we can find also some Linux kernel framework, but they resulted to be too much simple (QoSPM) or with scalability limits (DPM [14]).

3 Constrained Power Management

The design of a cost-effective solution, for system-wide power and performances optimization, requires to tackle the problem at all the abstraction levels, considering both low-level architectural details and high-level application requirements.

The proposed approach supports the identification of an optimal trade-off between expected performances and reduce power consumptions, especially focusing on mobile multimedia embedded systems. Our technique is based on the concept of *Constrained Power Management (CPM)* presented in this section.

3.1 Hierarchical Distributed Control

The CPM is an optimization technique based on a hierarchical distributed control model. An overall view of the proposed solution is depicted in Fig. 1. To overcame the complexity of centralized approaches, our solution splits the system-wide control problem into two different sub-problems: low-level devices local controls and an higher-level distributed agreement control. Drivers run a device specific fine-tune policy, based on the system requirements and working conditions, which exploits the detailed knowledge on the specific device capabilities. The global optimization policy instead is implemented at a higher abstraction level and exploits both low-level informations, related to resource availability and hardware capabilities, and high-level informations related to application's QoS requirements.

Among the high-level global optimization policy and the multiple low-level local optimization policies, we focused our attention on the upper layer. This choice has a double motivation, on one hand it is the more interesting part, the other many local optimization policies have already been investigated. The upper layer is the more interesting because being the more abstract layer, the designed solution will be completely platform independent and thus it can be directly implemented within a portable framework. Instead, lower layer's policies must be strictly related to the devices and thus they require a detailed analysis of each specific device class, which could itself require a complete study. Moreover, as described in the prior-art, many researches have already focused on the definition of local optimization policies for different classes of peripherals.

3.2 Cross-Layer Framework Design

From a design perspective CPM is a cross-layer technique which involves three different levels: a low-level abstraction layer, a middle-level modeling layer and finally an high-level optimization layer. The first two layers allow respectively to

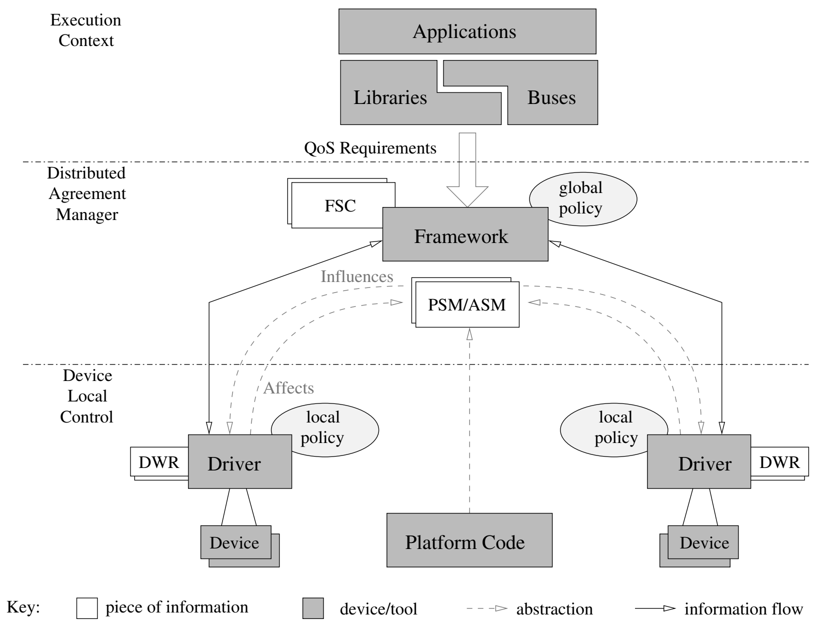


Fig. 1. A bird-eye-view of the proposed system-wide control architecture. The main system components are represented by platform code, drivers and the optimization framework code itself. These components either define or use some entities: System-wide Metrics (ASM/PSM), Device Working Modes (DWR) and Feasible System Configurations (FSC). Policies are both locally and globally defined and are implemented as modular components and can be changed at runtime.

abstract a real platform and to model that abstraction, thus getting a platform independent system description. The last layer use this description to build an efficient and portable optimization strategy. A representation of these layers, their components and the relationships among them is depicted in Fig. 2.

The Abstraction Layer provides a suitable abstraction for: resources, architecture and devices. System resources, regardless of their nature either hardware (e.g., frequencies) or not (e.g., latencies), are represented by the concept of *"System Wide Metrics" (SWM)*. The "metrics" term was chosen to remark their numerical interpretation within the model abstraction: SWMs represents the possible optimization objectives. Thus, the set of all available SWMs defines a *"System-Wide Configuration Space" (SWCS)* which can be explored to search the optimal configuration. The architecture abstraction, satisfying the 'fine-detail' requirement, allows to represent some platform specific details within the model. This is achieved by specializing the SWM into: *"Abstract System-Wide Metric" (ASM)* and *"Platform System-Wide Metric" (PSM)*. The former are abstract metrics, which can be exposed to user-space without compromising solution portability, while the latter are platform specific metrics, that allow

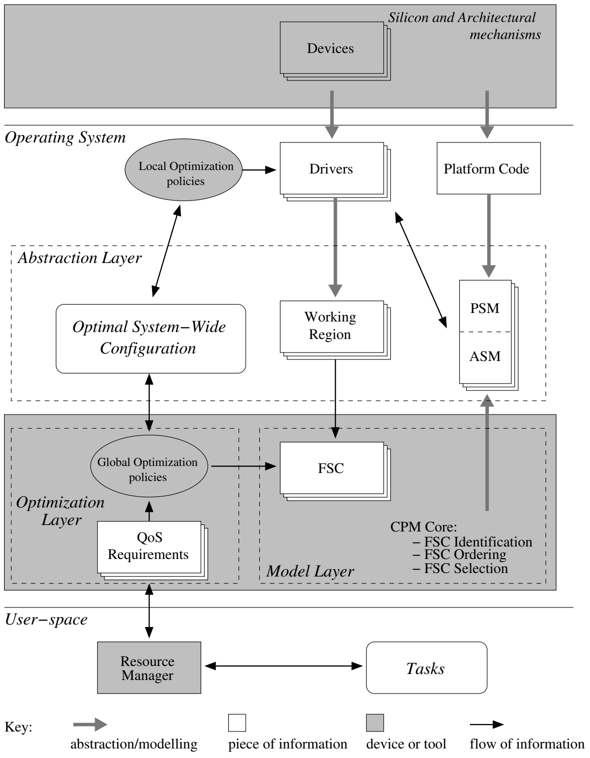


Fig. 2. A real computing system, with many different devices and applications, is so complex that it is not convenient to model all this complexity in order to solve the consumption and performances optimization problem. Therefore we perform an abstraction and modeling to properly support the optimization technique.

the framework to consider target system details (e.g., devices dependencies). Finally, each device in the system is described within the model by a set of *"Device Working Region" (DWR)*. A device generally can have different WMs, which correspond to different resources usage and supported QoS. WMs could be as simple as 'device on' and 'device off', or even more complex such as all the different operating frequencies of a CPU or the different connection protocols supported by a 3G modem. What exactly are the WMs of a device is defined by the corresponding driver. Thus, a DWR is an abstract representation of a device WM and it is defined by a set of ranges on each SWM which is sensible for the considered device. A device is "sensible" to a SWM if any change of its value can imply a reconfiguration of the device and vice versa. In instance, a DVFS driver for the control of the processor clock frequency is sensible to a SWM like 'CPU Frequency'.

The Model Layer takes as input the information representing the system resources and capabilities, exploiting the abstraction defined in the lower layer, and generates as output an architecture independent representation of all the

"Feasible System-wide Configurations" (FSC) available for the target system, that we named: FSC model. A FSC is a region on the SWCS, thus is defined by a set of validity ranges for each SWM, where it is granted that each device could be configured to operate in a WM that does not have any conflict with any other device. These regions are particularly important because they grant that any inter-dependency among devices is safely solved. Though a number of interesting theoretical techniques can be defined to identify at run-time what is the optimal system configuration, according to both the available resources and the required performance, every outcome is useless if it cannot be actually applied to the real system because of implicit inter-dependencies or hardware constraints ignored by the optimization policy itself. Indeed an optimized configuration cannot be identified regardless of its feasibility. Thanks to their interesting property, the identification of all system's FSCs is especially important. Considering this, the optimization technique proposed is based on the 'a-priori identification' of all and only the system feasible configurations. Thus, any optimization policy that will be developed on top of this framework, it will be granted to operate on a set of real and valid configurations and consequently each result can be safely applied to real system.

The Optimization Layer exploits the system view offered by the underlying FSC model to support the global optimization policy of the proposed hierarchical distributed control. This is obtained by the definition of a strategy to assign a "weight" to each feasible configuration according to the running optimization policy. The weight associated to FSC is defined to be a sufficiently abstract metric which can be easily adapted for a generic multi-objective optimization. The run-time tracking of application requirements is another goal of this layer. The abstract system model, based on the concept of FSC and their representation in the SWCS, is properly exploited at this layer to translate application requirements on constraints for the research of the optimal configuration. Indeed, application requirements are translated on constraints for the optimization problem which could invalidates some FSCs. Thus, this layer provides support for both: *FSC pre-ordering*, according to the optimization objectives of the running policy, and *optimal FSC selection*, considering user-space requirements to filter out run-time invalidated FSC.

3.3 Formal Validation of the Optimization Policy

The hierarchical distributed control problem can be conveniently reformulated using an appropriate formal model. A transposition of this type has been done not only to provides a rigorous description of the problem and a formal proof of the solution quality, but also allows to identify more easily a possible alternative solution strategy by taking into account the particularities of the specific formulation. We have reformulated the problem using *Linear Programming*.

A simple representation of the LP formulation is depicted in Fig. 3. In this simple scenario we consider a system with three devices (d_1, d_2 and d_3) and two SWMs (p_1 and p_2). The available FSCs are only three, but at run-time it could happen that some of them (FSC_3 in the example) are invalidated by

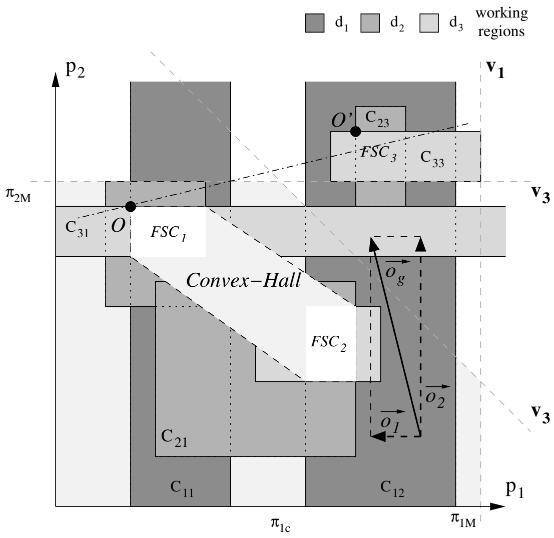


Fig. 3. Example of LP formulation for the global optimization problem

the constraints representing application requirements. The optimization policy is represented in the LP formulation by an objective function o_g which identifies the direction of a multi-objective optimization in the domain of the SWM. Indeed, the global optimization policy is defined by the composition of multiple objectives, each one corresponding to a different SWM metric with an associated optimization priority. It is possible to prove that the solution of the LP problem, O in the example, can always be mapped to one ore more FSC. These will represent the feasible configurations which are optimal w.r.t. the running policy.

3.4 Framework Implementation

The formal representation of the problem certifies the goodness of the configuration identified. Moreover, it also suggests an efficient implementation of the optimization algorithm which is composed by three steps:

1) *FSC Identification*: at boot time all the drivers register to CPM by exposing their DWRs using the abstraction layer interface. Thereafter, all the FSCs can be automatically identified by the model layer.

2) *FSC ordering*: a multi-objective optimization policy is settled by simply defining the optimization priority for each metric in the configuration space, i.e. by associating a "weight" to each SWM. Thus, starting from the set of all FSCs which are identified by the model layer, it is possible to pre-order the solutions of the optimization problem. In the previous example, according to the optimization policy represented by the vector o_g, the solutions ordered starting from the best one are: FSC_3, FSC_1 and finally FSC_2.

3) *FSC selection*: at run-time the requirements asserted by applications are aggregated and properly translated into constraints for the optimization

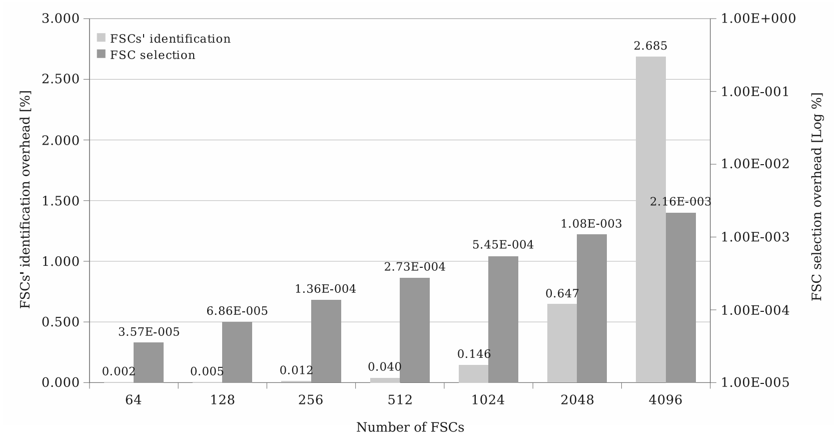


Fig. 4. CPM: overhead of FSC identification and selection

problem. These constraints are used to invalidate all FSCs that violate them. The optimal feasible solution is the first one valid in the list of the ordered FSCs. In the previous example, FSC_3 is invalidated by the assertion of the constraint v_3 and so the optimal solution is the next one valid: FSC_1.

4 Experimental Results

The proposed optimization model has been implemented in the "Constrained Power Manager" (CPM), a Linux framework based on kernel 2.6.32. This implementation has been used as a workbench for a worst-case complexity analysis of the developed optimization algorithms, using a demo-board with the Nomadik STn8815 MPSoC by STMicroelectronics. In this section we present: first the test results, and then a use-case to demonstrate the application of the framework in a real usage scenario targeting the power and performance optimization of a multimedia mobile device.

4.1 Overhead's Evaluation

We have measured the execution time of the algorithms for the identification and the selection of FSCs obtaining the results in Fig. 4. These overhead measurements refer to a 60s execution of the use-case and focus on the worst-case.

This measures prove the negligible impact of the framework with respect to a system not using it. Indeed, the identification algorithm shows a maximum of 2.5% overhead for a quite complex system with 4096 feasible configurations, which is much more than the 415 of the considered use-case. This means that over the 60s of use-case execution, around 1.5s are devoted to the framework execution. However it should be considered that this algorithm runs just one time at system boot and can be easily replaced by a look-up table. Indeed, especially in embedded systems, where the platform's configuration for a final product does not change, all the FSC can be pre-computed and then just loaded at boot time.

While the identification algorithm has a complexity which is exponential in the number of the FSCs, the selection algorithm not only has a better (linear) complexity but is also three orders of magnitude better in absolute values. This is also another important result, because the identification algorithm is the one executed more frequently, i.e. each time a new requirement is asserted by an application. The experimental setup considered one run every 10 seconds and the measurements show a really negligible overhead which is always less than 0.01%.

4.2 Use Case Definition

The presented use case shows the benefits of using CPM to manage resources such as the Internet connection bandwidth according to the actual applications demand. Moreover it shows how CPM can keep track of dependency between subsystems. The STn8815 SoC has an ARM host CPU and two accelerators for multimedia: an audio DSP (DSP_A) and a video DSP (DSP_V). The host CPU is clocked by a signal identified with CPU_CLK, while the DSPs are clocked with a signal labeled DSP_CLK. The SoC architecture is characterized by a strict dependency between CPU_CLK and DSP_CLK which constrains the operational frequency of the accelerators.

We considered the following ASMs in the use case:

– *connection bandwidth*: a resource on which applications compete. An additive function is used to aggregate the requirements, asserted by applications, and identify a constraint on the QoS level for this resource.
– *audio and video codecs*: an information that is related to multimedia content that the application has to play, thus it directly impacts on the operating mode of the DSPs.

The PSMs (DSP_CLK and CPU_CLK) are platform specific informations defined in the platform code to keep track of hardware inter-dependencies described so far. Finally, the driver of each involved device defines its own DWRs as represented in Tab. 1a, while the platform code define the architectural constraints represented in Tab. 1b.

The use case begins with the user selecting a video stream content to be played. As soon as the download of audio and video data starts, the player application collects informations about the connection bandwidth required to have good playback (264 Kbit/s)[1], the audio codec (mp3) and video codec (h.263) of the encoded content and set a QoS requirement on the corresponding ASMs. These requirements are expressed as a lower bound value on the ASM *bandwidth* and as a single value on the ASMs *audio codec* and *video codec.*

[1] i.e., no jitters and buffer underruns.

Table 1. Devices' operating modes and platform constraints. Each device defines its operating modes (a) by mapping them on SMWs' ranges: DSP_CLK (lower bound) and BANDWIDTH (upper bound). The platform code defines some constraints (b) to track dependencies between CPU_CLK and DSP_CLK.

(a) Operating Modes

SWM	**Audio DPS**			
	PCM	MP3	WMA	OGG
DSP_CLK	0	50	70	100

SWM	**Video DPS**			
	OFF	MPEG4	H.263	H.264
DSP_CLK	0	40	60	100

SWM	**3G Modem**				
	GPRS	EDGE1	UMTS	EDGE2	HSDPA
BANDWIDTH	57.6	236.8	384	473.6	7200

(b) Constraints

Platform	
CPU_CLK	DSP_CLK
19.2	19.2
100.8	100.8
201.6	100.8
264.0	132.0
302.4	75.6
393.6	98.4
451.2	112.8

The assertion of these constraints invalidates the current FSCs. Thus, CPM coordinates the selection of a new candidate FSC and the corresponding DWRs are communicated to the modem and DSPs for a distributed agreement. The required codecs are bound to a specific DSP_CLK frequency: 50MHz for DSP_A and 60MHz for DSP_V. The platform constraints (Fig. 1b) allow to manage the dependency with the CPU_CLK which is setup to support the desired frequency. Therefore the CPU frequency optimization policy will be able to scale the processor frequency according to the imposed constraint (not less than 100.8MHz). After the agreement, the candidate FSC is activated and all involved subsystems will move to the new working mode, e.g., the modem switches from GPRS to EDGE1.

The use-case continues and during the playback, it starts an application that downloads data from the web, e.g., an email update application. This new download application asserts a requirement on the bandwidth for an amount of 200 kb/s. Since the ASM *bandwidth* is of type *additive*, the aggregation function takes into account all the previously asserted requirements and aggregates with a sum. Thus, 464 kbit/s becomes the new active constraint on the bandwidth and thus a new FSC is selected, which brings the modem device to move to the EDGE2 working mode in order to satisfy the requirements.

Finally, the video stream playback ends and the corresponding requirements on bandwidth and codecs are de-asserted. This leads to a new aggregation on the bandwidth, which results in a subtraction of the value on the ASM *bandwidth*. At this point only the download application is still active and a new FSC is selected and activated.

4.3 Discussion on Use Case Results

System resources management. The declaration and aggregation of requirements allows to keep a correct and precise view of used and still available system resources. This could be exploited to configure the hardware devices on the best *feasible* operating mode that supports the resources demand. A positive effect of this method is the energy saving that could be achieved by selecting, for each QoS demand, the optimal working mode not only with respect to a multi-objective performances optimization policy but also considering the system-wide power consumption, which can be associated to every FSC.

Dependency tracking. CPM allows to track hardware dependencies among different subsystems of a SoC that may prevent a correct operation of a system. Instead of patching each device driver to adapt to the platform, developers declare platform DWRs to solve dependencies issues. In that way code portability is improved.

Identification of Feasible System-wide Configurations. The automatic computation of the FSCs allows to identify all the feasible working points of an entire platform. This is done by exploiting the information defined, independently, in each device driver code. Other approaches to PM require to code all the working points by hand. Considering that in the presented use case the total number of FSCs was 415, we understand how interesting is the ability to automatically compute these point. Thus, this is a relevant result by itself. Moreover, it improves the portability of the solution across different platforms because allows to reuse drivers defined informations.

Additive aggregation. This is a new concept, introduced in CPM, to solve a limitation present in the implementation of QoSPM where even for resources that are intrinsically additive (e.g., bandwidth) the aggregation function is of type min/max: not allowing to keep a correct view of system resources and bringing to select devices' WMs that actually can't support the QoS level required, e.g., if two applications require 300 kb/s each, QoSPM aggregates with the max thus with a final value of 300 Kb/s, i.e. an incorrect view of the resource's requirement.

5 Conclusions

We have presented CPM, Linux kernel framework for system-wide power and resources' optimization. The proposed method efficiently implements a well-known formal technique to solve optimization problems. The cross-layer design of the framework allows to collect and aggregate QoS requirements from the application layer and to coordinate the reconfiguration of device drivers working mode. A system-wide optimization, of both perceived performances and energy consumption, is supported by the definition of a global dynamic and multi-objective policy.

As revealed both theoretically and experimentally, the CPM approach allows to capture energy savings while fulfilling QoS constraints, thanks to a system-wide cross-layer dynamic optimization. Work is in progress, within a FP7 EU-funded project, to extend the approach towards multi-core architectures.

References

1. Keating, M., Flynn, D., Aitken, R., Gibbons, A., Shi, K.: Low Power Methodology Manual: For System-on-Chip Design. Springer Publishing Company, Incorporated, Heidelberg (2007)
2. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Comput. Surv. 37(3), 195–237 (2005)
3. Pedram, M.: Power optimization and management in embedded systems. In: ASP-DAC 2001: Proceedings of the 2001 Asia and South Pacific Design Automation Conference, pp. 239–244. ACM, New York (2001)
4. Lorch, J.R., Smith, A.J.: Operating system modifications for task-based speed and voltage. In: MobiSys 2003: Proceedings of the 1st international conference on Mobile systems, applications and services, pp. 215–229. ACM, New York (2003)
5. Pettis, N., Lu, Y.H.: A homogeneous architecture for power policy integration in operating systems. IEEE Transactions on Computers 58(7), 945–955 (2009)
6. Zeng, H., Ellis, C.S., Lebeck, A.R., Vahdat, A.: Ecosystem: managing energy as a first class operating system resource. SIGPLAN Not. 37(10), 123–132 (2002)
7. Anand, M., Nightingale, E.B., Flinn, J.: Ghosts in the machine: interfaces for better power management. In: MobiSys 2004: Proceedings of the 2nd international conference on Mobile systems, applications, and services, pp. 23–35. ACM, New York (2004)
8. Tamai, M., Sun, T., Yasumoto, K., Shibata, N., Ito, M.: Energy-aware video streaming with qos control for portable computing devices. In: NOSSDAV 2004: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video, pp. 68–73. ACM, New York (2004)
9. Flinn, J., Satyanarayanan, M.: Energy-aware adaptation for mobile applications. In: SOSP 1999: Proceedings of the seventeenth ACM symposium on Operating systems principles, pp. 48–63. ACM, New York (1999)
10. Liu, X., Shenoy, P., Corner, M.D.: Chameleon: Application-level power management. IEEE Transactions on Mobile Computing 7(8), 995–1010 (2008)
11. Yuan, W., Nahrstedt, K.: Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 149–163. ACM, New York (2003)
12. Fei, Y., Zhong, L., Jha, N.K.: An energy-aware framework for dynamic software management in mobile computing systems. ACM Trans. Embed. Comput. Syst. 7(3), 1–31 (2008)
13. Snowdon, D.C., Sueur, E.L., Petters, S.M., Heiser, G.: Koala: a platform for os-level power management. In: EuroSys 2009: Proceedings of the 4th ACM European conference on Computer systems, pp. 289–302. ACM, New York (2009)
14. Brock, B., Rajamani, K.: Dynamic power management for embedded systems. In: Proceedings on IEEE International SoC Conference, September 2003, pp. 416–419 (2003)

Autonomic Workload Management for Multi-core Processor Systems

Johannes Zeppenfeld and Andreas Herkersdorf

Technische Universität München
{zeppenfe,herkersdorf}@tum.de

Abstract. This paper presents the use of decentralized self-organization concepts for the efficient dynamic parameterization of hardware components and the autonomic distribution of tasks in a symmetrical multi-core processor system. Using results obtained with an autonomic system on chip simulation model, we show that Learning Classifier Tables, a simplified XCS-based reinforcement learning technique optimized for a low-overhead hardware implementation and integration, achieves nearly optimal results for dynamic workload balancing during run time for a standard networking application at task level. Further investigations show the quantitative differences in optimization quality between scenarios when local and global system information is included in the classifier rules. Autonomic workload management or task repartitioning at run time relieves the software application developers from exploring this NP-hard problem during design time, and is able to react to dynamic changes in the MP-SoC operating environment.

1 Introduction

1.1 Multi-core Processing

Single-chip multi-processors became the mainstream architecture template for advanced microprocessor designs across a wide spectrum of application domains. Intel, the market leader for general purpose computing, abandoned its traditional strategy to primarily scale microprocessor performance through continued increase in core clock frequency and introduced parallel dual-, quad- and recently octal-core (Nehalem) Xeon processors [1] instead. In their research labs, Intel integrated the 80 core Tera-Scale processor [2] with a 2D network on chip and sophisticated 3D memory stack access. SUN Niagara, ARM MPCore, IBM Cell Broadband Engine, Nvidia GeForce, the CSX700 from ClearSpeed, and the TILE64 from Tilera are other examples of a non-exhaustive list of massively parallel multi-core architectures for use in mobile communications, graphics processing, gaming, industrial automation and high-performance scientific computing. While progress in deep sub-micron CMOS technology integration enabled the physical realization of this vast amount of nominal processing capacity on a single chip, the application programmer community – across all of the above mentioned systems – is minimally supported by tools and methods to efficiently exploit the available parallel resources [3]. We regard this circumstance a major challenge for a fast and efficient adoption of multi-core processors.

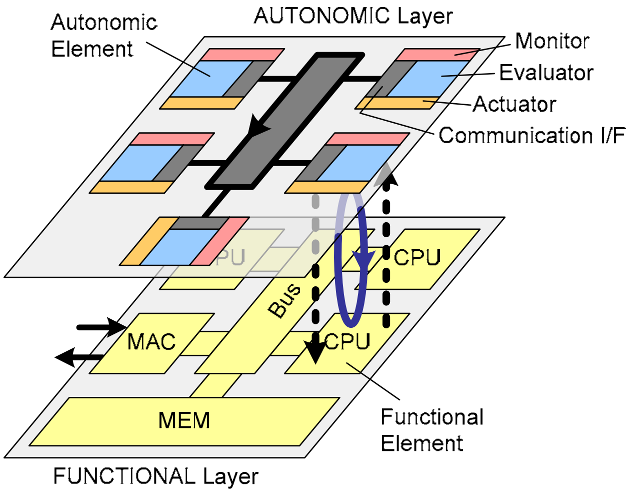


Fig. 1. Two-Layer ASoC Architecture

One of the key difficulties in efficiently executing an application composed of multiple, parallelizable tasks is to find an appropriate task distribution across available processing resources. Various approaches have been proposed to aid the designer in accomplishing this at design time, such as [4] which advocates the use of neural networks and support vector machines to map parallel programs onto multiple cores. Other approaches perform such a mapping at run time, but do not make use of learning techniques to improve mapping performance based on past experiences [5]. Combining the two, approaches such as [6] collect training data from function executions at run time, but perform any learning in additional design cycles offline. Numerous other publications propose similar solutions, however we are not aware of any existing approaches that make use of machine learning techniques to optimize task distribution in hardware at run time. In this paper we will therefore explore the applicability of the autonomic system on chip paradigm (ASoC, presented in the following section) to autonomously and dynamically partition individual tasks of a SW application among a set of homogenous processor cores at run time. Other optimization goals achievable using ASoC, such as reliability gain, power reduction or performance improvements, can then easily be included to fashion a complete reliable and efficient system on chip.

1.2 Autonomic Systems on Chip

The Autonomic System on Chip (ASoC) paradigm [7] proposes a two-layer architecture as shown in Figure 1. The functional layer corresponds to an ordinary SoC composed of various functional elements (FEs), such as CPU cores, bus interconnects, memory and I/O interfaces, etc. The autonomic layer contains a number of autonomic elements (AEs) used to optimize the corresponding FE's performance, power consumption and reliability. In order to achieve this, the AE monitors various aspects of the FE's operation (e.g. utilization). The resulting monitor signals are passed to an evaluator, which determines appropriate actions to be taken by an actuator (e.g. frequency adjustment). Finally, each AE contains a communication interface to share data with other AEs on the autonomic layer. This allows the AE evaluator to take into account global information when deciding on an action to perform, which results in better global optimization as demonstrated in section 3.

2 System Overview

The application chosen for this paper is Internet Protocol (IP) packet forwarding for a variety of packet header and payload processing scenarios under variable traffic workloads. In addition to being a common operation performed by modern network processors, packet forwarding lends itself well to the demonstration of autonomic enhancements, since it is easy to influence the system's workload by varying the incoming packet rate, type and size. In addition, since individual packets are relatively small and quick to process, a large number of packets can be processed in a relatively short amount of time. Not only does this reduce the required simulation time; it also allows for frequent changes in the system's functional behavior to demonstrate the adaptivity and learning capabilities of the autonomic layer.

From a more general perspective, IP packet forwarding is a representative example for a class of applications with the following properties: The application consists of a set of N tasks, T_1 to T_N, with every task having different processing requirements in terms of instructions per task (inst/T_i). Tasks can be traversed (partially) sequentially, conditionally or in iterative loops. The application is triggered by external events with varying parameter sets determining the system's overall workload. Mapping N tasks to a set of M homogeneous processing elements P_1 to P_M with the objective to achieve equal workload distribution corresponds to a function partitioning problem of exponential complexity, $O(M^N)$, which can't be solved with exact methods (e.g. ILP) for real-world problem dimensions. Instead of applying conventional partitioning heuristics (e.g. hierarchical clustering, Kernighan-Lin, simulated annealing or Tabu search) to explore limited partitioning alternatives during design-time, we propose adaptive run-time learning techniques to evaluate a much larger set of alternatives during run-time and to thus approach a globally optimal solution.

2.1 Functional Layer

The simulated hardware system used throughout the remainder of this paper is depicted in Figure 2. The yellow (unfilled) shapes make up the functional layer, while the blue (gray) rectangles represent the autonomic layer's evaluator. The evaluator consists of a Learning Classifier Table (LCT) [8] whose role is detailed in section 2.3.3 below. The underlying functional layer is a fairly simple, bus-based MP-SoC. Packets arrive and are transmitted over an Ethernet MAC (bottom center), are stored in memory (bottom right), and are processed by one or more of the CPUs (top). An interrupt controller (bottom left) is responsible for triggering processing tasks in the CPUs.

In simulation, the MAC actually functions as a packet generator, injecting different packet types and sizes into the system at varying rates. The memory is modeled

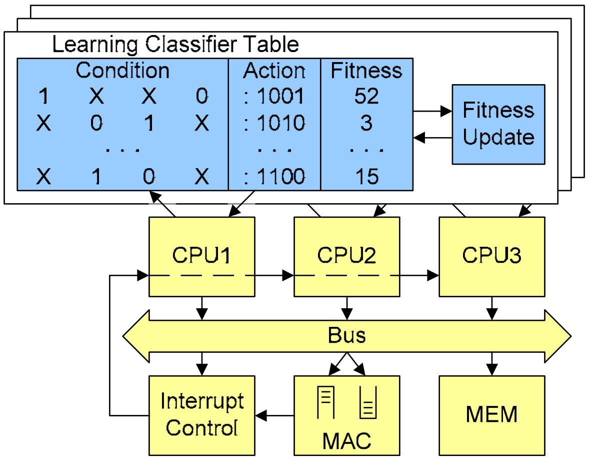


Fig. 2. Autonomic MP-SoC Network Processor Architecture

abstractly using fixed access delays to improve simulation performance, and could correspond either to an internal memory or to a memory controller connected to an off-chip RAM.

2.2 Application Software

The main focus of this paper is on workload management, which becomes more complex and "interesting" with an increasing number of processing tasks. In order to preserve maximum flexibility and the ability to run the application on a generic MP-SoC platform (shown in Figure 2), as many packet processing functions as possible are implemented as software tasks that can be moved freely among the system's CPUs. This also includes tasks such as transferring data between the MAC and memory, which alternatively could be accomplished easily – and perhaps more efficiently – by a dedicated hardware DMA controller.

In our application, every packet passes through five stages of execution:

Task 1: Transfer packet from MAC to memory
Task 2: Determine type of processing to be done on present packet
Task 3.1 through 3.N: Perform one of N packet header or payload processing tasks
Task 4: Reorder packets for in-order transmission
Task 5: Transfer packet from memory to MAC

While Tasks 1, 2, 4 and 5 are identical for every packet, Task 3 can be different for different packet types.

In principle, this task assembly allows for two orthogonal programming models on a generic hardware platform as introduced in Figure 2: Either balance incoming packets across all currently idle CPUs and execute all tasks for a packet on one and the same CPU (Run to Completion (RTC) model), or distribute the tasks among all CPUs and let each packet traverse the CPUs in a pipelined fashion (Pipelined model). Both models have unique characteristics, and both have advantages and disadvantages. RTC treats all cores as independent processing elements, thus eliminating the need for complex task partitioning among cores. RTC is easily scalable, but relies on no packet data sharing being required, and that the nature of the application leans towards event / packet balancing. Pipelining requires equal workload sharing of tasks among processing elements in order to achieve optimized throughput and efficiency (avoiding pipeline "bubbles"). On the positive side, the pipelining model is applicable to a larger class of parallel applications, achieves a smaller instruction footprint and supports local state and data caching.

2.3 Autonomic Layer

The autonomic layer used for this paper's simulations contains an autonomic element for each of the system's three CPUs. Autonomic elements for the bus, memory, MAC and interrupt controller are not included, as the CPUs are currently our primary target for optimization. However, AEs for the remaining functional elements are being considered for future development.

Along with an LCT evaluator, the CPU AEs contain several monitors and actuators to interact with the associated CPU FE. Each of these will be discussed in more detail in the following sections.

2.3.1 Monitors

Two local monitors keep track of the current CPU frequency and utilization, and are multiplied together to produce a third monitor value – the CPU's workload:

$$\mathrm{Load}_{\mathrm{CPU}} = \mathrm{Util}_{\mathrm{CPU}} \cdot \mathrm{Freq}_{\mathrm{CPU}} \tag{1}$$

Whereas the utilization indicates the percentage of cycles that the CPU is busy (i.e. is processing a packet rather than waiting in an idle loop), the workload indicates the actual amount of work that the CPU is performing per unit of time (useful cycles per second). Since the resulting value is helpful in comparing the amount of processing power being contributed by each CPU, it is shared with the other AEs over the AE interconnect. The workload information can then be averaged over all three CPUs, and by comparison with its own workload allows each CPU to determine whether it is performing more or less than its "fair share" of work. The difference between the CPU's and the average workload therefore provides another useful monitor value.

2.3.2 Actuators

In order to allow the AE to effect changes in CPU operation, two actuators are provided. The first of these simply scales the frequency by a certain value. Note that this is a relative change in frequency, i.e. the new frequency value depends on the old, which makes it easier to provide classifier rules that cover a larger range of monitor inputs. For example, with relative rules it is possible to express the statement "when utilization is high increase the frequency" using a single classifier rule, which would not be possible with an absolute actuator that sets the frequency to a fixed value.

The second actuator triggers a task migration from the AE's CPU to one of the other CPUs. After migration of a task, any further interrupts to start the execution of that task will be serviced by the target CPU instead. If a task migration is triggered while the task is already running, execution of the task is completed first (non-preemptive task migration). This minimizes the amount of state information that needs to be transferred from one core to another, reducing the performance impact of migrating tasks. In the current implementation, the task to be migrated is chosen randomly, as no monitoring is done to determine the workload of individual tasks, and therefore all tasks appear identical to the AE. The task is migrated to the CPU with the lowest workload. This targeting method requires no additional exchange of information, since workload information is already shared over the AE interconnect to allow calculation of the system's average workload. For simulations in which no global information is available, the target CPU is chosen randomly.

2.3.3 Evaluator

Choosing an appropriate action to be performed based on the incoming monitor signals is the responsibility of a learning classifier table (LCT) evaluator. LCT is a reinforcement based machine learning technique specifically designed for efficient integration in a HW system. LCT can be considered a "light-weight" XCS [9], the state of the art SW method for reinforcement learning classifier systems. LCT has previously been successfully applied for SoC robustness optimization under intermittent IP component failures [8]. In the following we will describe the rules and objective function used by the LCT for the network processor system presented in this paper.

Before a completed autonomic system can be deployed, an initial set of rules must be created and loaded into the system's LCTs. This initial rule set should cover the full input range of the utilization, frequency and relative workload monitors described above, with each rule proposing either a task migration or an increase or decrease in frequency. Overlapping rules are expressly permitted, allowing the LCT to choose which rule to apply. By keeping track of how well a certain rule performs, and by updating the fitness of that rule accordingly, the LCT learns to use the rule that has previously shown the best results in a certain situation.

Although the LCT can learn which rules are detrimental and can thereby avoid using them, it is preferable if the initial rule set has already undergone a filtering process to maximize the effectiveness of a newly deployed system. Figure 3 shows the general method that was used to generate the initial rule set for this paper. A non-critical sample system is initialized with random rules, and subjected to various representative traffic scenarios that might be expected in the field. When the system stabilizes, indicating that beneficial rules can be distinguished from counterproductive ones by means of their fitness, the current rule sets are extracted and stored. Thereafter, the system is loaded with new random rules and the process is repeated. If the system does not stabilize within a certain amount of time, the system is reset without storing the rule tables.

After a sufficient number of successful runs, the rules of all resulting rule sets are sorted by fitness, and a selection of the fittest rules is made for inclusion in the resulting rule set. This step is currently performed manually, but will eventually be automated with appropriate tools as part of our future work. The resulting rule set can then be used to replace the random initial rules and repeat the process. After one or more iterations of this process (just one was sufficient for generation of the initial rule set used by the simulations in section 3), the resulting rule set is ready for use as the initial rule set of the deployed system.

Determining the fitness of a rule, both during the initialization phase described above and during regular MP-SoC operation, is accomplished through the use of an

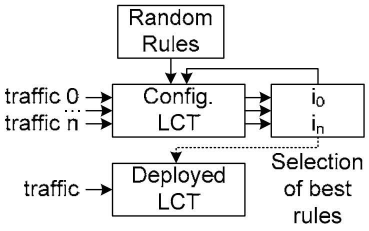


Fig. 3. Generation of the initial rule set

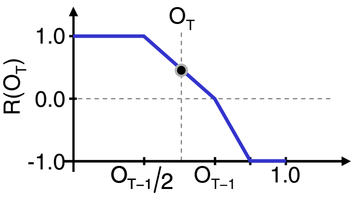


Fig. 4. Reward function

objective function, which evaluates all available monitor signals to determine how well the system is currently functioning. First, a delta value is calculated for each monitor signal to indicate how close that signal is to its optimal value (low values are desirable):

$$\delta_{frequency} \propto frequency \tag{2}$$

$$\delta_{utilization} \propto (100\% - utilization) \tag{3}$$

$$\delta_{workload} \propto \left| workload_{local} - workload_{average} \right| \tag{4}$$

The function used for each of these values is determined by the designer, and expresses the designer's optimization goal. For this paper, we would like each CPU to have as low a frequency as possible, which keeps the power consumption of the system low (voltage is automatically scaled in relation to the frequency). Likewise, we want the CPU utilization to be as high as possible, so that no processing cycles are wasted in an idle loop. Finally, the workload of all CPUs should be similar to avoid temperature hot spots and to ensure similar aging across all components. Depending on the available monitor signals and the designer's optimization goals, other delta functions may be chosen.

Combining the individual delta functions is accomplished by a simple weighting scheme, which yields the system's objective function:

$$f_{objective} = w_1 \cdot \delta_{frequency} + w_2 \cdot \delta_{utilization} + w_3 \cdot \delta_{workload} \tag{5}$$

The weights can be chosen according to which optimization goal is most important; for this paper each delta function is weighted equally.

Although the objective function provides a value indicating how well the system is performing at some instant in time, in order to determine the worth of a certain rule we need to calculate a reward R that compares the objective value sampled before and after the rule was applied. As shown in Figure 4, if the previous objective value O_{T-1} is larger than the new objective value O_T, indicating that the system has gotten closer to its optimal state, a positive reward is chosen based on the magnitude of the change. Otherwise, a negative reward is returned. This reward is then used to update the fitness of the applied rule, which allows the LCT to make use of past experience when determining which rule should be applied during similar situations in the future.

3 Simulation Results

Given the network processor MP-SoC presented above, this section presents simulation results that compare a classical system implementation with one enhanced by an autonomic layer. In order to show the adaptive nature of the autonomic system, various types of packet bursts, each consisting of 1000 packets, are sent in alternating fashion to the input MAC for processing. The packet inter-arrival rate is fixed at 4 μs, resulting in a burst duration of 4 ms independent of the packet processing duration. For each of the incoming burst scenarios, the system must either adapt to meet the processing requirements, or maintain a sufficient reserve of processing power to be able to cope with even the largest and most processing intensive packets.

All systems used below are based on a common base system configuration. The hardware components and their functionality, as well as the software tasks used for packet processing were described in sections 2.1 and 2.2, respectively. Unless stated otherwise, the initial task distribution is such that one CPU is responsible for ingress path processing (Tasks 1 and 2), one CPU is responsible for payload processing (Task 3), and the third CPU is responsible for egress path processing (Tasks 4 and 5). The initial CPU frequencies are chosen such that the system would be able to handle all burst scenarios without requiring parameter adaptation. Those systems containing autonomic enhancements use the monitors, actuators and LCT evaluator presented in section 2.3.

3.1 Comparison of Autonomic and Static Systems

The simulation results shown in Figure 5 compare the objective value and frequency adjustment of four different system configurations. The first two systems are static without any autonomic enhancements, where the first corresponds to the base system configuration described above. In the second static system, both the task distribution and CPU frequency were hand optimized with prior knowledge of the incoming packet traffic. The results of this optimized system demonstrate how an ideally parameterized static system compares to a system with autonomic enhancements.

The two autonomic systems are both based on the common system configuration, but are differentiated by the monitor signals available to them. While the second system uses the AE interconnect to share workload information globally among the CPU AEs, the first autonomic system must base its optimization decisions solely on local monitor information. Both systems remain capable of migrating tasks, however.

In comparison to a static system, the trend of the objective value clearly shows the benefits of an autonomic system regarding fulfillment of the objective function chosen in section 2.3.3. The autonomic system with global information is able to maintain system operation within approximately 10% of the optimum for all incoming traffic scenarios. Although the hand optimized static system is able to top this for the fifth and most processing intensive traffic scenario, it is not able to maintain such a high level of optimization across all bursts. This is a direct consequence of the fact that a system optimized for one scenario is not necessarily optimized for other scenarios. Whereas the designer must choose a certain scenario for which the static system

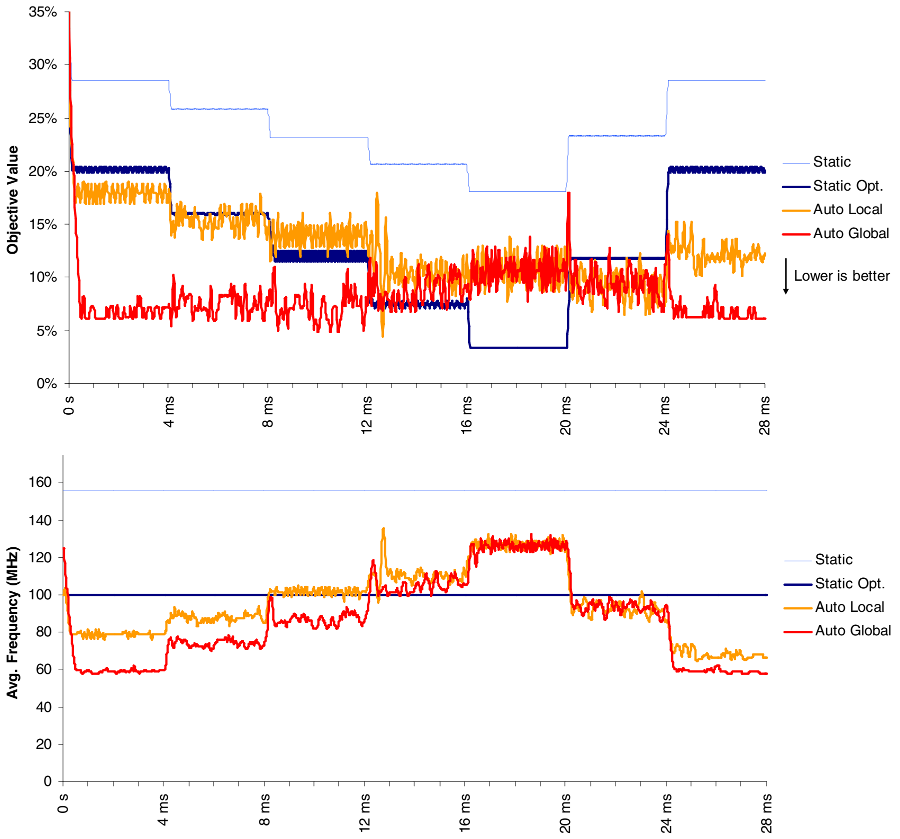


Fig. 5. Objective value and average CPU frequency of static and autonomic systems across various traffic scenarios. Lower values are better.

is optimized during design time, the autonomic system can optimize itself during run time to whatever scenario it is confronted with. Not only does this provide decent optimization for a much larger set of foreseeable and unforeseeable scenarios, it also relieves the designer of having to optimize all aspects of the system at design time.

Examining the trend of the average CPU frequency confirms the problem that a static system must meet the worst case processing requirements in order to remain functional over all workload scenarios, thereby wasting processing power during periods of lower activity. Except for processing of the most workload intensive fifth burst, the autonomic system is able to achieve an equal or lower average CPU frequency than the hand optimized system. The benefits of global information also become visible here, as the locally optimized system is not able to share the workload across the CPUs as effectively, requiring an increased average CPU frequency to meet the processing requirements.

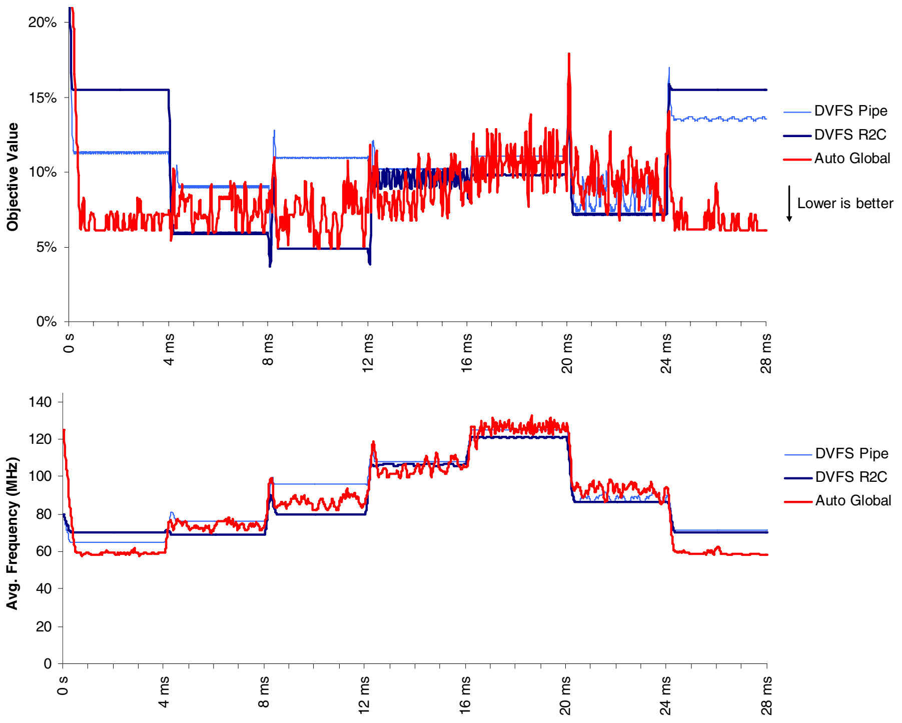


Fig. 6. Objective value and average CPU frequency of dynamic and autonomic systems across various traffic scenarios. Lower values are better.

3.2 Comparison of Autonomic and DVFS Systems

Figure 6 compares the globally optimized autonomic system presented in the previous section with two systems that employ dynamic voltage and frequency scaling (DVFS). DVFS allows the non-autonomic system to adjust its operating frequency in a fashion similar to the autonomic frequency actuator. The pipelined dynamic system corresponds to a DVFS-enhanced version of the hand-optimized static system from section 3.1. The dynamic run-to-completion system combines all tasks necessary for the processing of a packet onto a single CPU, allowing each of the system's three CPUs to completely process any packet type.

Looking at the objective function at the top of Figure 6, it can be seen that the autonomic system achieves results quite similar to those of the dynamic run-to-completion system. The difference in objective and frequency values for the first burst is due to the fact that only two CPUs are utilized by the run-to-completion system, since the processing time of the incoming packets is less than twice the interarrival rate. The first CPU has therefore completed processing before the third packet arrives, which means that at least one CPU is idle at any point in time. This results in two CPUs performing all the work, and keeps their workload high enough that the DVFS controller does not further reduce their frequency. Since the minimum frequency of a CPU in these simulations is 50 MHz, even for one that is idle, this results in a

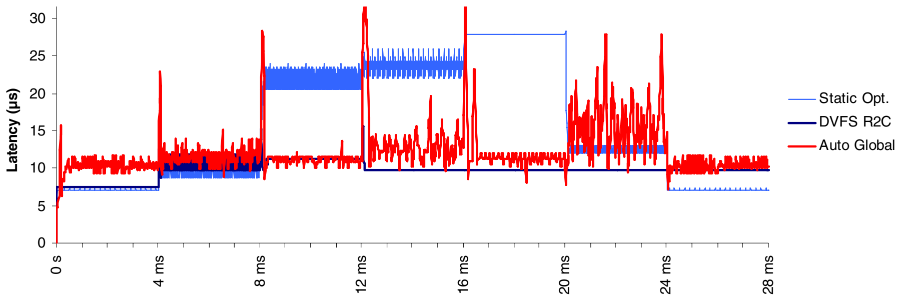


Fig. 7. Packet latency across various traffic scenarios

system-wide average frequency which is larger than that of the pipelined and autonomic systems, which split the load across all three CPUs. This also increases the delta value for workload (recall equation 4 from section 2.3.3) of the run-to-completion system, which further worsens the system's objective value.

Figure 7 shows the impact that the autonomic enhancements have on the packet latency, a global system behavior of which the autonomic evaluator is neither aware, nor has a direct influence over. At the beginning of each burst, the packet latency of the autonomic system shows a brief increase as the system adapts itself to the change in workload. Thereafter, the autonomic enhancements optimize the system such that the packet latency remains nearly constant at 10 µs across all traffic scenarios. During the sixth burst, where the packet latency rises slightly above this value, the autonomic system continues to search for a more preferable system configuration, but is interrupted prior to finding one by the following burst.

4 Conclusion and Future Work

In this paper, we have demonstrated the applicability of self-organization concepts and HW-based machine learning techniques for the run-time binding of SW tasks to homogeneous multi-core processors. SW application developers can follow established design flows for functional partitioning of applications into sub-functions (tasks) without being forced to consider the underlying parallel MP-SoC HW architecture. LCT-based hardware evaluators take care of balancing CPU workloads among available processing resources, without requiring a special programming language or fundamental OS modifications. It has been shown that the autonomic system is capable of parameterizing a pipelined architecture so that it performs similarly to a comparable, DVFS-enabled run to completion architecture. The resulting system combines the benefits of both architecture types, while delegating solution of the disadvantages, most notably the difficulty of efficiently distributing the workload of a pipelined system, to the autonomic layer.

Further work is planned for the completion of an FPGA hardware prototype to verify the presented simulation results in a functional MP-SoC, and to show that resource overheads are as small as preliminary synthesis results seem to indicate. We also plan to fully investigate the stability of the resulting system, although our simulations so far have shown that with a reasonable initial rule set, optimization generally occurs only for a short period following a change in workload, after which the system behavior enters a stable state. Finally, in addition to the autonomic elements connected to the CPU cores, further AEs are planned for the bus, memory and I/O modules. This will enable the parameterization of all system components, and provide additional global monitor signals to allow an even better optimization of the autonomic system.

Acknowledgements

We would like to thank our project partners Abdelmajid Bouajila and Walter Stechele from the Technische Universität München, Andreas Bernauer and Wolfgang Rosenstiel from the University of Tübingen, and Oliver Bringmann from the FZI in Karlsruhe. The ASoC project is sponsored by the DFG under SPP 1183 Organic Computing.

References

[1] Singhal, R.: Inside Intel Next Generation Nehalem Microarchitecture, `http://blogs.intel.com/idf/2008/08/sample_idf_sessions_inside_neh.php`
[2] Held, J., Bautista, J., Koehl, S.: From a Few Cores to Many: A Tera-scale Computing Research Overview. White Paper, `http://www.intel.com`
[3] Henkel, J.: Closing the SoC design gap. Computer 36(9), 119–121 (2003)
[4] Wang, Z.: Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 75–84 (2009)
[5] Streichert, T., Strengert, C., Haubelt, C., Teich, J.: Dynamic Task Binding for Hardware/Software Reconfigurable Networks. In: SBCCI 2006, pp. 38–43 (2006)
[6] Li, J., Ma, X., Singh, K., Schulz, M., de Supinski, B., McKee, S.: Machine Learning Based Online Performance Prediction for Runtime Parallelization and Task Scheduling. In: ISPASS, pp. 89–100 (2009)
[7] Bernauer, A., et al.: An Architecture for Runtime Evaluation of SoC Reliability. In: Informatik für Menschen. Lecture Notes in Informatics, vol. P-93, pp. 177–185 (2006)
[8] Zeppenfeld, J., Bouajila, A., Stechele, W., Herkersdorf, A.: Learning Classifier Tables for Autonomic Systems on Chip. Lecture Notes in Informatics, vol. 134, pp. 771–778. Springer, Gesellschaft für Informatik (2008)
[9] Wilson, S.: Classifier fitness based on accuracy. Evolutionary Computation 3, 149–175 (1995)

Firefly Flashing Synchronization as Inspiration for Self-synchronization of Walking Robot Gait Patterns Using a Decentralized Robot Control Architecture

Bojan Jakimovski, Benjamin Meyer, and Erik Maehle

Institute of Computer Engineering, University Lübeck, Germany
bojan@iti.uni-luebeck.de,
benjamin.meyer@informatik.uni-luebeck.de,
maehle@iti.uni-luebeck.de
www.iti.uni-luebeck.de

Abstract. In this paper we introduce and elaborate a biologically inspired methodology for robot walking gait pattern self-synchronization using ORCA (Organic Robot Control Architecture). The firefly based pulse coupled biological oscillator concept has been successfully applied for achieving self-organized synchronization of walking robot gait patterns by dynamically prolonging and shortening of robot's legs stance and swing phases. The results from the experiments done on our hexapod robot demonstrator show the practical usefulness of this biologically inspired approach for run-time self-synchronizing of walking robot gait pattern parameters.

Keywords: self-synchronization, organic robot control architecture, dynamically prolongation and shortening of robot's walking gait patterns, emergent robot gait synchronization, firefly synchronization, decentralized robot control architecture, robot gait pattern self-synchronization.

1 Introduction

The increasing complexity of technical and robotic systems and complicated robotic systems modeling [1] [2] has introduced a need for development and applying new concepts and methodologies towards creating self-capable, more robust and dependable systems. To achieve this, engineers have used different biologically and organically inspired approaches [3][4]. For example for the domain of joint leg walking robots some of the algorithms for motor control and walking gait pattern generations have been inspired and developed on observations seen within animals [5] and functioning of the neural circuitry [6]. In that context research has been done on Central Pattern Generators (CPG) - related to nerve cells in animal's spinal cord dedicated for generating periodic leg movement in animals and their application for robot control [7] [8] [9]. Additionally to this research various combinations have been also experimented on using artificial neural networks and genetic algorithms for robot gait pattern generation and gait control [10] [11] [12] [13]. However, just predefining some robot leg movements or generating gait patterns is sometimes not adequate for robust

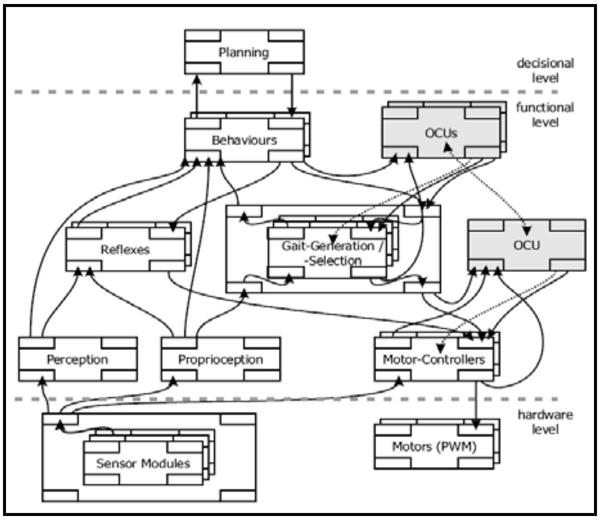


Fig. 1. ORCA - Organic Robotic Control Architecture

and fault-tolerant robot operation and it is usually an exhausting task on analyzing all the situations the robot may operate in. Self-organizing robotic systems on the other hand would overcome such problems by dynamically adapting to the situation without any complete pre-defining and modeling of the robotic system.

We have done research on combining our ORCA - Organic Robotic Control Architecture (Fig. 1) [14] with organically inspired approaches in order to achieve a self-organizing walking robotic system. ORCA architecture is modular architecture that consists of several OCU (Organic Control Unit) and BCU (Basic Control Unit) elements. The OCUs are related to monitoring tasks and observing the correct functioning of BCUs. BCUs are related to tasks such as: motor control, sensors, leg gait generation, etc.

The system is said to be self-organizing when the overall behavior of the system is a result of emergence, and not of a pre-ordained design [15]. The emergent property [16] [17] [18] of the systems arises as a result of simple local interactions of the components of the system and it cannot be anticipated even from complete information about the individual components constituting the system.

One type of emergence is synchrony, which is a collective organized behavior that occurs in populations of coupled oscillators [19]. Synchrony can be also observed in nature within fireflies and their flashing [20] [21].

In this paper we present the results of investigations done on applying biologically inspired synchronization for achieving self-synchronization for gait pattern parameters by joint-leg walking robots using the distributed and fault-tolerant robot control architecture - ORCA.

The rest of the paper is organized as follows: In the second chapter we give a short overview on firefly-based synchronization as biological inspiration for our concept.

In the third chapter we introduce our robot demonstrator on which we conduct the experiments and we present how the ORCA architecture is mapped to the morphology of the six-legged robot OSCAR in this self-synchronizing approach. In the fourth chapter we describe in detail our concept for self-synchronization of robot gait patterns. In the fifth chapter we present the results from real experiments done on our hexapod robot.

2 Firefly Flashing Synchronization

Firefly flashing is an example of synchronization seen in nature. Neuropsychological studies of the mechanism of flashing within fireflies [22] [23] has shown that rhythmic flashing of the male fireflies is controlled by a neural timing mechanism in the brain that gives a constant frequency of the flashing. Several studies [24] [25] have shown that external light, such as the light from other neighboring fireflies has an effect on firefly's flashing rhythm. In [26] [27] [28] results from experiments were presented that suggest that the external flash signal received from another firefly resets the flash-timing oscillator in the brain and therefore provides mechanism for synchronization of fireflies flashing.

In general, when one firefly sees the flashing of another neighboring firefly it shifts its rhythm of flashing in order to get in synchrony with another firefly's flashing. As a result a synchrony in firefly flashing takes place.

Such ideas of biologically inspired synchrony have been practically applied in engineering domains and mostly for achieving self-organized synchronization in networks [29] [30] [31]. For the domain of robotics, research has been conducted on synchronizing the behavior of robots in multi-robotic systems [32]. On the other hand, we are here interested on applying the firefly biologically inspired synchronization within one single robotic system – our hexapod walking robot.

The idea behind this is that we consider the robot's legs as individual units that can interact like fireflies and synchronize their gait patterns.

3 Robot Demonstrator - OSCAR (Organic Self Configuring and Adapting Robot)

We have conducted the experiments for firefly inspired gait pattern self-synchronisation on our 18 DOF, hexapod robot OSCAR (Organic Self Configuring and Adapting Robot) [33] [34] shown in Fig. 2 (a). The robot has six legs, with three servos per leg - named as alpha, beta and gamma (see Fig. 2), feet sensors, onboard control hardware and other sensors like ultrasonic, infrared, etc. Depending on their spatial location on the leg, the servos on the robot's legs are related to: protraction and retraction; elevation and depression; extension and flexion. In Fig. 2 (b), swing and stance phases of the robot's leg and their trajectories are presented. The swing and stance movements of the legs are essential for the robot movement. In the swing phase, the leg is moving from the posterior extreme position (PEP) to anterior extreme position (AEP) – shown in the same figure. In the stance phase, the leg is moving from AEP to PEP, which produces thrust that moves the robot over the ground.

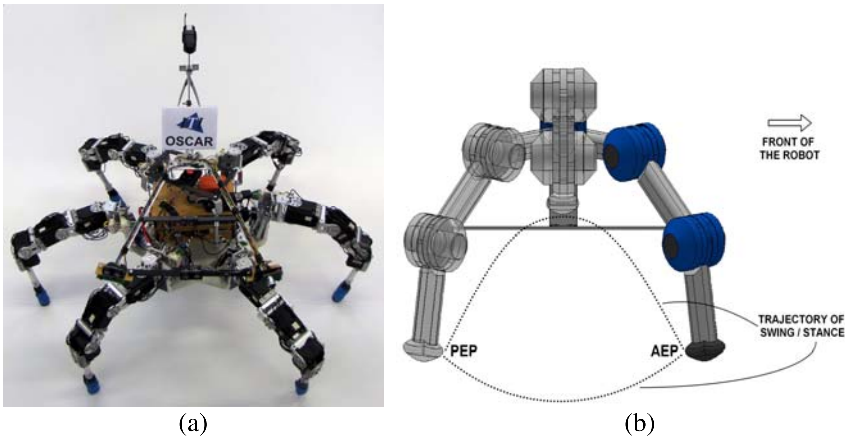


Fig. 2. (a) Hexapod robot OSCAR (Organic Self Configuring and Adapting Robot); (b) Robot's leg swing and stance phases and their trajectories

The swing and stance phases are characterized with the length of their respective trajectories. The lengths of swing and stance trajectories have direct influence on the speed with which the robot is moving over the terrain. The longer the trajectories of swing and stance, the bigger the distance travelled by the leg on the ground.

The tripod walking gait is the fastest and commonly observed gait by insect walking and lately transferred to robot walking, where in every moment of time three legs are in swing phase and the other three are in stance phase. This is shown in Fig. 3 (a).The legs of our hexapod robot and their numbering are shown on the robot model in Fig. 3 (b).

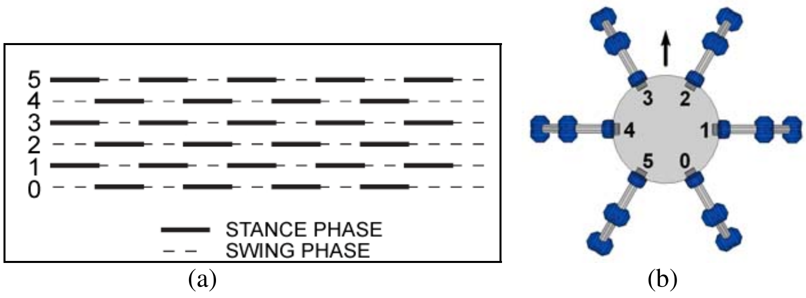


Fig. 3. (a) Tripod gait pattern; (b) Model of hexapod robot OSCAR with legs numeration. The arrow is showing the front direction of the robot.

Our firefly synchronization approach has also been designed to be used for robot walking using the ORCA architecture and emergent gait patterns (decentralized robot control). Each leg consists of an OCU (Organic Control Unit) which performs monitoring of the BCU (Basic Control Unit) actions. The BCUs are related to generating gait patterns, controlling of leg servo joint movements, etc. They send feedback signals to an OCU which monitors the 'health' status of the leg. The BCU related to leg gait generation sends synchronization signals to other BCUs of other neighbouring legs. This is represented in Fig. 4. The self-synchronization achieved by this is explained in the next section.

Therefore, besides the possibility for gait pattern generation in an emergent way [33], also the synchronization of walking gait patterns is achieved in a decentralized and self-organizing way. Such a decentralized leg control architecture is very important for developing fault-tolerant robotic systems.

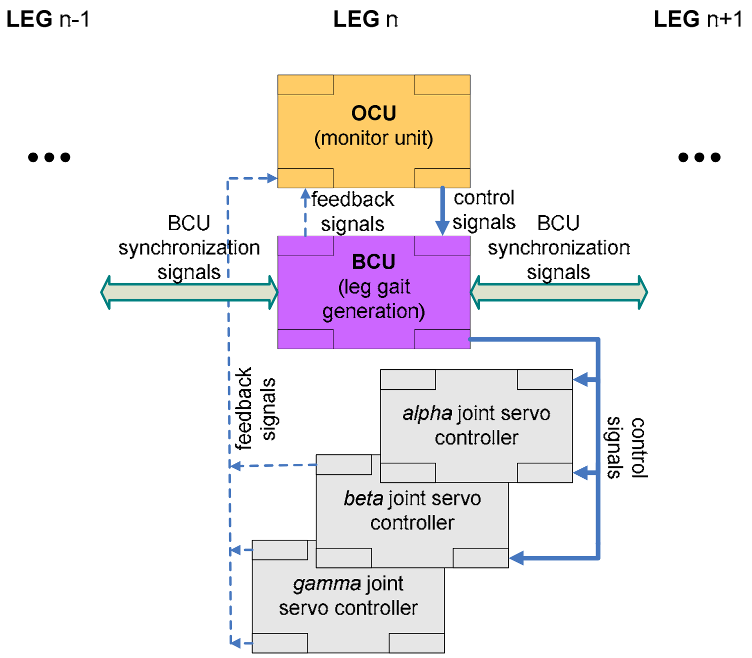

LEG n-1
LEG n
LEG n+1
OCU (monitor unit)
feedback signals
control signals
BCU synchronization signals
BCU (leg gait generation)
BCU synchronization signals
feedback signals
alpha joint servo controller
control signals
beta joint servo controller
gamma joint servo controller

Fig. 4. Decentralized robot leg control architecture for gait pattern self-synchronization

4 Walking Robot Gait Pattern Self-synchronization Inspired by Firefly Flashing Synchronization

The idea behind the synchronization of the gait patterns is that sometimes the gait pattern of the robot can change while the robot is walking. This may happen for example when the emergent walking pattern is used for the robot's walking and some situations appear like: the leg is approaching some rock and some sensor (ultrasonic, infrared, etc.) related to the leg's movement "informs" the leg that it should change the length of its stance/swing phase in order to overcome the obstacle. In that case the other legs have to synchronize their stance/swing phase as well. A similar situation may occur when perhaps the servo related for protraction of the leg gets somehow blocked and it can only move within a very limited range. In that case the other legs within the robot should also synchronize the length of their stance/swing phases.

The self-synchronization that we are introducing in this paper is inspired by firefly synchronization seen in nature, i.e. when one firefly sees the flashing of another neighbor firefly it shifts its rhythm of flashing in order to get in synchrony with another firefly's flashing. In our robot this would be translated to when one leg changes (by shortening or prolonging) its own gait, then the other neighboring legs adapt to this change by shortening or prolonging of their own gait as well. One more rule that we add to this is that the prolongation takes place only in their stance phase of the leg and the shortening of the gait takes place only in the swing phase of the leg.

As result, the walking gait of each of the robot's legs adapts to the change of walking gait of the neighboring legs, and by that the synchronization of walking gait patterns is achieved without using global coordination for the change of the gait pattern by each of the robot's legs.

The principle of functioning of this concept is shown in Fig. 5. There are two columns: "Synchro By Prolongation" and "Synchro By Shortening" describing the concept of self-synchronization when the length of the stance phase is prolonging or when the length of the swing phase is shortening. Each of the columns has a, b, c, d figure subparts, where it is shown how the synchronization of the gait patterns takes place by each of the legs. The lines at the legs show if the leg is in a stance phase – with "U" shaped line, or is in a swing phase - "∩" shape. The number of dots on the line represents the duration of the stance or swing phases. For example 3 dots will represent 3 length units of the swing and stance phase. The length units by the real robot movement can be translated into degrees of the movement of the leg. For example length of 3 units may represent 30 degrees of protraction leg movement, or similar.

For describing our concept we assume that all the legs at start have the same swing and stance parameters, i.e. 3 length units. In Fig. 5 in column "Synchro By Prolongation" - subfigure (a) the leg number 3 prolongs its stance phase from 3 length units to 4 length units. In subfigure (b) that follows in the same column, using the firefly inspired synchronization the neighbor legs numbered 2 and 4 in their stance phase synchronize their gait length from 3 length units to the length of the gait of leg 3 i.e. 4 length units. In subfigure (c) in the same column, the synchronization wave spreads further to the other neighbor legs numbered 1 and 5 which also synchronize in their stance phase their length from 3 length units to 4 length units. In subfigure (d) in the same column, the leg numbered 0 in its stance phase synchronizes its length parameter from 3 length units to 4 length units. With this step, the self-synchronization of the walking gait pattern is finished and the robot continues to walk further with all legs having gait 4 length units resulting in a greater speed of the robot over the ground. In Fig. 5 in column "Synchro By Shortening" a self-synchronization of the gait pattern is shown when the legs are shortening the length of the gait within their swing phases. In subfigure (a) in the same column, the leg number 2 decreases its gait length in the swing phase from 3 length units to 2 length units. In the subfigure (b) in the same column the next cycle is represented where the other neighbor legs numbered as 1 and 3 are decreasing their gait lengths in their swing phases from 3 length units to 2 length units. In subfigure (c) in the same column, the legs numbered 0 and 4 are the next ones that are shortening their gait length from 3 length units to 2 length units. At the end also the leg numbered 5 is getting in synchronization and shortens its gait length from 3 length units to 2 length units. This is represented in subsection (d) of the same column. With this ends the self-synchronization of the gait pattern by the robot's legs and the robot continues to walk with slower pace due to the decreased length of the swing and stance phases. The synchronization time depends on the final gait length units to which the legs synchronize their swing and stance phases. And the stability of the whole synchronization process depends on how often such synchronization gets started by some external influences in some rather short time domain.

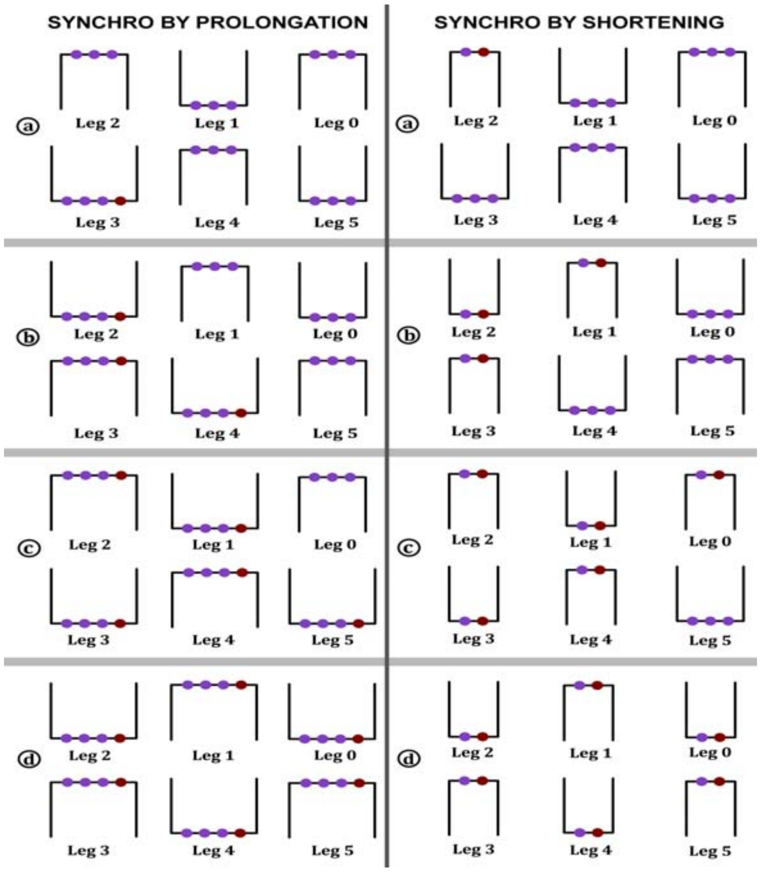


Fig. 5. Self-synchronization by shortening and prolongation of walking gait patterns

5 Results from the Experiments

In order to prove our biologically inspired concept for self-synchronization of robot's legs gait parameters we have conducted several experiments on our hexapod robot demonstrator OSCAR - Fig. 2 (a). The results from experiments are represented in Fig. 6 (a, b, c). In figures Fig. 6 (a), (b), (c), the x-axis represents the time in seconds and the y-axis represents the swing / stance phases by the robot's legs.)SW-swing phase; ST-stance phase.)

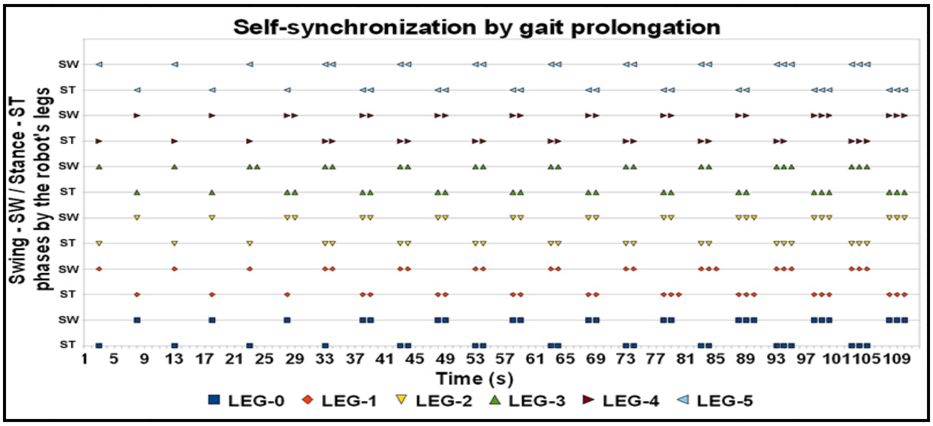
Normally we perform our experiments on our hexapod robot using emergent gait patterns generated in a distributed manner by each of the robot's legs mapped suitably into the ORCA architecture. However, in these experiments we have chosen to use the insect-inspired tripod gait walking pattern [35] [36] (which can be simulated with centralized robot control) *just to better represent the results of our experiments* done on self-synchronization. In this case swing/stance phases by robot's legs can be easily distinguished on the figure resulting from experiments, in contrast to emergent type of walking where the gait pattern often changes and therefore might obscure the idea of this research. Another factor that has been brought in the experiments is that the start of the synchronization is not done by some external influence, but this start has been pre-programmed so the experiments can be focused on principle of the synchronization process itself.

The experiments were defined as following:

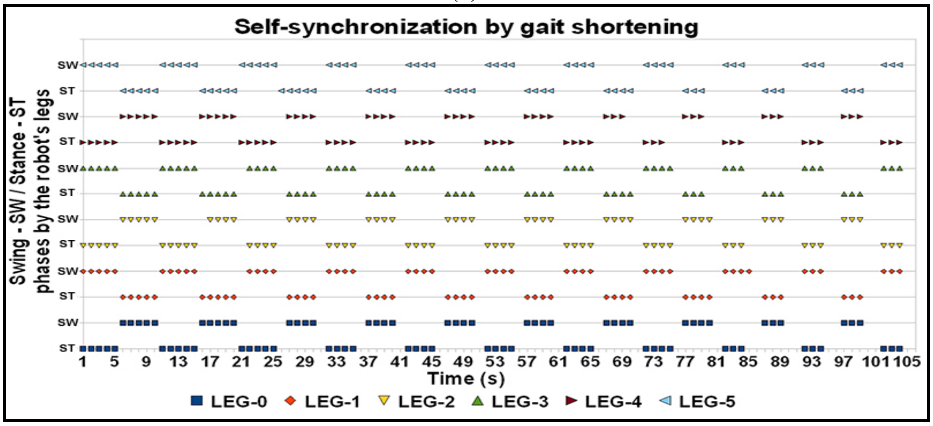
Fig.6 (a): The first experiment is about self-synchronization by prolongation of the robot's swing and stance phases. The robot starts to walk with a parameter length of 1 for the swing and stance phases. At first this parameter gets changed by the leg number 3 at time 23s during its stance phase. Then in the next cycle the neighboring legs 2 and 4 in the stance phase prolong their length parameters during their stance phases at time 28s. After this, at time 33s, the legs numbered 1 and 5 also synchronize their swing and stance length units to 2 in their stance phases. At time 38s also the leg numbered 0 synchronizes its length of swing and stance to 2 length units. In a similar fashion the self-synchronization and prolongation of stance and swing phases is further done for each of the legs prolonging from 2 length units to 3 length units. At the end a gait pattern gets self-synchronized with length units 3 for the swing and stance phases.

Fig.6 (b): In the second experiment a shortening of the gait pattern takes place from 5 length units to 3 length units over the course of the experiment. The robot starts to walk with length parameter of 5 for swing and stance phases. At first at a time 17s the leg number 2 decreases its swing phase the swing / stance parameter from 5 length units to 4 length units. After that, in the second cycle the legs numbered 1 and 3 at time 22s in their swing phases decrease their stance / swing parameters to length of 4 units. Then follows the synchronization by legs numbered 0 and 4 at time 27s in their swing phases and they synchronize their length parameters to length of 4 units. At time 32s also leg number 5 synchronizes its stance / swing length parameter to length of 4. In similar fashion the self-synchronization also continues for stance / swing lengths up to 87s of time when all the legs have the length of the stance / swing parameter of 3.

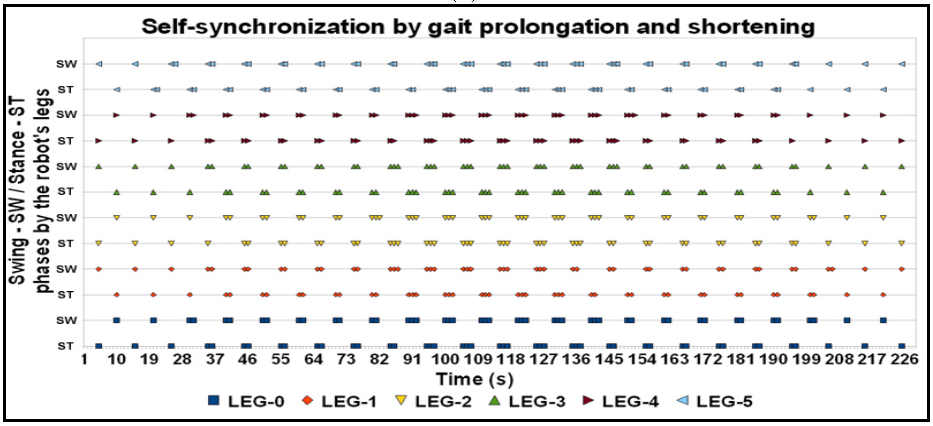
Fig.6 (c): In this experiment we have performed dynamic self-synchronization of the robot walking gait pattern starting from stance / swing parameter length of 1 then increasing to parameter length of 3 and then decreasing again to stance / swing parameter length of 1. The first change of the parameter occurs by the leg number 5



(a)



(b)



(c)

Fig. 6. (a) Self-synchronization by prolongation of the robot's swing and stance phases; (b) Self-synchronization by shortening of the robot's swing and stance phases; (c) Self-synchronization by prolongation and shortening of the robot's swing and stance phases within its stance phase at time 20s. After this cycle, the next parameter change occurs by neighboring legs 0 and 4 in their stance phases at about 30s increasing from length parameter of 1 to length parameter of 2. In the next cycle the legs numbered 1 and 3 in their stance phases get synchronized to parameter length of 2 at about 35s. The synchronization continues within the leg number 2 which adjusts its parameter from length 1 to length 2 at 40s. In similar fashion the self-synchronization continues in the same experiment also for prolongation of the stance / swing phases from length parameter 2 to length parameter 3 for each of the legs. At time 95s all the legs are in synchronization having the stance / swing length parameter of 3.

Similar to the prolongation, the shortening of stance / swing parameters in the same experiment takes further place and the length parameters by all the legs get shortened from length 3 to length 1 within their swing phases. With this last experiment we have demonstrated that also dynamic self-synchronization by increasing and decreasing of the gait parameters can be performed one after another during the walking of the robot.

6 Conclusion

In this paper we have introduced a biologically inspired approach for gait patterns self-synchronization by joint leg walking robots. Throughout the paper we have described the relation to the firefly synchronization and by schematic representation of the self-synchronization by prolongation and shortening we have explained the principles of this concept.

The results for experiments done on our hexapod robot OSCAR have proved that this approach can be practically applied for achieving a self-synchronization of the walking robot gait patterns using a decentralized robot control architecture. Such a concept can be also very useful for gait generation and speed adaptation for fault-tolerant walking machines where legs sometimes get malfunctioned during the robot's operation.

Further work will include conducting more real case experiments in order to test the adaptability and robustness of the proposed methodology. Additional investigations will be also done on improving the information flow in such a decentralized robot control architecture.

Acknowledgment. This work is partly supported by German Research Foundation - DFG (associated to SPP 1183, MA 1412/8-1).

References

1. Han, B., Luo, Q., Wang, Q., Zhao, X.: A Research on Hexapod Walking Bio-robot's Working Space and Flexibility. In: IEEE International Conference on Robotics and Biomimetics, pp. 813–817 (2006)
2. Cifuentes, N.J.R., Porras, J.H.G.: Modeling of legged robot based on Colombian insect observations. In: Electronics, Robotics and Automotive Mechanics Conference (CERMA), pp. 506–511 (2007)

3. German Science Foundation (DFG) Priority Program SPP 1183, Organic Computing (2004), http://www.organiccomputing.de/spp
4. Müller-Schloer, C.: Organic Computing – On the Feasibility of Controlled Emergence. In: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 2–5 (2004)
5. Suzuki, H., Nishi, H., Aburadani, A., Inoue, S.: Animal Gait Generation for Quadrupedal Robot. Second International Conference on Innovative Computing, Information and Control (ICICIC), pp.20 (2007)
6. Delcomyn, F.: Neural basis for rhythmic behaviour in animals. Science 210, 492–498 (1980)
7. Barnes, D.: Hexapodal robot locomotion over uneven terrain. In: Proceedings of the 1998 IEEE International Conference on Control Application, vol. 1, pp. 441–445 (1998)
8. He, J., Lu, C., Yin, S.: The Design of CPG Control Module of the Bionic Mechanical Crab. In: Proceedings of the 2006 IEEE International Conference on Robotics and Biomimetics, Kunming, China (2006)
9. Billard, A., Ijspeert, A. J.: Biologically inspired neural controllers for motor control in a quadruped robot. In: IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN 2000), vol. 6, pp. 6637(2000)
10. Heinen, M.R., Osório, F.S.: Applying Neural Networks to Control Gait of Simulated Robots. In: 10th Brazilian Symposium on Neural Networks, pp. 39–44 (2008)
11. Manoonpong, P., Wörgötter, F.: Neural Control and Learning for Versatile, Adaptive, Autonomous Behavior of Walking Machines. In: International Conference on Advanced Computer Theory and Engineering, pp. 24–28 (2008)
12. Valsalam, V.K., Miikkulainen, R.: Modular Neuroevolution for Multilegged Locomotion. In: GECCO 2008, pp. 265–272 (2008)
13. Heinen, M.R., Osório, F.S.: Morphology and Gait Control Evolution of Legged Robots. In: IEEE Latin American Robotic Symposium, pp. 111–116 (2008)
14. Brockmann, W., Maehle, E., Mösch, F.: Organic Fault-Tolerant Control Architecture for Robotic Applications. In: 4th IARP/IEEE-RAS/EURON Workshop on Dependable Robots in Human Environments, Nagoya University, Japan (2005)
15. Serra, R., Zanarini, G.: Complex Systems and Cognitive Processes. Springer, New York (1990)
16. As, G.C., Odell, J.: Agents and Emergence. ACM, Distributed Computing (1998)
17. Mainzer, K.: Self-Organization and Emergence in Complex Dynamical Systems, pp. 590–594. GI Jahrestagung (2004)
18. Randles, M., Lamb, D., Taleb-Bendiab, A.: Engineering Autonomic Systems Self-Organisation. In: Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems, pp. 107–118 (2008)
19. Zheng, Z., Hu, G., Hu, B.: Phase Slips and Phase Synchronization of Coupled Oscillators. Physical Review Letters 81 (1998)
20. Buck, J.: Synchronous rhythmic flashing of fireflies, II. Quart. Rev. Biol. 63, 265–289 (1988)
21. Bottani, S.: Pulse-Coupled Relaxation Oscillators: From Biological Synchronization to Self-Organized Criticality. Phy. Rev. Lett. 74(21)
22. Case, J.F., Strause, L.G.: Neurally controlled luminescent systems. In: Herring, P.J. (ed.) Bioluminiscence in Action, pp. 331–366 (1978)
23. Buonomicini, M., Magni, F.: Nervous control flashing in the firefly Luciola italica L. Archives italiennes de Biologie, 323–338 (1967)

24. Buck, J.: Studies on firefly: I. The effects of light and other agents on flashing in Photinus pyralis, with special reference and diurnal rhythm. Physiological Zoology, 45–58 (1937)
25. Magni, F.: Central and peripheral mechanisms in the modulation of flashing in the firefly Luciola italica L. Archives italiennes de Biologie, 339–360 (1967)
26. Hanson, F.E., Case, J.F., Buck, E., Buck, J.: Synchrony and flash entrainment in a New Guinea firefly. Science 174, 161–164 (1971)
27. Winfree, A.T.: Biological rhythms and the behavior of populations of coupled oscillators. Journal of Insect Physiology 15, 597–610 (1967)
28. Buck, J., Buck, E.: Synchronous fireflies. Scientific American 234, 74–85 (1976)
29. Babaoglu, O., Binci, T., Jelasity, M., Montresor, A.: Firefly-inspired Heartbeat Synchronization in Overlay Networks. In: First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, pp. 77–86 (2007)
30. Lessons in Implementing Bio-inspired Algorithms on Wireless Sensor Networks. In: NASA/ESA Conference on Adaptive Hardware and Systems, pp. 271–276 (2008)
31. Self-Organized Synchronization in Wireless Network. In: Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 329–338 (2008)
32. Wischmann, S., Hülse, M., Knabe, J., Pasemann, F.: Synchronization of internal neural rhythms in multi-robotic systems. Adaptive Behavior 14, 117–127 (2006)
33. El Sayed Auf, A., Mösch, F., Litza, M.: How the six-legged walking machine OSCAR handles leg amputations. In: Workshop on Bio-inspired Cooperative and Adaptive Behaviours in Robots, Rome, Italy (2006)
34. Jakimovski, B., Litza, M., Mösch, F., El, S.A.: Development of an organic computing architecture for robot control. In: Informatik 2006 Workshop on Organic Computing - Status and Outlook, Dresden (2006)
35. Ferrell, C.: A Comparison of Three Insect Inspired Locomotion Controllers. Robotics and Autonomous Systems (RAS) 16, 135–159 (1995)
36. Beer, R.D., Quinn, R.D., Chiel, H.J., Ritzmann, R.E.: Biologically inspired approaches to robotics what can we learn from insects. Communications of the ACM 40(3), 30–38 (1997)

The JoSchKa System: Organic Job Distribution in Heterogeneous and Unreliable Environments

Matthias Bonn[1] and Hartmut Schmeck[2]

[1] Steinbuch Center for Computing
[2] Institute AIFB
Karlsruhe Institute of Technology – KIT –, 76128 Karlsruhe, Germany
{Matthias.Bonn,Hartmut.Schmeck}@kit.edu

Abstract. This paper describes a job distribution system which focuses on standard desktop worker nodes in inhomogeneous and unreliable environments. The system is suited for general purpose usage and supports both batch jobs and object-oriented interactive applications using standard Internet technologies. Advanced scheduling methods minimize the total execution time and improve execution efficiency, specialized to deal with unreliable failing worker nodes.

Keywords: Distributed computing, scheduling, monitoring, web services, desktop grids, parallel applications, organic computing, cloud computing.

1 Introduction

During the last 15 years, Personal Computers (PCs) spread more and more. Being irreplaceable in office and university service, they are also widely used in home areas. Today, a standard PC is equipped with a multi-core multi-GHz CPU and at least 2–4 MB of main memory. When used for its common dedication – office applications, email or web surfing – this power remains unused, because the box nearly all the time waits for user input.

In the course of the ubiquitous Internet almost every PC is connected to, many projects were built to utilize this idle time, BOINC [1] or zetaGrid [2], for example. To build up a local area desktop grid, there exist systems like Condor [3] or Alchemi [4] sometimes using flexible algorithms to distribute individual applications to a pool or desktop PCs. But they are not necessarily suitable to utilize the idle time of normal Internet-connected PCs, and, in general, not programming language independent.

The idea of JoSchKa (Job Scheduling Karlsruhe) [5] is to distribute the jobs in a request/response way: A central managing server has knowledge of about all jobs, while small autonomous background applications running on the worker-PCs query for them, similar to the tuple-spaces in [6]. When asked for a job, the server uses sophisticated (and in the scope of public resource/desktop computing unique) methods to select one suitable job matching the agent's characteristics, and responds to the agent what has to be done. The agent then has to download all specified files, execute the given command and transfer all result files back to the server (Fig. 1).

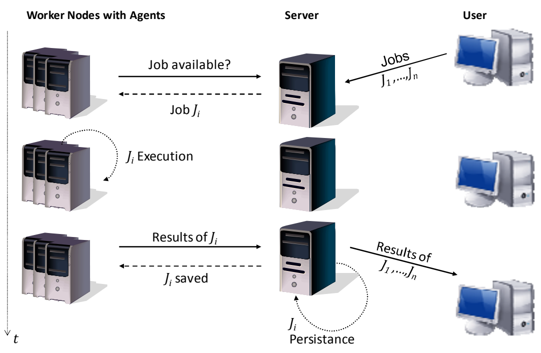


Fig. 1. JoSchKa principle

Typical features of JoSchKa are fully autonomous agents running on heterogeneous worker nodes and the support for any programming language (it has to be executable on the worker node, of course). The management communication and even the file transfer rely on HTTP only, there is no common file system. So the nodes can be run behind firewalls and NAT routers without problems. To support the users, there are both a graphical user interface for batch job upload and management, and a modern object-oriented API to develop interactive parallel applications.

In the following chapter, the server and its two mainly used distribution methods are described briefly. In the next sections, the agent and the user interface are described shortly, followed by its relation to organic computing and some practical experiences. The paper finishes with an outlook related to cloud computing.

2 Server

The server is the central component in the JoSchKa system. It has to deal with the up-/download of files, the management of the jobs and the decision, which one to select for a node if the agent running on this node polls for a job.

The file transfer interfaces used by the agents are standard HTTP-based interfaces, the ones for the user are SMB-based, due to convenient drag and drop file transfer. The most complex interface is the interface used to exchange management and control data, perform job queries, upload job description and so on. This interface is a SOAP-based webservice [7] using HTTP as transfer protocol.

The agents on the nodes address the SOAP-interface only. So they can be run behind firewalls and NAT routers. As a result, the server is not able to distribute the jobs or send other management data to the agents using a pushing-style but it has to react on queries by the agents and send commands as a response to these queries. If the server wants to tell something to an agent (*"execute this job:…"*), it has to wait for the agent performing a request for exactly this information (*"job available?"*). If an agent queries for a job, the server first creates a candidate list containing all jobs which are able to be run on the node. To this, the server performs some checks with every job in the database (the server manages a record of 22 values per job; the detailed description of the data model is omitted here):

- Do the platform specifications of the job and the agent match (operation system or memory requirements, for example)?
- Does the type of the job match with the type queried by the agent?
- All specified source files are physically available for download?
- If the job has predecessors defined, are they all finished?

If all tests evaluate to true, the job is added to a candidate list from which the scheduler has to select one single job. But before describing the selection process, it is necessary to explain why it is even a problem to select a suitable job and why not just selecting any job. Many publications (like [8, 9]) are dealing with intelligent scheduling algorithms, but they all are running on a reliable cluster or a super computer and generally have full knowledge about the nodes and the jobs' runtime requirements. None of the common desktop job distribution systems like BOINC, Condor or Alchemi try to observe the nodes and to exploit this additional knowledge, especially to improve the total system performance. Here, we have to distribute jobs to heterogeneous, unpredictably failing nodes. So other methods are needed, especially a monitoring of the nodes is essential. These methods are described in the following sections.

2.1 Fault Tolerance

When using a dedicated cluster, one can be sure that every node executes its jobs without failing before the current running job is finished. But when the nodes are part of an unsecure pool or if they are used as standard desktop computers we have to face with failures and unfinished jobs, because users normally shut down or reboot the PCs. So there are some mechanisms integrated to counteract this problem:

- The upload of intermediary results from the agent to the server and
- a heartbeat signal from the agent to the server.

The user has to activate the intermediary upload explicitly. If active, the agent uploads all new or changed result files in periodic intervals. So one can decide if a job has to be stopped or – in case of a node failure – started again. But this decision hast to be made by the user and of course depends on the characteristics of the problem the jobs are working on. If the user does not block a failed job explicitly, the server starts it again after a certain time period. For every job, the server manages an internal counter, which is periodically decreased by 1. Every time the agent which executes the job sends the periodic heartbeat signal, the counter is set back to its maximum value. If the node with the agent shuts down, the heartbeat for this job will be missing and the counter reaches zero. Then the server creates a new ID for the job and marks it available for a new run. Due to programming mistakes or other software errors it may happen that jobs fail even on reliable nodes. A special mechanism takes care of restarting only for a finite number of times. Jobs which are failing too often are blocked automatically by the server until the user fixes the problem and removes the lock manually. The acceptable number of failures until auto-blocking can be changed by the user.

2.2 Node Monitoring

A central assumption during the design of JoSchKa was the heterogeneity of the nodes. They not only differ with respect to their hard- and software configuration, but also in their reliability characteristics. Some nodes process every job successfully but others break down in an unpredictable manner. They may also change their reliability or switch from unreliable to robust and vice versa. Imagine a PC in a student pool, which runs nonstop for many hours in the night, but gets rebooted very often at daytime, when students use this PC.

To assess the behavior of the nodes a monitoring component was developed. The goal is to reach the ability to predict the behavior of the node in the near future [5]. Amongst others, the following values are monitored and calculated for each node, respectively:

- B: The relative CPU power of the node compared to the others, $B \in \mathbb{R}$.
- avU: The average uptime in wall clock minutes, $avU \in \mathbb{N}$.
- acU: The current uptime in wall clock minutes, $acU \in \mathbb{N}$.
- R: A synthetic reliability index of the node, $R \in [-1,1]$.

In the following, let $EWA(v_n, \dots, v_1)$ be the exponential weighted average, which processes the single input values v_i considering their age. It is defined recursively: $EWA(v_n, \dots, v_1) = \tilde{v}_n = \alpha v_n + (1-\alpha)\tilde{v}_{n-1}$ with $\alpha \in [0,1]$ and $\tilde{v}_1 = v_1$. The v_i are sorted by age, v_n is the youngest. More formally, the values for a single node $i \in \{1, \dots, l\}$ are calculated as follows:

- $B_i = \frac{\sum_{j=1}^{l} rB_j}{l \cdot rB_i}$, where rB_i denotes the time span (in wall clock milliseconds) the node needs to process a fixed arithmetic-logical benchmark. This benchmark is done by the node in periodic intervals.
- $avU_i = EWA\left(u_{t_u}, \dots, u_1\right)$, where the u_j are the nodes' last t_u uptimes.
- $acU_i = T_{now} - T_i$, where T_{now} denotes the current wall clock time and T_i denotes the starting wall clock time of the node.
- $R_i = EWA\left(r_{t_r}, \dots, r_1\right)$, where the r_j are reliability criteria for the last t_r processed jobs. If a job is done successfully, then $r_j = 1$, if failed $r_j = -1$.

In practice, we chose $t_u = t_r = 10$ and $\alpha = 0.25$. These data which are collected by the server for each single node can be summarized as follows: For each node, we now know,

- how fast it can work compared to all other nodes (B),
- how long it is available (on average) before it breaks down (avU),
- how long it is up since its last boot (acU) and
- how reliable it behaved when processing the last jobs (R).

This knowledge is used by the different distribution strategies, which are described in the following section.

2.3 Basic Job Distribution Strategies

As pointed out, the JoSchKa system is not designed for specialized clusters (although it can be run there too, of course), but to distribute jobs within a pool of strongly varying nodes. The nodes differ in performance but mainly they differ in their reliability. It is possible that a node works perfectly reliably for many days, but gets suddenly rebooted repeatedly and unpredictably. Then, long running jobs can't be processed any more on such a node. On the other hand, it would be a waste of valuable uptime, if a reliable node just processes jobs which are finished after a few minutes. So, if the distribution system disposes of many jobs which belong to many different users, it's obvious to distribute the jobs in an intelligent way. The goal is to minimize the breakdown-caused waste of processing time and to be concerned with fairness to the users, too. No user should feel disadvantaged.

At first, we describe the basic distribution strategies, in the following section (simulation) they are evaluated. JoSchKa uses various (combined) strategies, for lack of space we only show two of them here: The simplest algorithm and the one which normally is used in practice.

(a) Balanced (fair) distribution: Every user gets the same share of nodes to process his jobs.

(b) Uptime-based distribution: The decision, which job is selected for a polling agent is based on the agent's uptime behavior.

Normally, the scheduler does not differentiate between individual users, but single $JobTypes$. A job type is a user-defined subset of his jobs. So the different needs of an individual user who has his jobs grouped to probably more than one job type can be satisfied much better. If the number of known agents is less than the number of types, the scheduler does differentiate only the users, the different job types of the same user are aggregated to guarantee a fair distribution. In the following it is assumed that the system has knowledge of about l nodes and n jobs to be processed. The jobs belong to m individual types or users, respectively. Let $l, n, m \in \mathbb{N}$ and $m \leq n$. Furthermore

- J_i, $i \in \{1, \dots, n\}$ denotes job i,
- JT_k, $k \in \{1, \dots, m\}$ denotes job type k (or user k, if $l < m$),
- $jT: \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ with $jT(i) = k$ denotes the type index of job i and
- avT_k denotes the (exponentially weighted) average runtime of the finished jobs belonging to type JT_k, measured in wall clock minutes.

Balanced Distribution

Let $nrWORKING_k \in \mathbb{N}$ be the number of jobs of type JT_k, which are running when a node asks for a job. The scheduler selects the job i which is minimizing $nrWORKING_{jT(i)}$. In other words, a node always gets a job of the type with the least jobs running.

As a result, at any time every type/user gets the same number of nodes to work for it/him. This fairness only shifts when a user has not enough unprocessed jobs to keep the balance.

Uptime-based Distribution

As the name of this strategy suggests, this selection algorithm mainly considers the uptime values of the nodes. In practice, a typical node is available for a certain time period, but then it gets shut down with increasing probability. This proceeds every day in the same (or similar) manner. The values avU_i and acU_i, which are determined for each node i help to predict how long an asking node will still be available. From these values, a further value, $avTARGET$, is calculated. The closer the current uptime gets to the average uptime, the shorter are the jobs which have to be selected. If the node's uptime exceeds its average uptime, the scheduler little by little selects longer-running jobs, depending on the node's total reliability. To account for the possible different CPU performances among the nodes, the relative benchmark index is finally integrated to scale the average runtime. In other words, $avTARGET$ estimates how long the average runtime avT_k of a job type k maximally should be, to get a job of this type successfully done by the considered node before it becomes unavailable.

$$avTARGET' = \begin{cases} |avU_i - acU_i|, & acU_i \leq avU_i \\ (R_i + 1) \cdot |acU_i - avU_i|, & acU_i > avU_i \end{cases} \quad \text{with } R_i \in [-1,1]$$

$$avTARGET = avTARGET' \cdot B_i$$

To estimate, which job type has to be selected, the average type runtimes are sorted (having $avT_i < \ avT_j, i < j$) and the corresponding mean values are calculated:

$$avTM_k = \tfrac{1}{2}(avT_k + avT_{k+1})$$

Finally, the ultimate run length target value $RLTV$ is calculated:

$$RLTV = RLTV' + rnd(-2,2) \quad \text{where}$$

$$RLTV' = \begin{cases} avT^*, & |avTARGET - avT^*| < |avTARGET - avTM^*| \\ avTM^*, & |avTARGET - avT^*| \geq |avTARGET - avTM^*| \end{cases} \quad \text{with}$$

$$|avTARGET - avT^*| = \min_{k-1} |avTARGET - avT_k| \quad \text{and}$$

$$|avTARGET - avTM^*| = \min_{k} |avTARGET - avTM_k|$$

In other words, from the total set of average job type runtimes and their pair-wise mean values the single value $RLTV'$ is selected which is closest to the previous calculated ideal value $avTARGET$. Then the individual job J_i is selected and delivered which minimizes $\left|RLTV - avT_{jT(i)}\right|$. For clarity, the whole strategy is explained by an example now. We assume jobs of four types/users $JT_1, \dots, JT_4$ being in the data base. We also assume the following average runtimes (in minutes):

$$avT_1 = 10, \qquad avT_2 = 60, \qquad avT_3 = 180, \qquad avT_4 = 200$$

Furthermore, the following monitoring values of an asking node are assumed to be estimated so far:

$$avU = 240, \qquad acU = 220, \qquad B = 2, \qquad R = 0$$

This results in $avTARGET = (240 - 220) \cdot 2 = 40$. The average job runtime closest to this value is $avT_2 = 60$, the closest mean value is $avTM_1 = \frac{1}{2}(avT_1 + avT_2) = 35$. So we get $RLTV' = 35$. Because of $RLTV = RLTV' + \text{rnd}(-2{,}2)$, the node gets a job of the (randomly chosen) type JT_1 or JT_2. Fig. 2 explains the example.

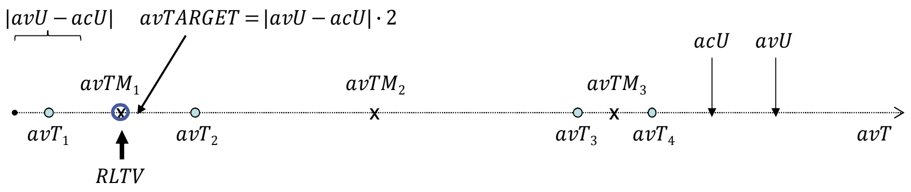


Fig. 2. Uptime-based distribution example

Due to the usage of the mean values and the randomization, the system performs a stochastic selection between the two best suited job types, if the ideal target value $avTARGET$ resides in the mean of two neighbored job types. The example shows also, why the largest average runtime (avT_m, in the example avT_4) is excluded when calculating $RLTV$. Otherwise, the nodes with the largest remaining uptime would work for the user with the longest running jobs only. In other words, a node gets

- a job of type JT_k if $avTARGET$ lies in the interval I_k or
- a job of type JT_k or JT_{k+1} (selected randomly), if $avTARGET$ lies in the interval $I_{k,k+1}$.

The interval bounds for the example above are graphically shown in Fig. 3. In general, the intervals are described as follows (m types, $k \in \{1, \dots, m-1\}$):

- $I_k = \left[\frac{1}{2}(avTM_{k-1} + avT_k), \frac{1}{2}(avT_k + avTM_k)\right]$, where
$$I_1 = \left[0, \frac{1}{2}(avT_1 + avTM_1)\right]$$
- $I_{k,k+1} = \left[\frac{1}{2}(avT_k + avTM_k), \frac{1}{2}(avTM_k + avT_{k+1})\right]$, where
$$I_{m-1,m} = \left[\frac{1}{2}(avT_{m-1} + avTM_{m-1}), \infty\right[$$

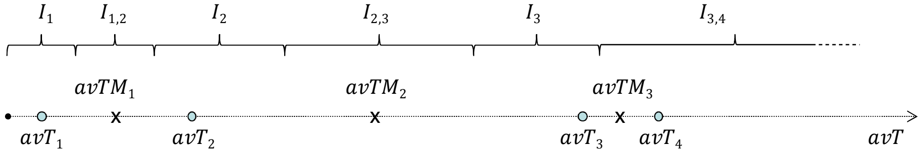


Fig. 3. Interval bounds example

2.4 Simulation

To test the strategies, a simulator was developed. It allows seeing how the system would behave in a specific job and node situation. Every simulated node has to be parameterized in XML:

- $power$: The bench value rB which the node estimates in the beginning (see 2.2) and sends to the server.
- $zerofp1$, $zerofp2$: The number of simulation steps the node is capable to work without failure.
- $incfp1$, $incfp2$: Within this time span the failure probability increases linearly (after the period of working reliable).
- $fail1$, $fail2$: The maximum probability (which is finally reached after the increasing period) for this node to fail during every simulation step.
- cnt: The number of nodes which are parameterized in this manner.

First, all nodes are simulated using the parameters $power$, $zerofp1$, $incfp1$ and $fail1$. After the first third of the simulation is done, all nodes switch to the second parameter set ($power$, $zerofp2$, $incfp2$ and $fail$). Hence, it is possible to check the behavior of the scheduling algorithm in case of a change in the reliability characteristics of a node. For clarity, the simulated characteristic is explained by an example:

```
<clients>
   <client cnt="5" power="4000"
           zerofp1="60" incfp1="40" fail1="3"
           zerofp2="20" incfp2="90" fail2="1"/>
   …
</clients>
```

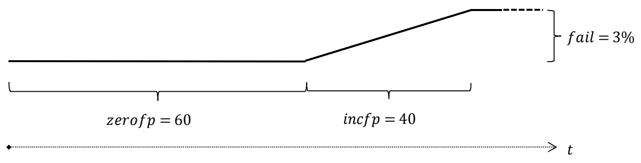


Fig. 4. Reliability behavior example

The five such configured nodes would perform as if they had done the initial benchmark in 4000 milliseconds. After startup, they would work 60 steps without failure and then, within a period of 40 steps, fail with increasing probability per step, up to a maximum value of 3% per step. After failing, the period would start again from the beginning with 60 new reliable steps. After the first third of the total simulation, the nodes would switch to the second parameter set (20, 90, 1).

Every simulated job type ($jobtype$) can be parameterized by the number of its jobs (cnt), the average duration ($jobduration$) of a job belonging to this type (if running on a node with $power = 5000$) and the number of simulation steps ($steps$) the simulator has to execute after adding these jobs:

```
<simulation>
   <step cnt="4000" jobtype="job_10" jobduration="10" steps="0"/>
   <step cnt="800" jobtype="job_50" jobduration="50" steps="0"/>
   <step cnt="200" jobtype="job_200" jobduration="200" steps="0"/>
   <step cnt="100" jobtype="job_400" jobduration="400" steps="5990"/>
</simulation>
```

Out of the many simulated configurations, the one shown above indicates perfectly (Fig. 5, Fig. 6), what one could achieve by using intelligent distribution strategies: Even in difficult situations (in practice, the real future behavior of the nodes and the real future runtimes of the jobs are fully unknown in principle!) the so called *makespan* is improved. The makespan describes the overall time needed to process all available jobs. We simulated a very heterogeneous set of 120 nodes (to save space, their XML specification is omitted here).

The following diagrams show for each job type the amount of jobs which are done at a specific simulation step. Every type produces a total workload of $jobduration \cdot cnt = 40000$ steps, so in an ideal distribution case all types should reach the 100%-line at the same time. This would state that every user gets the same amount of (successfully used) computing power.

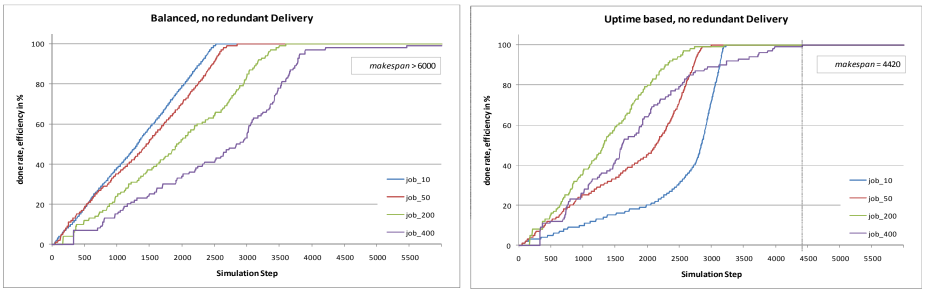
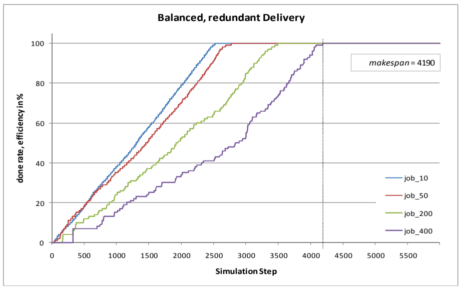


Fig. 5. Balanced and uptime-based distribution, no redundancy

Fig. 5 compares the balanced and the uptime-based distribution. Using the uptime-based strategy, the makespan improves significantly. It also explains how the uptime-distribution achieves this improvement: the short jobs are used to fill the remaining uptimes until the expected failure of the nodes.

But the diagrams also show a problem which typically occurs, when the system has to distribute long running jobs only. Then, they are processed by the not well-suited nodes, too. As the number of available jobs drops further, it could happen that reliable nodes will not get a job because all jobs are already running (maybe on the unreliable nodes). This is caused by the fact the server always delivers a job, independent on the polling node's quality. To guarantee the execution of the jobs by the reliable nodes, an additional, strategy-independent, redundant job execution was implemented: When an agent asks for a job and no free jobs are available, the server delivers one which is already running.



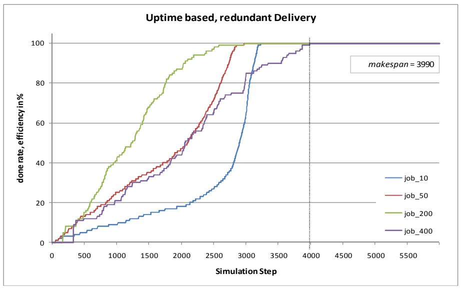


Fig. 6. Balanced and uptime-based distribution with redundancy

Fig. 6 shows the improvement achieved by the redundant delivery. The balanced strategy gains a large makespan improvement. The uptime-based mode improves too, but less significantly, because it finishes the long running jobs earlier, even without redundant delivery. So the negative effect mentioned above does not occur as strongly.

In practical scenarios, a mixture of the balanced and the uptime-based strategies, controlled by a parameter $fairlevel \in [0,1]$, is used:

(1) As necessary for the balanced strategy, for each job type JT_k, $nrWORKING_k \in \mathbb{N}$ (the number of running jobs of JT_k) is estimated.

(2) If $\min_k nrWORKING_k / \max_k nrWORKING_k < fairlevel$, the balanced strategy is used.

(3) Otherwise, the scheduler selects the best uptime-based job.

So the degree of balance/fairness is configurable and normally set to 0.1–0.3.

3 Agent and Usage

Initially, this section describes the autonomous component of the system, the agent which runs on the worker nodes, and then gives a short overview of the graphical tool needed by the user to manage his jobs.

3.1 Agent

The agent is the component that is running on the worker nodes. It requires no user-interaction and behaves totally autonomously, so it can be started as a background service, optionally with an integrated auto-updating mechanism, and performs the following tasks:

(1) Initially, it has to determine the local system parameters that are essential for job selection and execution. These are the installed memory, the CPU architecture (x86/x64), the availability of some runtime environments (.NET, Mono, Java, Python, Perl, GAMS), and the operating system (Windows or Unix-like).

(2) It starts the working loop with a query to the server for a job. If the server responds with a job description, the agent downloads the specified files.

(3) The agent starts the job with low operating system priority and waits for its completion. During the execution, it periodically contacts the server to submit a "heartbeat". Standard and error out are redirected, if desired by the user.

(4) When the job has finished with no error, all result files and the standard/error out data are uploaded to the server.

(5) If all files are transferred successfully, a final commit is sent to the server. Then, and only then, the job is accepted as successfully done and the agent proceeds by sending the next job query to the server (step (2)).

3.2 Parallelization and Management Tool

It is very easy for a developer to distribute concurrent jobs using JoSchKa. When using the batch system, the application has to be split in single pieces; each of them must (formally) be described by:

- a set of input files $I = \{i_1, \dots, i_n\}$,
- a command (or script) C to be executed on the worker node and
- a set of output files $O = \{o_1, \dots, o_m\}$ which are produced by the command or the script respectively.

If a problem is decomposable to such units $J = (I, C, O)$, and C is executable on the worker nodes, it can be distributed and run as batch jobs by the described system.

A user, who wants to create a more interactive parallel application, create jobs programmatically and react directly on their results, can use the JoSchKa thread API for .NET to create a parallel program. With this API it is possible to dynamically upload program code to the server, start threads executing this code and query their results. So, developing a distributed program is similar to the development of a multi-threaded local application.

Due to the lack of space, more details about the API and the graphical management tool are omitted, we just give a simplified API class diagram (Fig. 7) and a screenshot

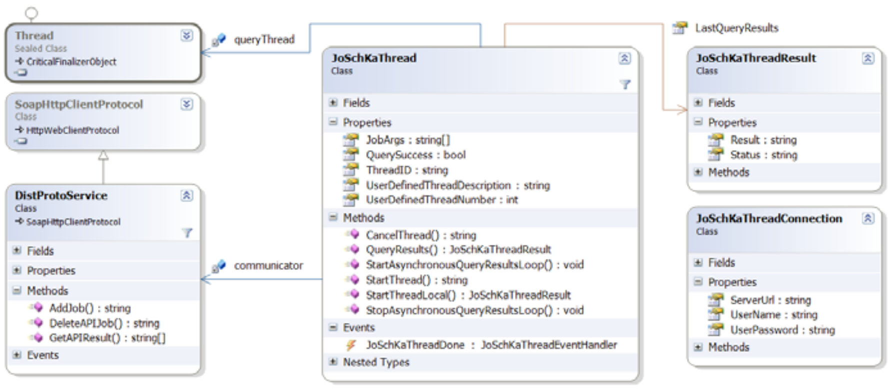
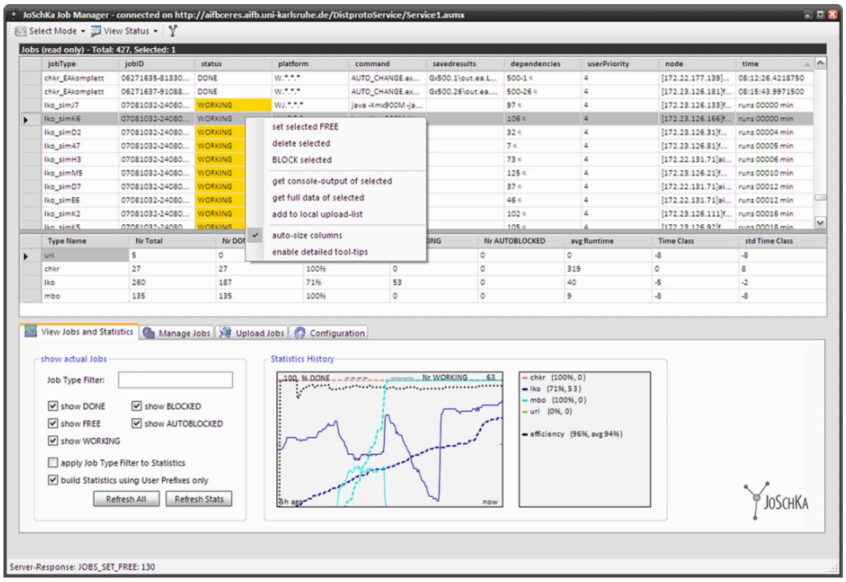


Fig. 7. The JoSchKa thread API classes


JoSchKa Job Manager - connected on http://aifbceres.aifb.uni-karlsruhe.de/DistprotoService/Service1.asmx

Jobs (read only) - Total: 427, Selected: 1

jobType	jobID	status	platform	command	savedresults	dependencies	userPriority	node	time
chkr_EAkomplett	06271635-81330...	DONE	W.*.*.*	AUTO_CHANGE.ex...	Gx500.1\out.ea.L...	500-1 «	4	[172.22.177.139]...	08:12:26.4218750
chkr_EAkomplett	06271637-91088...	DONE	W.*.*.*	AUTO_CHANGE.ex...	Gx500.26\out.ea....	500-26 «	4	[172.23.126.181]f...	08:15:43.9971500
lko_simJ7	07081032-24080...	WORKING	WJ.*.*.*	java -Xmx900M -ja...		97 «	4	[172.23.126.133]f...	runs 00000 min
lko_simK6	07081032-24080...	WORKING				106 «	4	[172.23.126.166]f...	runs 00000 min
lko_simD2	07081032-24080...	WORKING				32 «	4	[172.23.126.31]f...	runs 00004 min
lko_simA7	07081032-24080...	WORKING				7 «	4	[172.23.126.81]f...	runs 00005 min
lko_simH3	07081032-24080...	WORKING				73 «	4	[172.22.131.71]ai...	runs 00006 min
lko_simM5	07081032-24080...	WORKING				125 «	4	[172.23.126.21]f...	runs 00010 min
lko_simD7	07081032-24080...	WORKING				37 «	4	[172.22.131.71]ai...	runs 00012 min
lko_simE6	07081032-24080...	WORKING				46 «	4	[172.22.131.71]ai...	runs 00012 min
lko_simK2	07081032-24080...	WORKING				102 «	4	[172.23.126.111]f...	runs 00016 min
lko_simK5	07081032-24080	WORKING				105 «	4	[172.23.126.92]f	runs 00018 min

Type Name	Nr Total	Nr DO		ING	Nr AUTOBLOCKED	avg Runtime	Time Class	std Time Class
uri	5	0			0	0	-8	-8
chkr	27	27	100%	0	0	319	0	8
lko	260	187	71%	53	0	40	-5	-2
mbo	135	135	100%	0	0	9	-8	-8

Job Type Filter:

[✓] show DONE [✓] show BLOCKED
[✓] show FREE [✓] show AUTOBLOCKED
[✓] show WORKING
[ ] apply Job Type Filter to Statistics
[✓] build Statistics using User Prefixes only

Server-Response: JOBS_SET_FREE: 130

Fig. 8. Job management tool

of the frontend showing the state of the submitted jobs, their execution status and a context menu providing some management functions (Fig. 8). For more information, please refer to [5].

4 Relation to Organic Computing

Organic Computing (OC) [10] focuses on self-organizing systems adapting robustly to dynamically changing environments without losing control. It proposes a generic observer/controller architecture, which allows self-organization but enables reactions to control the overall behavior to the (technical) system [11]. We have both a system under observation and control (SuOC), and observing and controlling components, which are available for monitoring and influencing the system. The observer has to measure, quantify and predict further behavior of the monitored entities. The aggregated values are reported to the controller, who executes appropriate actions to influence the system. The overall goal is to meet the user's requirements. The user himself can influence the system by manual changing the objective function or by directly accessing the observed/controlled entities (Fig. 9).

The described situation of computational jobs, heterogeneous unreliable job executing worker nodes and a distribution system like JoSchKa is a perfect example for such an organic system. The user-given job data and the worker nodes represent the system under observation and control. In an autonomous way, they execute the jobs, while failing and rebooting in an uncontrollable way.

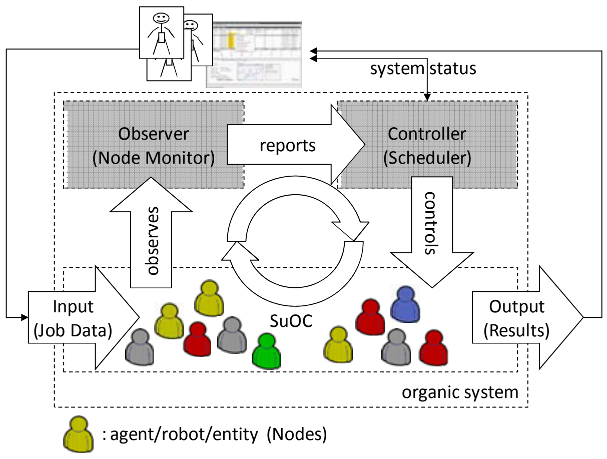


Fig. 9. The generic O/C architecture

The monitoring component of the JoSchKa tem has the role of an observer while the job selection component (scheduler) behaves as an OC controller. It does not control the nodes directly, of course, but with its intelligent job assignment (based on the observer's measured values and predicted node-behavior) it helps achieving all users' goals: The completion of all jobs (i. e. the production of result files) as fast and early as possible. The users can influence the system by manually blocking or deleting their jobs using the management GUI; but normally they do not have to, because the nodes and the distribution system act fully autonomous.

5 Practical Experiences and Outlook

Since 2006, JoSchKa runs on PCs of a computer pool of the Faculty of Economics and Business Engineering of the Karlsruhe Institute of Technology (KIT) and on some other small pools, completed by a few servers. The pool PCs are standard Windows or Linux desktops, equipped with 3 GHz CPUs and 1–2 GB of memory. Since then, the nodes finished about 250 000 jobs, mainly belonging to naturally inspired optimizing algorithms and organic computing simulations [12]. The students working interactively on the desktop did not notice the permanent load. As showed in [5], the execution time of standard desktop applications does not slow down when running a CPU-intensive background task with lowest process priority.

The students and the scientific staff of our faculty benefit well by JoSchKa: Students normally write their optimizing programs in Java on a Windows platform, as they learned it during their education. But the Unix-based HPC or Grid platforms of the computing centre require C/C++ or – even worse – Fortran, so the usage of them is quite difficult, especially at the end of a students' thesis work, when time runs out. The convenient handling of JoSchKa gave them the possibility to run the simulations concurrently on multiple worker nodes, even when a distributed execution was initially not intended.

Due to the raising availability and usage of cloud computing [13] providers (Amazon EC2/S3, Google AppEngine or Microsoft Azure, e. g.), it's imaginable to extend JoSchKa for cloud usage in the near future. If a user needs unsatisfiable computing resources, it should be possible to dynamically create them by using a virtual infrastructure as provided by a cloud service. The system tries to estimate how many new machines are necessary and how they have to be equipped (hard- and software), and tells the user the price for these virtual machines. If he agrees, the virtual machines are cloned from templates; configured according the user's needs, and started to execute the agents. So JoSchKa would not only react on the dynamic behavior of fixed machines, but also would extend/reduce them dynamically (and thus acting as a real intervening organic controller), if needed.

References

[1] Anderson, D.: BOINC: A System for Public Resource Computing and Storage. In: 5th IEEE/ACM International Workshop on Grid Computing, Pittsburg, PA, pp. 365–372 (2004)

[2] Wedeniwski, S.: ZetaGrid – Computations connected with the Verification of the Riemann Hypothesis. In: Foundations of Computational Mathematics Conference, Minnesota, USA, August 2002 (2008)

[3] Litzkow, M., Livny, M., Mutka, M.: Condor – a Hunter of idle Workstations. In: 8th International Conference on Distributed Computing Systems, pp. 104–111 (1988)

[4] Luther, A., Buyya, R., Ranjan, R., Venugopal, S.: Peer-to-Peer Grid Computing and a.NET-based Alchemi Framework. In: Guo, L.Y. (ed.) High Performance Computing: Paradigm and Infrastructure. Wiley Press, New Jersey (2005)

[5] Bonn, M.: JoSchKa: Jobverteilung in heterogenen und unzuverlässigen Umgebungen. Dissertation an der Universität Karlsruhe (TH), Universitätsverlag Karlsruhe (2008)

[6] Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)

[7] Dostal, W., Jeckle, M., Melzer, I., Zengler, B.: Service-orientierte Architekturen mit Web Services. Spektrum Akademischer Verlag (2005)

[8] Tsafrir, D., Etison, Y., Feitelson, D.: Backfilling using System-generated Predictions rather than User Runtime Estimates. IEEE Transactions on Parallel and Distributed Systems 18(6), 789–803 (2007)

[9] He, Y., Hsu, W., Leiserson, C.E.: Provably efficient two-level adaptive Scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (Hrsg.) JSSPP 2006. LNCS, vol. 4376, pp. 1–32. Springer, Heidelberg (2007)

[10] DFG Priority Program 1183 Organic Computing (2005), `http://www.organic-computing.de/SPP` (visited September 2009)

[11] Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for Organic Computing. In: Christian Hochberger and Rüdiger Liskowsky, INFORMATIK 2006 – Informatik für Menschen! GI-Edition – Lecture Notes in Informatics (LNI), vol. P-93, pp. 112–119. Bonner Köllen Verlag (2006)

[12] Richter, U., Mnif, M.: Learning to control the emergent Behaviour of a multi-agent System. In: Proceedings of the 2008 Workshop on Adaptive Learning Agents and Multi-Agent Systems at AAMAS 2008 (2008)

[13] Baun, C., Kunze, M., Nimis, J., Tai, S.: Cloud Computing: Web-basierte dynamische IT-Services. Auflage, vol. 1. Springer, Berlin (2009)

On Deadlocks and Fairness in Self-organizing Resource-Flow Systems*

Jan-Philipp Steghöfer[1], Pratik Mandrekar[2], Florian Nafz[1], Hella Seebach[1], and Wolfgang Reif[1]

[1] Universität Augsburg, Institute for Software- & Systems-Engineering, Augsburg, Germany
{steghoefer,nafz,seebach,reif}@informatik.uni-augsburg.de
[2] Birla Institute of Technology and Science Pilani, Goa, India
pratik.mandrekar@gmail.com

Abstract. Systems in which individual units concurrently process indivisible resources are inherently prone to starvation and deadlocks. This paper describes a fair scheduling mechanism for self-organizing resource-flow systems that prevents starvation as well as a distributed deadlock avoidance algorithm. The algorithm leverages implicit local knowledge about the system's structure and uses a simple coordination mechanism to detect loops in the resource-flow. The knowledge about the loops that have been detected is then incorporated into the scheduling mechanism. Limitations of the approach are presented along with extension to the basic mechanism to deal with them.

1 Introduction

In a resource-flow system agents handle resources by receiving them from another agent, processing them and handing them over to another agent. An instance of this are flexible manufacturing systems. If agents can only process one resource at a time and there are no buffers available, such systems are prone to deadlocks. If the agents are arranged in a way that an agent can receive a resource it had already processed before, the resource has been processed by agents arranged in a loop. This loop can fill up with resources, in a way that the agent can not give its currently held resource to another agent and can thus not accept a new resource. Such a situation is depicted in Fig. 1.

This paper introduces a deadlock avoidance mechanism that leverages the implicit knowledge available to the agents in self-organizing resource-flow systems modelled with the Organic Design Pattern (ODP) [16] and incorporates the mechanism into a fair scheduler for this system class. The local knowledge of the agents along with the structure that is given by the definition of the resource flow allows the algorithm to be efficient both in terms of messages sent and in computational time required while effectively avoiding deadlocks and preventing

* This research is partly sponsored by the priority program "Organic Computing" (SPP 1183) of the German research foundation (DFG) in the project SAVE ORCA.

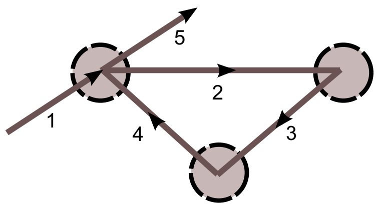


Fig. 1. A situation prone to deadlocks induced by cyclic resource processing

starvation without the need to globally analyse the system or trace the state of all agents during runtime.

The paper is structured as follows: Sect. 2 defines the terms associated with deadlocks and introduces how deadlocks can be dealt with. Sect. 3 shows how liveness hazards are dealt with in literature and Sect. 4 describes the system class the proposed approach is applicable to. Sect. 5 then introduces a fair role-selection algorithm that is extended with a deadlock avoidance mechanism in Sect. 6. Limitations of this mechanism and ways to overcome them are outlined in Sect. 7 and Sect. 8 concludes the paper.

2 Background and Definitions

There are a number of different liveness hazards that can occur in concurrent systems. The definitions of these hazards often vary a great deal between different authors and domains. Therefore, the following brief definitions introduce the terms as they are used in this paper.

Deadlocks. A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. Coffman et al. [10] give four conditions that must hold for a deadlock to occur:

1. "Mutual exclusion" i.e., agents claim exclusive control of the resource they require;
2. "No preemption" i.e., resources cannot be forcibly removed from the agents holding them until processing of the resources is completed;
3. "Wait for condition" i.e., Agents hold resources allocated to them, while waiting for additional ones;
4. "Circular wait" i.e., a circular claim of agents exists, such that each agent holds one or more resources that need to be given to another agent in the claim.

Livelock. A liveness failure is referred to as a livelock when an agent although not blocked cannot proceed further and keeps retrying. As an example consider two mobile agent's who obstruct each others paths. Both decide to give way to the other at the same time and move to the right leading again to an obstruction. Now both decide at the same time to move to the left and the process keeps repeating leading to no agent moving forward. Another example is the 'after you after you' politeness protocol.

Starvation. Starvation occurs when an agent is continuously denied access to a resource. Bustard et al. illustrate the starvation problem using the dining philosophers problem in the field of autonomic computing [8]. It can also be characterized as a special case of livelock [21].

Fairness. Fairness in its most general definition guarantees that a process will eventually be given enough resources to allow it to terminate [26]. This is usually achieved by implementing a scheduling mechanism. Especially in distributed systems that share resources, the ability to terminate may depend on other components or on interacting with them. Fairness is classified into weak and strong fairness where weak fairness implies that a component will eventually proceed if it can almost always proceed and strong fairness implies that a component which can proceed infinitely often does proceed infinitely often [11].

There are three different ways on how potential deadlocks can be dealt with. Their applicability depends on the structure of the systems under consideration and the assumptions that can be made about them.

Deadlock Prevention prohibits circular waits among agents while the system is not running. The strength of the approach is also its greatest weakness: a prevention mechanism requires global knowledge of the system and all its reachable states which often yields a straightforward often-simple control law but acquiring this knowledge can be difficult and computationally intractable. Additionally, limiting concurrency to prevent any deadlock state from occurring can be overly conservative, potentially leading to low utilization of system resources.

Deadlock Avoidance suitable governs the resource flow to prevent circular waits. This is a dynamic approach that can utilize the knowledge of the current allocation of resources and the future behaviour of processes to control the release and acquisition of resources. The main characteristic of deadlock avoidance strategies is that they work during runtime and base their decisions on information about the resource and agent status. More precisely, a deadlock controller inhibits or enables events involving resource acquisition or release by using look-ahead procedures.

Deadlock Detection and Resolution monitors the agents and/or the flow of resources and detects deadlocks at runtime. Resources can then be removed from the system or put into buffers to resolve the deadlock. This strategy can in general allow higher resource utilization than that prevention or avoidance methods can offer. However, it should only be used when deadlock is rare and detection and recovery are not expensive and much faster than deadlock generation.

3 Related Work

Deadlocks have been dealt with in several domains that are concerned with software-systems, e.g. deadlocks which are inherent in databases that use locking mechanisms [5] and involve detecting cycles in the wait-for graphs when the dependency relations between agents are known locally by using serialization techniques or resource allocation logic in operating systems [10]. However, the primary realm of related work that is relevant for the self-organizing resource-flow systems considered in this paper stems from the domain of manufacturing control and Flexible Manufacturing Systems (FMS). For an overview of such approaches, see, [14].

Several proposed **Deadlock Prevention** strategies are based on the use of transition systems, mostly Petri nets. [19] provides a mechanism to generate an automated supervisor to check for deadlocks in the given Petri nets and hence prevent them from occurring. A survey of techniques using Petri nets has been made by Zhi Wu Li et al. [23] with respect to AGV (Automated Guided Vehicle) systems in manufacturing. Siphon based policies are shown to achieve a good balance of the best of all properties of the Petri net based methods [18]. However all these techniques rely on known interactions between the agents (mostly the AGVs in the referred cases) and restrict transitions to states that could lead to a deadlock based on a universal control policy generated during Petri net analysis.

The methods discussed above all require a global view of the system, offline computation or computation during the design phase. The deadlock prevention approach establishes the control policy in a static way, so that, once established, it is guaranteed that the system cannot reach a deadlock situation if the system is not changed [13]. Serial execution of the processes on the resource like scheduling in the case of software processes or threads involves assumptions or knowledge of the time of execution which might not be available. The computations required can become infeasible for large scale systems.

Deadlock Avoidance mechanisms using Petri nets [33,17] restrict the maximal number of parts that can be routed into the system. The Petri nets are either constructed upfront or constructed dynamically which involves a communication overhead. A Petri net allows to check at each state whether the next state would lead to a deadlock or not. This analysis can be used to obtain suitable constraint policies for the system [3,33]; a similar mechanism has also been utilized for deadlocks in software programs [30] using supervisory control of Petri nets. A resource-allocation graph can be used to construct a local graph to check for deadlocks based on a global classification of agents into deadlock risk and deadlock free agents and combines the restrictions of deadlock prevention with the flexibility of deadlock avoidance [34]. [6] also uses a resource-allocation graph in which resource allocation and release is modelled by the insertion and deletion of edges. Deadlock avoidance is achieved by prohibiting actions that would lead to the insertion of an edge that would make the graph cyclic. If the components of a system know the nature of all the jobs they will be performing and all the components they will be communicating with in advance, deadlocks can be avoided in a distributed fashion [27].

Deadlock Detection and Resolution [28] often involves construction of a wait-for graph [7]. A wait-for graph is a graph to track which other agent an agent is currently blocking on. Maintaining such a global picture or agent state involves a lot of communication overhead while locally distributed deadlock detection schemes like Goldman's Algorithm [15] pass information locally to agents in the form of tables or other data sets. Ashfield et al. [2] provide dedicated agents for deadlock initiation, detection and resolution in a system with mobile agents. Deadlock Recovery Mechanisms using buffers [32] or rollback propagation [31] have been proposed.

Livelocks are closely related to deadlocks [1] but have not been dealt with as rigorously as deadlocks have been. A combination of different formal methods is used in [9] to check a system for livelock freedom at design time. Similarly, [20] introduces a type system that can be used to prove that processes and their communication are deadlock- and livelock-free.

In [22], the author examines different characterizations of **fairness** in trace-based systems and explores the practical usefulness of generalized fairness properties. [4] introduces least fairness, a property that is sufficient to show freedom of starvation as well as the concept of conspiracy that describes how concurrent processes interact to induce starvation. Parametric fairness, a fairness measure based on a probabilistic model is used in [29] to show that most executions of concurrent programs are fair. Markov Models are used to study unfairness and starvation in CSMA/CA protocols [12].

4 Self-organising Resource-Flow Systems

In order to analyse, specify, model, and construct self-organising resource-flow systems, the Organic Design Pattern (ODP) has been developed [16]. It describes self-organizing systems in terms of *agents* that process *resources* by applying capabilities according to *roles*. Roles are data structures that contain a precondition, a list of *capabilities* to apply and a postcondition ODP is part of a methodology that contains a design guideline, the pattern description with its static and dynamic parts as well as methods and tools for formally verifying properties of systems modelled with ODP. In the following, only the relevant parts of the pattern will be described. More information can be found in the cited resources.

4.1 Set-Based Description of ODP

An ODP system consists – among other things – of a set *agents* of the agents in the system, a set *capabilities* which is the set of all capabilities in the system and a set *roles* which denotes the set of all possible roles in the system. An agent $ag \in agents$ consists of two sets of agents ($input_{ag}$ and $output_{ag}$), the former one defining the agents from which ag may accept resources and the latter one to which ag may give resources, as well as a set cap_{ag} of capabilities the agent can apply and a set $roles_{ag}$ which determine the function of the agent in the system. A condition c is an element of the set *conditions*, containing an agent to exchange a resource with, a sequence of capabilities[1] that have been applied to a

[1] In this definition, *capabilities* is treated as an alphabet whose symbols can form words which describe the state, task and capabilities to apply to a resource.

resource and a sequence of capabilities that have to be applied to an agent (also called the *task* to be performed on the resource). A role $r \in roles$ is composed of a precondition that describes where a resource comes from, its state and task when it is accepted by the agent, a sequence of capabilities to apply on the resource, and a postcondition that describes to which agent the resource has to be given and its state and task after processing.

ODP systems are reconfigured by changing the allocation of roles to agents. Whenever an agent can not fulfil its capabilities any more or is no longer available, a new allocation of roles to the agents is calculated (e.g. by employing constraint satisfaction techniques [25]) and the newly calculated roles are adopted by the agents. This ensures that as long as all the capabilities that are required to process resources are still available in the system, a valid configuration can be found.

The role that has been applied to process a resource by agent ag_{proc} is also used to determine which agent ag_{next} it has to be given to. ag_{proc} sends ag_{next} a message informing it of the intent to transfer a resource. The message contains information about the resources status that can be used by the ag_{next} to determine the role whose precondition fits the resource status and the sending agent. The role that has been derived is then stored in a map in which the ag_{proc} is the key and the role is the value. The set of values in this map is used at a later point to select the next role to be executed by ag_{next}. Sect. 6.2 describes the message reception and role selection process formally.

4.2 Liveness in Self-organising Resource-Flow Systems

Preventing starvation of an agent is straight-forward: it has to be ensured that the resource the agent holds is eventually taken by another agent. This can be ensured by a role-selection mechanism that will eventually select each applicable role. Dealing with deadlocks, however, is a more complicated matter.

Systems with a flow of physical resources are prone to deadlocks whenever they are in a configuration that establishes a cycle in the resource-flow graph. In Sect. 2, four different characteristic conditions for deadlocks were mentioned. The first three of these (mutual exclusion, no preemption, and wait-for condition) are always true in ODP systems as resources can not be shared between agents but are claimed exclusively by an agent for processing, processing of a resource is always completed before it is available to another agent, and an agent has to wait for a subsequent agent to take the resource away before it can process another one. Thus, the only way to deal with deadlocks in ODP systems is to break the "circular wait" property. A circular wait occurs when two or more agents are arranged in a loop and each agent waits for the subsequent agent to take the resource it has been offered.

All centralized approaches of deadlock prevention and avoidance suffer from the same limitations: pre-calculating all states a system can reach is very expensive and in many cases computationally infeasible even for relatively small cases. Supervising the state of all agents involves a massive amount of communication and limits the agents' autonomy considerably. Additionally, ensuring a consistent state to make decisions about applicable state transitions is by no means simple and may require the introduction of synchronization between agents, thus severely reducing system performance. A centralized solution also always suffers from standard problems: it is a bottleneck and a single point of failure. Finally, a distributed solution can be employed regardless of an existing centralized entity, that, e.g., handles reconfiguration.

Deadlock detection might be feasible in resource-flow systems as there are distributed mechanisms available (see [28] for a survey). However, the resolution mechanisms proposed are hardly adaptable to resource-flow systems. Rollback propagation [31] is not applicable since resources undergo an irreversible set of operations. Use of buffers [32] has to be avoided as in many applications agents can not store resources and it can not be assumed that there are spare agents to temporarily hold resources. A conceivable mechanism for deadlock resolution is to let an agent dump the resource it currently holds, which is undesirable because resources are potentially valuable and should be processed to completion.

5 A Fair Role-Selection Mechanism

The main purpose of a fair role-selection mechanism is to avoid starvation. As described above, starvation occurs if an agent is waiting for a resource to be taken by another agent and this never happens. In ODP terminology, the agent ag_{rec} that is the receiver of the resource never executes the role r_x that would take the resource from the sending agent ag_{sen}. The scheduling algorithm described below thus ensures that each role that is applicable is eventually executed.

Definitions and Initialization: An agent contains the following data structures: a map of roles associated to agents that sent a request to the current agent $requests \subseteq agents \times roles$, a map $applicationTimes \subseteq roles \times \mathbb{N}$ that stores the value of the counter at the point in time when a role has last been executed and contains one value per role $(\forall (r_1, t_1), (r_2, t_2) \in applicationTimes : r_1 = r_2 \rightarrow t_1 = t_2)$, and a number $counter \in \mathbb{N}$ that counts the number of times the agent executed a role. These structures are initialized as $requests = \emptyset$, $applicationTimes = \{\forall r \in roles | (r, 0)\}$, and $counter = 0$.

Reception of Message: Whenever an agent ag_{sen} is ready to transfer a resource to an agent ag_{rec}, ag_{sen} sends a message $m = (ag_{sen}, c)$ where c is the postcondition of the role that ag_{sen} executed. ag_{rec} executes a function $getRole : agents \times conditions \rightarrow roles^{+}$ as $appRoles = getRole(ag_{sen}, c)$ where

$$\begin{aligned} getRole(ag_{sen}, c) = \{r : \; & r \in roles_{ag_{rec}} \wedge \\ & r.precondition.from = ag_{sen} \wedge \\ & r.precondition.state = c.state \wedge \\ & r.precondition.task = c.task\} \end{aligned}$$

and updates the $requests$ set: $requests = requests \cup \{(ag_{sen}, r)\} \forall r \in appRoles$. Each tuple in $requests$ is also associated with the timestamp of its reception which can be retrieved by $ts : agents \rightarrow \mathbb{N}$.

Choose a Role: Whenever ag_{rec} has to decide which role to execute next, it goes through *requests* and applies for each of the roles stored in it a fitness function $f : Roles \rightarrow \mathbb{N}$. In its simplest form, f is instantiated as $f(r) = (counter - t)$ where $(r, t) \in applicationTimes$. This will yield the highest value for the role that has not been executed the longest, thus ensuring that each role that is applicable will eventually be executed if $f(r)$ is maximized.

Formally, the selection of the role to execute is done by evaluating a function $max : (agents \times roles)^+ \rightarrow (agents \times roles)$ which yields $(a, r) = max(requests)$ where

$$\begin{aligned} max(requests) \;=\; & (ag, r_{next}) \text{ if } \forall (ag', r') \;\in\; requests : \\ & f(r') < f(r_{next}) \vee (f(r') = f(r_{next}) \;\wedge\; ts(ag) \leq ts(ag')) \end{aligned}$$

The agent then chooses r_{next} as the next role to execute. The incorporation of ts ensures that even if two or more requests evaluate to same fitness value, max yields an unambiguous result, namely the oldest request.

Update: Once a role has been selected for execution, the agent needs to update its structures to reflect the changes by setting $t = counter$ where $(r_{next}, t) \in applicationTimes$, $requests = requests \setminus \{(ag, r_{next})\}$, $counter = counter + 1$.

6 Distributed Deadlock Avoidance

The basic principle of the deadlock avoidance algorithm is very simple: If a loop is detected, determine the size of the loop n and allow only $n - 1$ resources to be processed by the n agents that are part of the loop at any one time.

The algorithm described below, however, constitutes a kind of local simulation of the surroundings of an agent and works in a distributed fashion. It is therefore applicable for systems with a centralized *and* a distributed configuration mechanism. Thus, it is more versatile than a centralized loop detection mechanism, but suffers some limitations because of its distributed nature and the limited knowledge that is available at the agents.

6.1 Distributed Loop Detection

The algorithm is split in two parts: The first part is run locally on each agent after it has received a new role allocation (i.e. after it has been reconfigured). It determines whether the agent *potentially* is the entry of a loop. The second part then verifies this assumption by sending a message that will eventually reach a sink – meaning that the agent is not part of a loop – or return to the sending agent – meaning of course that the agent is part of a loop.

Loop Estimation: Define a set $S : roles \times roles$ as follows:

$$\begin{aligned} S = \{(r_i, r_j) \mid\; & r_i.precondition.state \sqsubseteq r_j.precondition.state \;\wedge \\ & r_i.precondition.task = r_j.precondition.task \;\wedge \\ & r_i, r_j \in roles \wedge r_i \neq r_j\} \end{aligned}$$

where '$\sqsubseteq$' is the list prefix operator. Also define a function *min* as

$$min(S) = abs(|p_q| - |p_r|), \text{ if } \forall\, (p_k, p_l) \in S : abs(|p_q| - |p_r|) \;\leq\; abs(|p_k| - |p_l|)$$

where $p_i = r_i.precondition.state$, for $i \in \{k, l, q, r\}$. This allows to determine whether a loop exists and give an estimate of the length of the loop:

$$|S| \geq 1 \Rightarrow loop = \texttt{true} \wedge n_{est} = min(S)$$

This algorithm also yields an estimate n_{est} for the number of agents in the loop that is an underestimate when we assume that each agent applies at most one capability.

Loop Determination: To determine the actual size of the loop and to eliminate potential loops the agent ag_{org} creates a message m that has a list of agent identifiers m_{agents} initialized with the identifier of ag_{org}, a boolean flag m_{sink} that determines if the message has reached a sink and that is initialized with `false`, the postcondition of role r_j in $m_{condition}$, and the tuple $(r_i, r_j) \in S$ in m_{roles}. The message is sent to the agent in the output of the condition[2].

When an agent ag_i receives a loop-detection message, it uses the *getRole*() function to find the role that would be chosen if the sending agent had delivered a request to take a resource. ag_i then adds its own identifier to m_{agents}. If ag_i is not a sink, i.e. the role chosen by ag_i does contain a postcondition, it replaces $m_{condition}$ by the postcondition of the selected role and forwards the message to the agent that is set as the output in the chosen role. In case ag_i is a sink, it sets m_{sink} to true and returns the message to its originator, i.e., the first entry in m_{agents}[3].

Eventually, the message will return to the original sender ag_{org}, either because it passed through the loop (in which case m_{sink} will be `false`) or because it reached a sink and was returned. If a sink was reached, a loop does not exist and S can be updated: $S = S \setminus \{m_{roles}\}$. Otherwise, the actual length of the loop is determined by counting the elements in m_{agents} and updating the agent-global n that contains the number of agents in the smallest loop the agent is part of. n is updated only if the number of elements in m_{agents} is less than the current n.

The loop determination part of the algorithm terminates when all messages send out by ag_{org} have returned to ag_{org}.

6.2 Extension of the Scheduling Mechanism to Avoid Deadlocks

If a minimal loop size n has been calculated, the role-selection algorithm presented in Sect. 5 can be extended as follows:

[2] We assume that agents operate in a stable communication environment and message loss is thus not considered.

[3] In case direct communication is not possible, the message is passed back along the way it reached the sink. If an agent ag_j receives a message with m_{sink} set to `true`, it looks up the agent that is in front of ag_j's agent identifier in m_{agents} and sends the message to this agent.

1. Define $\mathit{executions} : \mathit{roles} \times \mathit{roles} \times \mathbb{N}$
2. Initialize $\mathit{executions} = \{\forall s \in S | (s, 0)\}$
3. Choose:
$$f'(r) = \begin{cases} 0, & \text{if } \exists((r_i, r_j), x) \in \mathit{Executions} : r_i = r \wedge x \geq n - 1 \\ f(r), & \text{otherwise} \end{cases}$$
4. Update:
$$\forall((r_i, r_j), x) \in \mathit{executions} : \begin{cases} x = x + 1 & \text{if } r_{next} = r_i \\ x = x - 1 & \text{if } r_{next} = r_j \end{cases}$$

The fitness function $f'(r)$ ensures that role r_i is only executed if the difference between the number of times r_i and r_j have been executed is less than $n - 1$. If role r_i is not executed because of this condition, the entry in *requests* will be evaluated again the next time a role is selected. If a resource leaves the loop (i.e., r_j is executed), the difference becomes less than $n - 1$ and r_i is eligible for execution again. Loop-induced deadlocks are thus avoided while the fairness guarantees are still upheld.

If $\forall(ag, r) \in \mathit{requests} : f'(r) = 0$ (i.e., no role can be executed at the moment without causing a deadlock), the agent stays idle until *requests* is updated by a new request and the role-selection algorithm is executed again.

7 Alternatives and Extensions

The basic mechanism described above works for many systems. However, there are situations that can not be dealt with as easily. This section describes these situations and how the mechanism can be adapted to accommodate more complex system structures and concurrent, distributed reconfiguration.

7.1 Dealing with Distributed Reconfiguration

Usually, the cycle detection algorithm would be executed whenever an agent receives a new configuration. However, in some systems with a distributed reconfiguration algorithm, the agents of a cycle might be reconfigured without the agent at the entry of the cycle even noticing. The process might break the cycle, change the number of agents in it or alter the cycle's structure in other, unforeseen ways. If the agent at the entry point upholds its deadlock avoidance strategy, there might thus be deadlocks occurring.

To counter this kind of situation, two strategies are conceivable. Firstly, after a reconfiguration occurred, information about this can be distributed to adjacent nodes and the algorithm can be run after receiving this information. Secondly, the agent at the entry point just runs the cycle detection periodically. As the algorithm requires only few messages and very little computational effort, this approach seems more efficient and also avoids problems with consistency and information dissemination in large-scale systems.

7.2 More Than One Entry Point into a Loop

A situation where a loop has several entry points is depicted in Fig. 2. It is most likely to occur when some agents of the loop do not apply a capability but just forward the resource and if there are two independent upstream resource processing lines that produce resources with the same state.

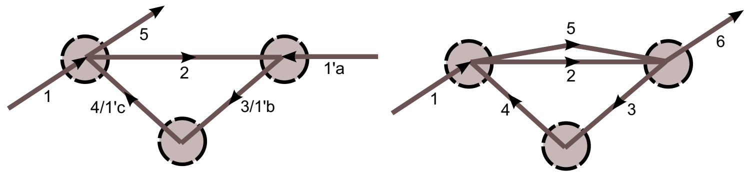


Fig. 2. Examples of loops of three agents and two entry points

It is possible that resources enter the loop at two different points thus causing a congestion. Such a situation can only be remedied if the two agents that let resources into the loop are cooperating to limit the number of resources processed in the loop. The requirement can be described as follows: if a loop consists of n agents and has $n - k$ entries where $n - 1 < k \geq 0$, the agents at the entries have to coordinate to ensure that a maximum of $n - 1$ resources are within the loop.

If an agent ag_i sent out a loop detection message and at a later point receives such a message originating from another agent ag_j, ag_i can suspect that a_j is an entry point to the same loop. After ag_i and ag_j determined they are part of a loop, they can exchange information about the loops they are part of and thus establish their relationship. During productive phases, the agent at the exit of the loop (say, ag_i) can award the other agent (ag_j) a quota of resources a_j may allow into the loop. After a resource has been allowed into the loop, ag_j will inform ag_i. When ag_j's quota is exhausted and ag_i has detected that resources exited the loop again, ag_j will be granted a new quota.

Although it is suspected that this simple coordination mechanism scales well for loops with more than two entry points, an investigation of such situations as well as a detailed description of the mechanism will be left as future work.

7.3 Bifurcations

An agent can have two roles with the same precondition but different postconditions, e.g. to balance the load for the successive agents. In such a case, the resource-flow graph bifurcates and the loop-detection message has to be sent to both outputs of both postconditions. As the originating agent ag_{org} now has to expect more than one message as a reply to its cycle detection message, it has to be informed about the bifurcation. ag_{org} then waits until the original message and all bifurcated message return. It is then able to establish the length of the minimal loop it is part of by the mechanism described above.

This theoretically rather simple procedure becomes tedious when implemented. The main problem is that it is not certain when a message will arrive at a_{org}. If the message that went through a cycle arrives before the message that indicates a bifurcation, the agent might have finished its cycle determination already. Special cases might have to be introduced to deal with such a situation.

7.4 Multiple Tasks

If resources with several tasks are processed in a system simultaneously, circular waits can now be induced by roles that are dealing with different tasks as shown in Fig. 3. However, such waits do not necessarily occur in every system with more than one task. If resources flow only in one direction through the system or if each agent is configured to handle only one task, the proposed approach is still applicable. In the general case, an extended loop detection mechanism would be able to detect circular waits induced by the processing of resources with different tasks and reduce this problem to the case of loops with several entry points. This, however, is left as future work.

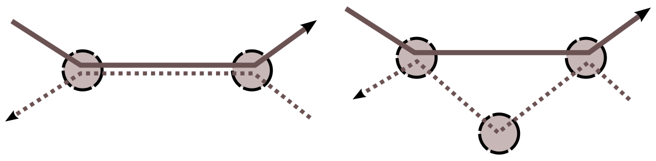


Fig. 3. Cycles induced by roles dealing with multiple tasks

8 Discussion and Conclusion

This paper described mechanisms for fair scheduling and deadlock avoidance based on local knowledge and minimal communication that are suitable for large scale dynamic system with a changing structure. In comparison to the approaches in literature this method is not limited by the computational time that is required to construct and simulate a global graph, works during runtime of the system and specifies the concrete application of the knowledge about potential deadlocks in the context of the scheduler. It also does not involve a massive amount of message passing and does not undermine the self-organizing and autonomous properties of a system and its entities respectively.

Although some details of extensions for more complex system structures are not yet fully elaborated, the approach already proves useful and has been implemented in the ODP Runtime Environment [24] on a case study of an adaptive production cell. The preliminary results are very encouraging and the open issues will be evaluated and solved with the help of this implementation.

References

1. Abate, A., Innocenzo, A.D., Pola, G., Di Benedetto, M.D., Sastry, S.: The Concept of Deadlock and Livelock in Hybrid Control Systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 628–632. Springer, Heidelberg (2007)
2. Ashfield, B., Deugo, D., Oppacher, F., White, T.: Distributed Deadlock Detection in Mobile Agent Systems. In: Hendtlass, T., Ali, M. (eds.) IEA/AIE 2002. LNCS (LNAI), vol. 2358, pp. 146–156. Springer, Heidelberg (2002)
3. Banaszak, Z.A., Krogh, B.H.: Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. IEEE Transactions on Robotics and Automation 6(6), 724–734 (1990)
4. Bandyopadhyay, A.K.: Fairness and conspiracy concepts in concurrent systems. SIGSOFT Softw. Eng. Notes 34(2), 1–8 (2009)
5. Barbosa, V.C.: Strategies for the prevention of communication deadlocks in distributed parallel programs. IEEE Transactions on Software Engineering 16(11), 1311–1316 (1990)
6. Belik, F.: An efficient deadlock avoidance technique. IEEE Transactions on Computers 39(7), 882–888 (1990)
7. Bracha, G., Toueg, S.: Distributed deadlock detection. Distributed Computing 2(3), 127–138 (1987)
8. Bustard, D., Hassan, S., McSherry, D., Walmsley, S.: GRAPHIC illustrations of autonomic computing concepts. Innovations in Systems and Software Engineering 3(1), 61–69 (2007)
9. Buth, B., Peleska, J., Shi, H.: Combining Methods for the Livelock Analysis of a Fault-Tolerant System. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, p. 124. Springer, Heidelberg (1998)
10. Coffman, E.G., Elphick, M.J.: System deadlocks. Computing Surveys (1971)
11. Costa, G., Stirling, C.: Weak and strong fairness in CCS. Information and Computation 73(3), 207–244 (1987)
12. Durvy, M., Dousse, O., Thiran, P.: Self-Organization Properties of CSMA/CA Systems and Their Consequences on Fairness. IEEE Transactions on Information Theory 55(3), 1 (2009)
13. Ezpeleta, J., Colom, J.M., Martinez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation 11(2), 173–184 (1995)
14. Fanti, M.P., Zhou, M.C.: Deadlock control methods in automated manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics, Part A 34(1), 5–22 (2004)
15. Goldman, B.: Deadlock detection in computer networks. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1977)
16. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A Specification and Construction Paradigm for Organic Computing Systems. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 233–242. IEEE Computer Society, Washington (2008)
17. Hsieh, F.S.: Fault-tolerant deadlock avoidance algorithm for assembly processes. IEEE Transactions on Systems, Man and Cybernetics, Part A 34(1), 65–79 (2004)
18. Huang, Y.S., Jeng, M.D., Xie, X., Chung, D.H.: Siphon-based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans 36(6), 1249 (2006)

19. Kim, K.H., Jeon, S.M., Ryu, K.R.: Deadlock prevention for automated guided vehicles in automated container terminals. OR Spectrum 28(4), 659–679 (2006)
20. Kobayashi, N.: Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) TCS 2000. LNCS, vol. 1872, pp. 365–389. Springer, Heidelberg (2000)
21. Kwong, Y.S.: On the absence of livelocks in parallel programs. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 172–190. Springer, Heidelberg (1979)
22. Lamport, L.: Fairness and hyperfairness. Distributed Computing 13(4), 239–245 (2000)
23. Li, Z.W., Zhou, M.C., Wu, N.Q.: A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 38(2), 173–188 (2008)
24. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A generic software framework for role-based Organic Computing systems. In: SEAMS 2009: ICSE 2009 Workshop Software Engineering for Adaptive and Self-Managing Systems (2009)
25. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A universal self-organization mechanism for role-based Organic Computing systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 17–31. Springer, Heidelberg (2009)
26. Park, D.: On the Semantics of Fair Parallelism. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980)
27. Sanchez, C., Sipma, H.B., Manna, Z., Gill, C.D.: Efficient distributed deadlock avoidance with liveness guarantees. In: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 12–20. ACM, New York (2006)
28. Sfinghal, M.: Deadlock detection in distributed systems. Computer 22(11), 37–48 (1989)
29. Wang, J., Zhang, X., Zhang, Y., Yang, H.: A Probabilistic Model for Parametric Fairness in Isabelle/HOL. In: Theorem Proving in Higher Order Logics: Emerging trends Proceedings, pp. 194–209 (2007)
30. Wang, Y., Kelly, T., Kudlur, M., Mahlke, S., Lafortune, S.: The application of supervisory control to deadlock avoidance in concurrent software. In: 9th International Workshop on Discrete Event Systems, WODES 2008, pp. 287–292 (2008)
31. Wang, Y.M., Merritt, M., Romanovsky, A.B.: Guaranteed deadlock recovery: Deadlock resolution with rollback propagation. In: Proc. Pacific Rim International Symposium on Fault-Tolerant Systems (1995)
32. Wu, N., Zhou, M.C.: Deadlock resolution in automated manufacturing systems with robots. IEEE Transactions on Automation Science and Engineering 4(3), 474–480 (2007)
33. Wu, N., Zhou, M.C., Li, Z.W.: Resource-oriented Petri net for deadlock avoidance in flexible assembly systems. IEEE Transactions on Systems, Man and Cybernetics, Part A 38(1), 56–69 (2008)
34. Zajac, J.: A deadlock handling method for automated manufacturing systems. CIRP Annals-Manufacturing Technology 53(1), 367–370 (2004)

Ad-Hoc Information Spread between Mobile Devices: A Case Study in Analytical Modeling of Controlled Self-organization in IT Systems

Kamil Kloch[1], Jan W. Kantelhardt[2], Paul Lukowicz[1], Patrick Wüchner[3], and Hermann de Meer[3]

[1] Embedded Systems Lab, University of Passau, Innstraße 43, D-94032 Passau Germany
[2] Institut für Physik, Martin-Luther-Universität Halle-Wittenberg, D-06099 Halle (Saale) Germany
[3] Computer Networks and Computer Communications, University of Passau, Innstraße 43, D-94032 Passau Germany

Abstract. We present an example of the use of analytical models to predict global properties of large-scale information technology systems from the parameters of simple local interactions. The example is intended as a first step towards using complex systems modeling methods to control self-organization in organic systems. It is motivated by a concrete application scenario of information distribution in emergency situations, but is relevant to other domains such as malware spread or social interactions. Specifically, we show how the spread of information through ad-hoc interactions between mobile devices depends on simple local interaction rules and parameters such as user mobility and physical interaction range. We show how three qualitatively different regimes of information 'infection rate' can be analytically derived and validate our model in extensive simulations.

1 Introduction

In this paper we present a specific example of the adaptation of an analytical complex system model to a self-organizing IT system. We show how logistic models from epidemiology can be applied to describe the spread of information or malware between mobile phones carried by people moving in a crowd. We derive the dependence of information penetration on crowd density, radio range, and motion speed. We show a 'phase transition'-like behavior: information is being spread with the speed either linearly increasing with the radio range, almost constant, or approximately exponentially. Another parameter-dependent emergent effect is the transition between local but dying 'information bubbles' and a continuous spread of information. The model is validated using extensive simulations showing good agreement with analytical predictions.
The value of the research presented in this paper is threefold:

First, on application level, it is part of a large, interdisciplinary EU project (SOCIONICAL, http://socionical.eu) to understand and exploit self-organization in large-scale systems in which intelligent mobile devices interact with each other and with humans. A specific application scenario within the project is the study how sensing, communication, and cooperation between mobile devices (e.g., sensor enabled smart phones) can be used to steer crowd behavior in emergency situations (e.g., to speed up evacuation or prevent panic). Ensuring that information can be effectively spread between the devices in an ad-hoc fashion (infrastructure, even including mobile phones, cannot be assumed to work in such situations, as the case of the London subway bombings has shown) is a key requirement. Another obvious application of our model is the spread of malware through peer-to-peer interactions between mobile devices.

Second, on theoretical level, it extends previous approaches on information and virus spread (see [1,3,8,9] for related work) by specifically considering the effects of mobility and radio range. We provide an analytical solution for the 'infection ratio' (which is the key parameter in system description) and validate our results in extensive simulations.

Third, in the area of organic computing, so far little attention has been given to the use of analytical complex system models as means of understanding and controlling the evolution of self-organizing IT systems (see [5] for some discussion). The system that we are investigating is easily mapped to the corresponding model, as information spread is obviously closely related to the spread of disease. Nonetheless, it is a good initial example of a design where global properties and phase transitions can be analytically derived from the rules for simple local interactions. We see it as a first step in the direction of using analytical models as means of achieving control in self-organized systems.

2 The Scenario

This work is part of a large project to investigate **large-scale, complex interactions** between intelligent mobile devices and humans in their environment. In essence, it aims to apply models and concepts from complexity science and organic computing to ambient intelligence environments. Ambient intelligence research to date has concentrated on how individual intelligent devices interact with a single user or small groups of users. The questions that we ask include: How can such devices influence collective behavior? What sort of global effect can emerge from the coupled interactions and feedbacks between many different users and devices? How can we predict such global effects (and thus design systems in the sense of controlled emergence) from local interaction rules embedded in each individual device?

A key issue underlying many of the research questions that we are addressing is the spread of information. Assuming that devices forward information to others **in their physical proximity** according to certain rules: How will information spread on a global scale? Can we, in the sense of controlled self-organization, use simple control parameters like user mobility and interaction range to ensure (or prevent) rapid spread?

The specific application that has motivated this work is disaster management. For example, as described in a London Assembly Meeting Report[1], during the London subway bombings mobile phone communication broke down and lack of information has been a key problem for many of the people stuck inside the stations and trains. Thus, the ability to propagate information through the crowd in an ad-hoc fashion would have been of significant use. In more advanced scenarios that we are examining, phones are able to sense crowd motion and environmental conditions and propagate them throughout the crowd to optimize evacuation ([4]).

Other applications include social interactions (e.g., 'flash mobs'), media distribution and the spread of malware.

2.1 Model Assumptions

For the simulation and modeling, some simplifying assumptions have been made. All mobile agents (representing, e.g., persons with cellular phones) are placed in a square board of a fixed size. To reduce the boundary effect of agents hitting the walls, the sides of the square are wrapped left-to-right and up-to-down (periodic boundary conditions), so that the actual simulation is being carried out on a torus.

At the beginning, the agents are placed uniformly on the board and only a fraction of agents has some interesting information or malware to be spread (status: **infected**), while all others are initially uninfected. The direction of motion of each agent is chosen randomly. Each agent moves along a straight line for a random number of steps. The length of the straight paths varies randomly from a few steps to half the size of the board. After that, the direction of the agent and the length of its straight path is randomly changed, and the procedure is repeated. The achieved motion is random and – at the same time – resembles the movement of agents in a crowd.

Each mobile agent carries a mobile device and is surrounded by the device's radio range. The radio range is modeled by a flat disc. If an uninfected device collides with the radio range of an infected device, it will also get infected immediately and change its status to infected. With a certain probability, in each simulation step, an agent will reset its infected mobile device, and thus, will get rid of the infection without getting immune.

The model parameters can be summarized as follows: (1) number of agents: N (since it is assumed that all agents carry only one device, N is equal to the number of devices), (2) initial infection ratio: $I \in [0 \ldots 1]$ (probability that a device will be infected initially), (3) system length: L unit lengths (overall area of $L \times L$ square unit lengths under consideration, kept constant in the presented scenario), (4) radio (interaction) range: R, (5) velocity of agents: v (in unit lengths per simulation step), and, optionally, (6) device reset probability: $P \in [0 \ldots 1]$ (probability that an agent will reset its infected mobile device within a simulation step).

[1] see `http://www.london.gov.uk/assembly/resilience/2005/77reviewdec01/minutes/transcript.pdf`

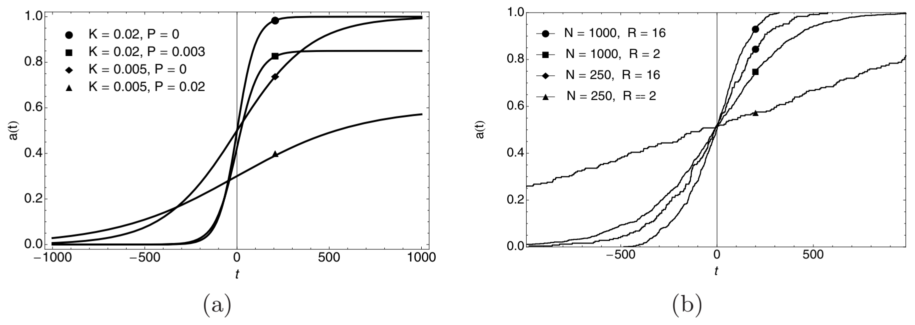


Fig. 1. (a): Plotting $a(t)$ against t for $T = 0$ and different values of K and P. (b): Relative number of infected devices $a(t)$ versus simulation steps t for various number N of agents and radio ranges R. The plot illustrates the definition of the time scale.

2.2 Logistic Model

A common model for the spread of diseases is the logistic model (see, e.g., [2]). This model also gets more and more attention from researchers in the field of computer science and engineering (see, e.g., [6] and [7]). In our scenario, the logistic equation takes the following form:

$$a'(t) = K\,a(t)\,(1 - a(t)) - P\,a(t). \tag{1}$$

Equation (1) can be motivated as follows. The maximum change in the infection ratio $a(t)$ $(0 \leqslant a(t) \leqslant 1)$ within an infinitesimal time interval around time t is given by the number of infected agents $a(t)$ at time t times the infection ratio K. Thus, K is the unimpeded infection rate that applies to the case when $P = 0$ and there is only one infected agent, and hence, each collision of this agent that occurs during the time interval leads to a new infection. The maximum change is delimited by the saturation effect caused by the finite number of agents to be infected, modeled by the factor $(1 - a(t))$ and by the number $P\,a(t)$ of infected agents that reset their device within the time interval.

By solving the differential equation (1) we get

$$a(t) = \frac{K - P}{K\left(1 + e^{-(K-P)(t-T)} - Re^{-(K-P)(t-T)}\right)},$$

where T is the location parameter that fixes the time when 50% of all agents is infected, i.e., $a(T) = 0.5$, in the case $P = 0$. In Fig. 1(a), $a(t)$ is plotted against time t for $T = 0$ and various values of K and P according to (1).

The slope of the logistic curves shown in Fig. 1(a) at $t = 0$ can be obtained from (1):

$$a'(0) = K\,a(0)\,(1 - a(0)) - P\,a(0) = K/4 - P/2. \tag{2}$$

Thus, knowing the value P, K can be obtained by investigating the slope of the curves at $t = 0$. The fix points are the expected values of $a(t)$ in steady state $t \to \infty$, i.e., $a'(t) = 0$. In our case, $\lim_{a\to\infty} a(t) = (K - P)/K$. This value can be

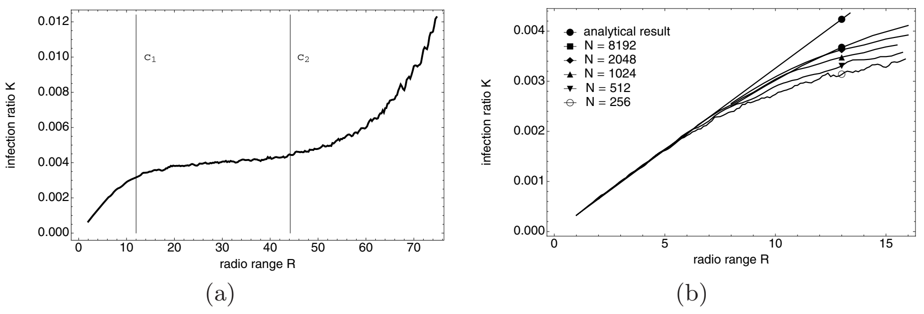


Fig. 2. (a): Simulation results: infection ratio K plotted versus radio range R for $N = 512$, $L = 2000$ and $2 \leqslant R \leqslant 75$. Vertical lines c_1 and c_2 mark the separations of the three regimes. (b): First regime – scaled simulation results for various N compared with the analytical value of K computed according to (4).

used to determine a suitable value for P depending on K. However, computing K by running simulations for each possible parameter set is unsatisfactory. In the following we deal with the primary concern of the paper, that is, achieving the closed-form equation for K that is based merely on scenario's parameters.

3 Model and Simulation

For the evaluation, the relative number $a(t)$ of infected devices at time t (simulation step) is of main interest. In Fig. 1(b), $a(t)$ is plotted versus t for four simulation runs with various values of N. The other model parameters are chosen as follows. The initial infection ratio I is chosen equal to $1/N$, which means one infected agent. The devices' radio range R is fixed to 10 unit lengths. The velocity v of the agents is one unit length per time unit. The time axis (horizontal axis) is chosen such that $a(0) = 0.5$. It can be seen that all curves show qualitatively the same behavior. However, the quantitative slope of the curves depends on the number of agents. Actually, as it will be shown later in more detail, the slope depends on the density of agents $\rho = N/L^2$ and on the system parameters that are kept constant in Fig. 1(b), like the agents' speed v and the radio range R.

In the following, we focus on the case $P = 0$ and $I = 1/N$, i.e., devices are not reset and exactly one device is initially infected. Figure 2(a) depicts the results of the simulation for 512 agents, radio ranges varying from 2 to 75, board size $L = 2000$, and velocity $v = 1$. The calculated values of the unimpeded infection ratio K, depicted on the vertical axis, are obtained from (2). For each value of R, the median of the slope $a'(0)$ at $t = 0$ is obtained from polynomial approximations for 50 simulation runs.

Simulation results shown in Fig. 2(a) reveal three distinct regimes with different dependences of the parameter K on R. At first, the infection ratio K grows linearly with radio range R. In the second regime the value of K remains almost constant. This is a very surprising result. Finally, in the third regime, the infection ratio starts to grow very rapidly with R. The three regimes are analyzed and described analytically in the following sections.

3.1 First Scaling Regime

In the first regime, we consider the case of very small radio ranges. Thus, agents travel long distances between coming into contact with each other and the infected and uninfected agents are typically spread uniformly across the board (see Fig. 3(a)). In order to derive the R dependence of K, we note that the unimpeded infection rate K specifies the expected number of new infections per time unit if no reset is involved. Hence, each collision of an infected agent with another agent has a high probability to cause a new infection, since the other agent is usually uninfected. By collision we mean that the distance between the two agents becomes as small as R. To calculate the expected number of collisions within one time step, we adapt the mean free path approach well-known from the kinetic theory of gases for a two-dimensional setting, agents of different sizes, and non-negligible radius of infected agents.

Consider an infected agent moving along a straight path with speed $\bar{v}$. During the next time unit the agent's radio range R will reach a previously untouched area of size $\bar{v} \cdot 2R$. All previously uninfected agents in this area will then be infected. The mean number of agents in the area is given by its size times the mean density of agents,

$$K = (\bar{v} \cdot 2R)\, N/L^2. \tag{3}$$

Note that $\bar{v}$ is not equal to the infected agent's velocity v in general, since both, infected and uninfected agents, are moving. Since in the presented scenario also the uninfected agents move randomly, $\bar{v}$ has to be calculated as the expected relative velocity of the uninfected agents with respect to the infected agent.

Therefore, we consider a reference system in which the infected agent is at rest. Without loss of generality, we can assume that it is moving to the right with velocity $\boldsymbol{v}_1 = (v, 0)$ (the two components of the vector indicating the speeds in x and y direction, respectively). In the considered moving reference system, the speed of the uninfected agents is thus given by $\boldsymbol{v}(\varphi) = \boldsymbol{v}_2 - \boldsymbol{v}_1 = v\,(\cos\varphi - 1, \sin\varphi)$ where $0 \leqslant \varphi < 2\pi$ denotes the (random) direction of motion of the uninfected agent in the original resting reference system. Consequently, the average absolute value of $\boldsymbol{v}$ is equal to

$$\begin{aligned}\bar{v} &= \frac{1}{2\pi}\int_0^{2\pi} |\boldsymbol{v}(\varphi)|\, d\varphi = \frac{1}{2\pi}\int_0^{2\pi} v\,\sqrt{(\cos\varphi - 1)^2 + \sin^2\varphi}\, d\varphi \\ &= \frac{v}{2\pi}\int_0^{2\pi} \sqrt{2 - 2\cos(\varphi)}\, d\varphi = \frac{4v}{\pi} \approx 1.2732 \cdot v.\end{aligned}$$

Hence, the unimpeded infection rate K can be calculated based on (3):

$$K = 8vR\rho\pi \approx 2.5465 \cdot vR\rho. \tag{4}$$

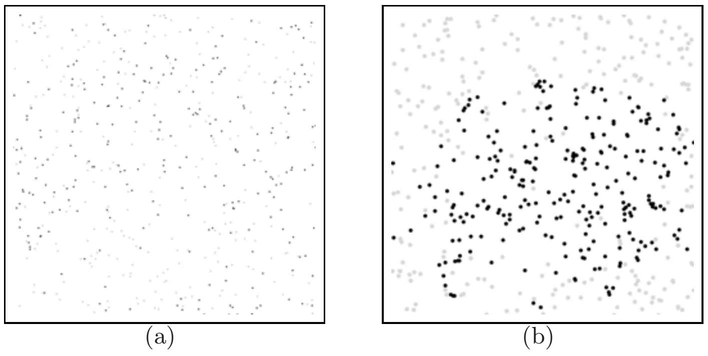


Fig. 3. Cross-over c_1 between the first and the second regime for 512 agents, infection ratio $a(0) = 0.5$, $R = 8$ (a) and $R = 20$ (b)

Fig. 2(b) shows a comparison between the simulation and the analytic result for several parameter settings. While keeping the size of the simulated area ($L = 2000$) constant, the number N of agents and the radio range R are varied. N varies from 256 (low density) to 8192 (highly crowded area). Each point in Fig. 2(b) represents the median of 50 simulation runs. For $N \neq 512$ the results on the horizontal axis have been scaled linearly with respect to ρ so that the analytical results derived from (4) are the same. The straight line depicts our analytical result based on the mean-value approach.

For $N = 512$ and radio ranges $R < 9$, the analytical model yields almost perfect results. Simulations confirm the linear growth of the infection ratio K with radio range R and agents' density ρ. Around $R = 10$ (for $N = 512$), the value of K ceases to grow linearly with R and we observe the crossover to the second regime with almost constant K.

The phase transition at c_1 (between the first and the second regime) can be understood by taking a look at the results of two experiments for 512 agents, with $R = 8$ and $R = 20$, respectively. Figure 3 shows the distribution of the agents at the time when half of the population is infected. This is the moment which determines the value of K (recall (2)). For $R = 8$, the infected agents are scattered across the entire board. For $R = 20$, they tend to conglomerate. Most of the infected agents are then surrounded by others which are infected as well, and thus, the number of agents to be infected in the next time stepts is much smaller.

3.2 Second Scaling Regime

With increasing radio range of the devices, the infected agents cease to be uniformly distributed across the entire board and our analytic description for the first regime fails. Instead, the agents tend to form a disk-like pattern (see Fig. 3). This behavior becomes more pronounced for larger values of R, for which the infected agents form an almost perfect disk growing linearly with time.

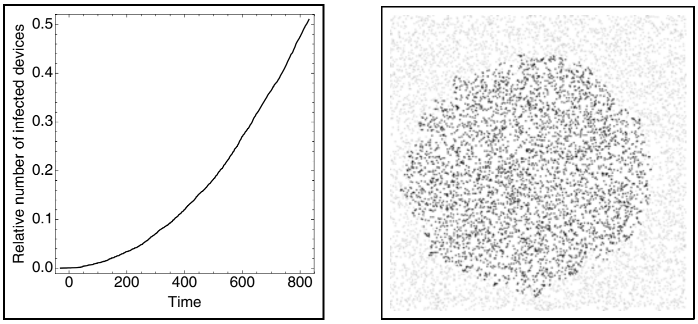


Fig. 4. Second regime: simulation run confirming the analytical model

We denote by $a(t)$ the relative number of the infected devices at iteration step t. Now, recall from Sec. 2.2 that parameter K is determined by the slope of $a(t)$ at the point $t = t_0$ such that $a(t_0) = 0.5$. We combine this fact with the knowledge about the distribution of the infected agents to develop the following analytical model. Firstly, we assume that at iteration step t_0 for which $a(t_0) = 0.5$ the infected agents form a disk D of radius r_D. Since all agents are distributed uniformly across the entire board, the area of D is half of the board's size, i.e.,

$$\pi r_D^2 = L^2/2, \tag{5}$$

which yields $r_D \approx 0.4\,L$. The second assumption is a crucial one: the radius of the disk formed by the infected agents at time t for $t \leqslant t_0$ is approximately equal to $v \cdot t$, i.e., the distance travelled by a single agent in t steps. In particular, this implies that

$$r_D = t_0\, v \quad \text{and} \quad a(t) = c \cdot t^2, \text{ for some constant } c \text{ and } t \leqslant t_0. \tag{6}$$

Both assumptions we have made are supported by the simulation results and follow from the special property shared by the radio ranges considered in the second regime: they are big enough to guarantee the compact, disk-like form of the infected agents, yet small enough not to cause chain infections and thus retaining agents' velocity as the prevailing factor of virus spread.

From (2), (5), (6) and the fact that $a(t_0) = 0.5$ we now get

$$K = 4\,a'(t_0) = 4 \cdot 2\,c\,t_0 = 8\,\frac{0.5}{t_0^2}\,t_0 = \frac{4}{t_0} = \frac{4\,v}{r_D} = \frac{4\sqrt{2\pi}v}{L} \approx \frac{10v}{L}. \tag{7}$$

The simulation run for $L = 2000$, $N = 8192$, $R = 10$, and $v = 1$ shown in Fig. 4 confirms this analytical model. Infection ratio grows quadratically with time, reaches 50% for $t_0 = 840$ and the infected agents form a disk of radius ≈ 800.

The position of the first crossover c_1 (see Fig. 2(a)) separating regimes one and two can be approximated by equating Eqs. (4) and (7). This yields

$$K = \frac{8vR_{c_1}N}{\pi L^2} = \frac{4\sqrt{2\pi}v}{L} \qquad \Longrightarrow \qquad R_{c_1} = \sqrt{\frac{\pi^3}{2}}\,\frac{L}{N} \approx 3.937\,\frac{L}{N}. \tag{8}$$

We see that the position of c_1 depends linearly on L/N.

3.3 Third Scaling Regime

The crossover c_2 from the second (nearly constant) regime to the third regime (see Fig. 2(a)) occurs when the mean distance between the agents becomes comparable with R. Then, immediate chain infections can occur, if a third agent happens to be within the radio range of the second (just infected) agent, even if it is not sufficiently close to the first (originally infected) agent.

The value of the crossover length R_{c_2}, i.e., the position of crossover c_2, can be calculated analytically as follows. Firstly, we have to determine the mean distance between nearest neighbor agents. Placing the agents randomly on the board can be interpreted as a Poisson process. Recall that a Poisson distribution is typically used to describe the probability of the occurrence of uncorrelated events over time, space, or length. Our use of the Poisson distribution is justified by the fact that the random locations of the agents are independent. The number n of agents within a given area A is described by the probability function

$$F(n) = \frac{(\rho A)^n}{n!} e^{-\rho A}, \text{ for } n = 0, 1, \dots .$$

We fix the position of a single agent p. Since the agents are independent, the probability that there is no other agent within distance r from p is given by $F(0) = \exp[-\rho\,\pi r^2]$. If x is the distance of the nearest agent from p, the probability that $x \leqslant r$ is equal to $P(x \leqslant r) = 1 - P(x > r) = 1 - F(0) = 1 - \exp[-\rho\,\pi r^2]$. Consequently, the density function of the distance from p to its nearest neighbor is given by

$$f(r) = \frac{dP(x \leqslant r)}{dr} = \frac{d}{dr}\left(1 - e^{-\rho\,\pi r^2}\right) = 2\pi\,\rho\,r\,e^{-\rho\,\pi r^2},$$

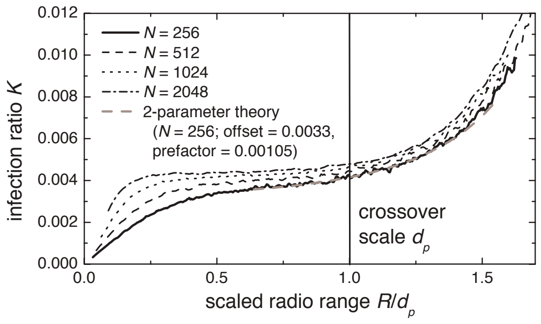
Finally, the mean nearest neighbor distance d_p is equal to

$$d_p = \int_0^{+\infty} f(r)\,r\;dr = \frac{1}{2\sqrt{\rho}} = \frac{L}{2\sqrt{N}} = R_{c_2}, \tag{9}$$

with L and N denoting system size and the number of agents, respectively.

The characteristic length scale d_p determines analytically the crossover c_2 from regime two to regime three, where chain reactions start to occur. It is therefore denoted by R_{c_2} in (9). We note that R_{c_2} scales with $L/\sqrt{N}$, while R_{c_1} scales with L/N, see (8). Figure 5 shows the results of our simulations, i.e. the infection ratio K, versus the scaled radio range R/d_p for four numbers N of agents and systems of length $L = 2000$. The scaling behavior is clearly confirmed. The curves in the third regime, i.e., for $R/d_p > 1$, differ just by a constant offset which is due to different absolute values occurring already in the second regime (for $R/d_p < 1$).

In general, the value of K in the second regime given by (7) for $L = 2000$ and $v = 1$ is $K = 0.05$. Figure 5 shows that this threshold is not fully reached in our simulations for smaller densities $\rho = N/L^2$. In particular, K does not exceed 0.03 before the onset of the third regime at $R = R_{c_2}$ if $N = 256$. The reason is



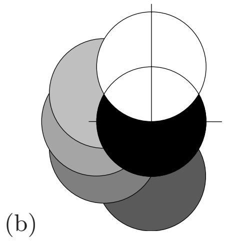


Fig. 5. (a) Scaling plot for the infection ratio K in the third regime. The K values for different N (see legend) and $L = 2000$ are plotted versus the scaled radio range R/d_p with d_p taken from (9). The numerical curves are parallel with a small offest in the third regime ($R/d_p > 1$). The dashed grey line shows the analytic theory. Two fitting parameters are needed: a prefactor and an offset; both will be different for $N > 256$. (b) Illustration of the geometical constraint determining chain infections for simultaneous infection of three agents.

the closeness of both crossovers for small ρ (and N). While the second regime is rather broad for $N = 2048$, it nearly vanishes for $N = 256$, where

$$\frac{R_{c_1}}{R_{c_2}} = \left(\sqrt{\frac{\pi^3}{2}}\frac{L}{N}\right) / \left(\frac{L}{2\sqrt{N}}\right) = \sqrt{2\pi^3/N} \approx 0.492$$

compared with $R_{c_1}/R_{c_2} \approx 0.174$ for $N = 2048$. Therefore, $K = 0.05$ is not fully reached for the lower values of N, and the curve in the third regime is consequently shifted downwards by a small amount as seen in Fig. 5. This, however, does not devaliate our analytical descriptions of both crossovers nor the unified scaling behavior seen in the third regime.

To derive the form of the scaling curve for $R/d_p > 1$ in Fig. 5, we have to consider the geometric constraints for immediate chain infections. The probability of a chain infection is not related with the motion of the agents. Therefore, one need not consider trajectories, and the analysis is mainly geometrical. A newly infected (second) agent is always located at distance R from the infecting agent, since it would have been infected earlier otherwise. Its radio range, i.e., the area in which a third agent could be infected, thus overlaps with the radio range area A_1 of the first agent (where no additional third agent could be infected). This extended radio range area A_2 is thus not a circle. Nevertheless, the corresponding area can be calculated analytically. Figure 5(b) illustrates A_1 (white) and A_2 (black). Without loss of generality, we assume that A_2 is to the right of A_1. A_2 is a half circle ($\pi/2\, R^2$) plus twice the (nearly triangular) part with height $R/2$ and length R; its exact area is $2.18918\, R^2$. The additional infection probability (first order term) is thus $F_1 = 2.18918\, \rho\, R^2$.

However, the chain reaction can go on, since there could be a fourth agent in the radio range of the third agent. Since the third agent can be anywhere in A_2

(black), different areas A_3 for the forth agent must be considered depending on the actual position of the third agent. Figure 5(b) illustrates several possibilities, for which we have calculated the area analytically (grey). All of these circles have the center on the edge of the black area. However, positions closer to the center of the black area are also possible. To take them into account in a reasonable averaging procedure, one must note that the range of possible centers for A_3 increases by r with the distance r from the center of A_2. A corresponding well-justified (although not analytically exact) averaging procedure yields $1.34331\,R^2$ for the average area A_3. The probability of the second order spontaneous infection is thus proportional to the product of first order $2.18918\,\rho\,R^2$ times $1.34331\,\rho\,R^2$, yielding $F_2 = 2.94076\,\rho^2\,R^4$ for the second term. Extending this rule further, we have approximated the third term by $F_3 = 3.63601\,\rho^3\,R^6$, etc.

The analytical curve included in Fig. 5 is furthermore based on approximations of the forth and fifth terms F_4 and F_5, which we also approximated. Clearly, a fast convergence of this series $F = F_1 + F_2 + F_3 + F_4 + F_5 + \ldots$ requires that $\rho R^2 \ll 1$. This is violated for large R or ρ of course. The calculation thus becomes inaccurate for large radio ranges R and large densities $\rho = N/L^2$. In particular, not taking into account terms for very large order (and stopping with F_5) will lead to K values which are too small for large R.

Figure 5 shows that data from the simulation of $N = 256$ can be fitted very well. We have to employ two fit parameters: the constant level in the second regime and a prefactor relating F to K,

$$K = \text{offset} + \text{prefactor}\,F. \tag{10}$$

The fit is very good except for very large R as expected. We note that the value of the offset parameter in (10) is close to the constant $K = 0.05$ in the second regime given by the analytic (7). It is somewhat lower for small N due to the shortness of the second regime as explained above. Therefore, the offset is not a real fitting parameter. In addition, the prefactor in (10) is related to the ratio of the mean velocity $v \approx 1.2732$ (see calculation for first regime) over the board length $L = 2000$, and thus also not a real fitting parameter. Again, the numerical values obtained for the fit in Fig. 5 deviate due to the specificity of the second regime. The deviations are less for larger values of N, where the same analytical theory with two fitting parameters applies although both parameters in (10) are slightly different.

4 Conclusion

We have shown that depending on the physical interaction range and the average speed of physical motion there are three distinct regimes for the information distribution rate. The proposed analytical model comprises both the three regimes and the two cross-overs between them:

(i) Phase of linear growth (see Sec. 3.1). For small radio ranges R and low density ρ of agents the infection ratio K is proportional to $R\rho$. The position of the first cross-over depends linearly on L/N.

(ii) Phase of an almost constant value of K, more pronounced for larger densities (see Sec. 3.2). Radio ranges are big enough to prohibit a homogenous distribution of infected agents, yet small enough not to cause a chain infection. In our analytical model, we prove that the value of K indeed does not depend on R. The second cross-over is determined by the mean distance between the agents and thus scales with $L/\sqrt{N}$.

(iii) Phase of a very rapid growth of K, when radio ranges are big enough to trigger chain infections (see Sec. 3.3). The probability of a chain infection is not related with the motion of the agents and indeed, in the obtained analytical model the value of K depends only on R and ρ.

Our models have made some simplifying assumptions (e.g., torus topology, random motion, and homogeneous distribution) so that applicability to real life emergency situations needs to be further investigated in more advanced simulations. However, the results are a good indication of the type of behavior that can be expected to occur. Most of all, the work is an initial example of how analytical complex systems models can be leveraged to facilitate control and predictability in large-scale organic systems.

References

1. Arai, T., Yoshida, E., Ota, J.: Information diffusion by local communication of multiple mobilerobots. In: Systems, Man and Cybernetics, 1993. Systems Engineering in the Service of Humans, Conference Proceedings, pp. 535–540 (1993)
2. Boccara, N.: Modeling Complex Systems. Springer, Heidelberg (2004)
3. Chen, Z., Gao, L., Kwiat, K.: Modeling the spread of active worms. In: INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 3. IEEE, Los Alamitos (2003)
4. Ferscha, A., Zia, K.: Lifebelt: Silent directional guidance for crowd evacuation. In: IEEE ISWC 2009 (2009)
5. Mainzer, K.: Organic Computing and Complex Dynamical Systems Conceptual Foundations and Interdisciplinary Perspectives. Organic Computing, 105 (2008)
6. Nicol, D.M., Liljenstam, M.: Models of active worm defenses. In: Proc. of Measurement, Modeling and Analysis of the Internet Workshop, IMA 2004 (2004)
7. Staniford, S., Paxson, V., Weaver, N.: How to own the internet in your spare time. In: Proceedings of the 11th USENIX Security Symposium, Berkeley, CA, USA, pp. 149–167. USENIX Association (2002)
8. Zou, C.C., Gong, W., Towsley, D.: Code red worm propagation modeling and analysis. In: CCS 2002: Proceedings of the 9th ACM conference on Computer and communications security, pp. 138–147. ACM, New York (2002)
9. Zou, C.C., Gong, W., Towsley, D.: Worm propagation modeling and analysis under dynamic quarantine defense. In: WORM 2003: Proceedings of the 2003 ACM workshop on Rapid malcode, pp. 51–60. ACM, New York (2003)

MLP-Aware Instruction Queue Resizing: The Key to Power-Efficient Performance

Pavlos Petoumenos[1], Georgia Psychou[1], Stefanos Kaxiras[1], Juan Manuel Cebrian Gonzalez[2], and Juan Luis Aragon[2]

[1] Department of Electrical and Computer Engineering, University of Patras, Greece
[2] Computer Engineering Department, University of Murcia, Spain

Abstract. Several techniques aiming to improve power-efficiency (measured as EDP) in out-of-order cores trade energy with performance. Prime examples are the techniques to resize the instruction queue (IQ). While most of them produce good results, they fail to take into account that changing the timing of memory accesses can have significant consequences on the memory-level parallelism (MLP) of the application and thus incur disproportional performance degradation. We propose a novel mechanism that deals with this realization by collecting fine-grain information about the maximum IQ resizing that does not affect the MLP of the program. This information is used to override the resizing enforced by feedback mechanisms when this resizing might reduce MLP. We compare our technique to a previously proposed non-MLP-aware management technique and our results show a significant increase in EDP savings for most benchmarks of the SPEC2000 suite.

1 Introduction

Power efficiency in high-performance cores received considerable attention in recent years. A significant body of work targets energy reduction in processor structures, striving at the same time to preserve the processor's performance to the extend possible. In this work, we revisit a class of microarchitectural techniques that resize the Instruction Queue (IQ) to reduce its energy consumption. The IQ is one of the most energy-hungry structures because of its size, operation (fully-associative matches), and access frequency.

Three proposals by Buyuktosunoglu et al. [1], Folegnani and González [2], and Kucuk et al. [3] exemplify this approach: the main idea is to reduce the energy by resizing the IQ downwards to adjust to the needs of the program using at the same time a feedback loop to limit the damage to performance. The IQ can be physically partitioned into segments that can be completely turned off [1][3] or logically partitioned [2]. While physical partitioning and segment deactivation can be more effective in energy savings, the more sophisticated resizing policy of Folegnani and González minimizes performance degradation. We consider the combination of the physical partitioning of [1] and [3] and the "ILP-contribution" policy of [2] as the basis for our comparisons.

Studying these approaches in more detail we discovered that, in some cases, small changes in the IQ size bring about significant degradation in performance. New found understanding of the relation of the size of the instruction queue to performance, by the work of Chou et al. [7], Karkhanis and Smith [6], Eyerman and Eeckhout [4], Qureshi et al. [8], points to the main culprit for this: Memory-Level Parallelism (MLP) [5]. *In the presence of misses and MLP the single most important factor that affects performance in relation to the IQ size is whether MLP is preserved or harmed.*

While this new understanding seems, in retrospect, obvious and easy to integrate in previous proposals, in fact it requires a new approach in managing the IQ. Specifically, while prior feedback-loop proposals based their decisions on a coarse sampling period measured in thousands of cycles or instructions, an *MLP-aware* technique requires a much more fine-grain approach where decisions must be taken at instruction intervals whose size is comparable to the size of the IQ. The reason for this is that MLP itself exists if misses appear within the instruction window of the processor and must be handled at that resolution. More accurately, MLP exists if the distance between misses is less than the size of the reorder buffer (ROB) [6]. In our case, we use a combined IQ and ROB in the form of a Register Update Unit [9], thus the ROB size defaults to the size of the IQ. In the rest of the paper we discuss MLP with respect to the size of the IQ.

Contributions of this paper:

- We expose MLP (when it exists) as the main factor that affects performance in IQ resizing techniques.
- We propose a methodology to integrate MLP-awareness in IQ resizing techniques by measuring and predicting MLP in *fine-grain* segments of the dynamic instruction stream.
- We propose an example practical implementation of this approach and show that it consistently outperforms in *total* EDP (for the whole processor) an optimized non-MLP-aware technique as well as improving EDP across the board compared to base case of an unmanaged IQ.

Structure of this paper— Section 2 presents related work, while Section 3 motivates the need for IQ resizing using MLP. In Section 4 we present our proposal for MLP-awareness in the IQ resizing and in Section 5 we delve into some details. Section 6 offers our evaluation and Section 7 concludes the paper.

2 Related Work

IQ Resizing Techniques— The instruction window (including possibly a separate reorder buffer, an instruction queue and the load/store queue) has been prime target for energy optimizations. The reason is twofold: first they are greedy consumers of the processor power budget; second, typically these structures are sized to support peak execution in a wide out-of-order processor. Although there are many proposals for IQ resizing, we list here the three main proposals that are relevant to our work.

The proposals by Buyuktosunoglu et al [1] and Kucuk et al [3], resize the IQ by physically partitioning it in segments and disabling and enabling each segment[1] as needed. Both techniques use a feedback loop to control the size of the IQ. The Buyuktosunoglu et al. technique [1] is based on the number of *active* entries (i.e., ready-to-issue entries) per segment to decide whether or not a segment deserves to remain active, while Kucuk et al. [3] argue that the *occupancy* of the IQ (in valid instructions) rather than active entries is a better indication of its required size. In both techniques the IPC is measured periodically; a sudden drop in IPC signals the need for IQ upsizing. However in both approaches there is no other mechanism to revert back to the full IQ size.

The Folegnani and González approach [2] is distinguished by being a logical resizing of the IQ (limiting the range of the entries that can be allocated) not a physical partitioning as the other two. In addition, the feedback mechanism is based on how much the youngest instructions in the IQ contribute to the actual ILP. If they do not contribute much, this means that the size of the IQ can be reduced further. However, there is no way to adapt the IQ to larger sizes except periodically revering back to full size. Nevertheless, the Folegnani and González resizing policy is very good at adjusting the IQ size so as to not harm the ILP in the program.

Modeling of MLP— Karkhanis and Smith presented a first-order model of a superscalar core [6] that deepened our understanding of MLP. This work exposed the importance of MLP in shaping the performance of out-of-order execution. Karkhanis and Smith show that in absence of any upsetting events such as branch mispredictions and L2 misses the number of instructions that will issue (on average) is a property of the program and is expressed by the so-called *IW characteristic* of the program.

The presence, however, of upsetting events, such as L2 misses, decreases the IPC from the ideal point of the IW characteristic depending on the apparent cost of the L2 miss. This is because, a L2 miss drains the pipeline, and eventually stalls it when the instruction that misses blocks the retirement of other instructions [6]. MLP, in this case, spreads the cost of accessing main memory over all the instructions that miss in parallel, making their apparent cost appear much less.

Existing IQ resizing mechanisms focus mainly on the variations of the ILP (effectively moving along the IW characteristic) ignoring what happens during misses. Our work, focuses instead on the latter.

MLP-Aware techniques— Qureshi et al. exploited MLP to optimize cache replacement [8]. MLP in their case is associated with data (cache lines) and the MLP prediction is done in the cache. Data that do not exhibit MLP are given preference in staying in the cache over data that exhibit MLP. Eyerman and Eeckhout exploit MLP to optimize fetch policies for Simultaneous Multi-Threaded (SMT) processors [4]. In their case, MLP is associated with instructions that miss in the cache. Our MLP prediction mechanism is similar to the one proposed by Eyerman and Eeckhout, but because we use it not to regulate the fetch stage, but to manage the entire IQ, we associate MLP information to larger segments of code.

[1] Sequentially from the end of the IQ in [1] or independently of position in [3].

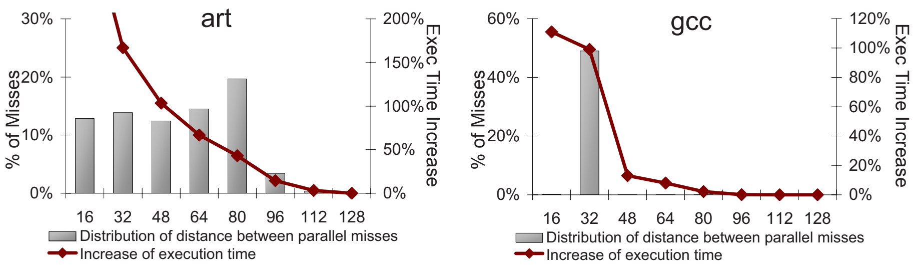


Fig. 1. Comparison of the distribution of distances between parallel L2 misses and performance degradation due to IQ resizing for gcc and art

3 MLP and IQ Resizing

In this section, we motivate the basic premise of this paper, i.e., that IQ resizing techniques must be aware of the MLP in the program to avoid excessive performance degradation. Figure 1 shows the distribution of the distances between parallel misses in two SPEC2000 programs: art and gcc. We assume here an IQ of 128 entries. The distance is measured in the number of intervening instructions between instructions that miss in parallel. Figure 1 also plots for each distance the increase in execution time if the IQ size is decreased below that distance.

Each time we resize the IQ, we eliminate the MLP with distance greater than the size of the IQ. In art, MLP is distributed over a range of distances, so its execution time is proportionally affected with the decrease of the IQ because we immediately eliminate some MLP. The more MLP is eliminated the harder performance is hit. In contrast, most MLP in gcc is clustered around distance 32. Below this point, we experience a dramatic increase in execution time (100%). For the intermediate IQ sizes, (between the maximum size and 32), execution time increases slowly due to loss of ILP. These examples demonstrate how sensitive performance is with respect to MLP and indicate that an efficient IQ management scheme must focus primarily on MLP rather than ILP.

Another important characteristic of MLP that necessitates a fine-grain approach to IQ management is that the distance among parallel misses changes very frequently. Figure 2 shows a window of 20K instructions from the execution of twolf. At each point the maximum observed distance among parallel misses is shown. Ideally, at

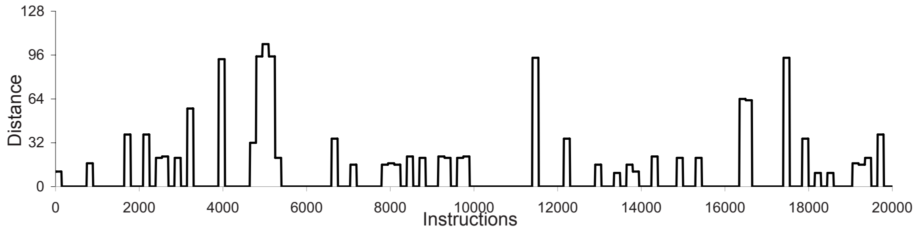


Fig. 2. Maximum Distances between parallel misses of twolf

each point in time, the IQ size should fit all such distances while at the same time be as small as possible. A blanket IQ size for the whole window, based on some estimation of the average distance between parallel misses, is simply not good enough since it would eliminate all MLP of distance larger than the average.

4 Managing the IQ with MLP

Our approach is to quantify MLP opportunities and relate this information back to the instruction stream via a prediction structure. Upon seeing the same instructions again, MLP information stored in the predictor guides our decisions for IQ resizing.

4.1 Quantifying MLP: *MLP-Distance*

Our first concern is to quantify MLP opportunities in a way that is useful to IQ resizing. Two memory instructions are able to overlap their L2 misses, if there are no dependencies between them and the number of instructions dispatched between them is less than the size of the IQ. This number of instructions, called *MLP-distance* in [4], is also the basic metric for our management scheme.

A straightforward way to measure the MLP-distance among instructions is to check the LSQ every time a miss is serviced and find the youngest instruction which is still waiting for its data from the L2. This technique does not "fit" in our case, since it can only identify overlapping misses *for the current size of the IQ*. To overcome this problem we need to check for misses that could potentially overlap over a number of instructions, as many as the *maximum* number of instructions that can fit in the unmanaged IQ. Always keeping information for as many instructions as the maximum IQ size, partially defeats the purpose of resizing the IQ. Thus, instead of keeping information for each individual instruction, we keep aggregate information for *instruction segments*, groups of sequentially dispatched instructions (*which coincide with the segments that make up a physically partitioned IQ*).

This MLP information is kept in a small cyclic buffer —which we call *MLP distance buffer* or *MDB*— with as many entries as the maximum number of IQ segments (Fig.3). MDB is not affected by IQ resizing. A new entry is allocated for each segment, but in contrast to the real IQ entries, MDB entries are evicted only when it fills. This means that MDB "retires" a segment only when newly dispatched instructions are farther away than the size of the non-managed IQ, and thus could not possibly execute in parallel with the retiring segment. Our approach to measure MLP distance, is similar to [4] but based on segments for increased power efficiency. Each time an instruction causes an L2 miss, the corresponding MDB segment is marked as also having caused a miss. Upon eviction of an entry, the MDB is searched for the other entries which have caused L2 misses. If there are such entries, this means that there could be possible MLP among them. We update each entry's *MLP-distance field* with the distance —measured in segments— from the youngest entry with a miss, if this distance is longer than the previously recorded value. MDB is infrequently accessed and it is only processed whenever segments which caused L2 misses are retired

The MLP-distance is not an entirely accurate estimation of actual MLP. To reside at the same time in the IQ is not the only requirement for two instructions to overlap their

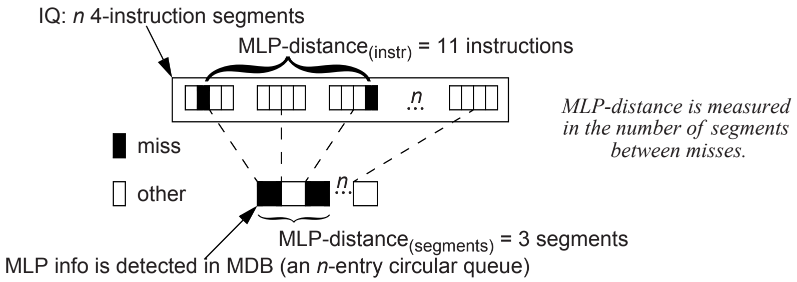

IQ: n 4-instruction segments
MLP-distance(instr) = 11 instructions
n
miss
other
n
MLP-distance(segments) = 3 segments
MLP info is detected in MDB (an n-entry circular queue)
MLP-distance is measured in the number of segments between misses.

Fig. 3. MLP Distance Buffer (MDB)

misses i.e. possible dependencies between the instructions may cause their misses to be isolated. In any case, the "actual" MLP-distance will be less than or equal to the value produced by our approach. This in turn means we might miss some opportunities for downward IQ resizing but we will not incur performance degradation due to loss of MLP. Our experiments showed that for most benchmarks falsely assuming parallel misses causes few overestimations of the MLP-distance. Considering the simplicity of our mechanism, these results indicate a satisfactory level of performance.

Measuring the MLP-distance allows us to control the IQ size for the relevant part of the code so as to not hurt MLP. We need, however, to relate this information back to the instruction stream so the next time we see the same part of the code we can react and correctly set the size of the IQ.

4.2 Associating MLP-Distance with Code

For the purpose of managing the IQ in a MLP-aware fashion, we dynamically divide program execution into fragments and associate MLP-distance information with these fragments. Execution fragments should be comparable in size to the size of the IQ. Much shorter fragments would be sub-optimal since the information they carry will be used to manage the whole IQ, which contains multiple such fragments. This could lead to very frequent and —many times— conflicting resizing decisions. Longer fragments, such as program phases, also fail to provide us with the fine-grain information we need to quickly react to fast-changing MLP-distances in the instruction stream.

To achieve the desired balance in the size of the fragments we use the notion of *superpaths.* A superpath is nothing more than an aggregation of sequentially executed basic blocks and loosely corresponds to a trace (in a trace cache) [13] or a hotspot [12]. The MLP-distance of a superpath is assigned by the MDB: when all of the MDB entries belonging to a superpath are "retired," the longest MLP-distance stored in these entries is selected to update the MLP-distance of the superpath. Note that the instructions which establish this MLP-distance do not have to belong to the same superpath. An MLP-distance that straddles a number of superpaths affects all of them (if it is the maximum observed MLP-distance in each of them).

The next time the same superpath is observed in dispatch, its stored information is retrieved to manage the IQ. A more detailed discussion about superpaths and how we actually implemented the aforementioned mechanisms can be found in Section 5.

4.3 Resizing Policy

When we start tracking a superpath in dispatch, we check whether we have stored information about its behavior, including its MLP-distance information. If there is such information then we have an indication about the minimum IQ size which will not affect the MLP of this superpath. In many cases, however, there is no MLP-distance information available or the MLP-distance does not restrict IQ downsizing. The question is how much can we downsize the IQ is such cases? As explained in Section 3, an efficient IQ resizing scheme has to find the IQ size that minimizes energy consumption without hurting *performance*. This means that besides not hurting MLP we must also protect ILP.

This gap can be filled by any of the existing IQ resizing techniques. For example, the ILP-feedback approach of [2] can provide a target IQ size that does not hurt ILP while the MLP-aware approach judges whether this size hurts MLP and if so it overrides the decision. For the rest of this paper the ILP-feedback information will be provided by the decision making mechanism in Folegnani and González [2]. This mechanism examines the participation of the youngest segment of the IQ to the IPC within a specific number of cycles. If the contribution of the youngest part is not important, namely the number of instructions that issue from this segment is below a *threshold*, the IQ size is decreased. In our case, we deactivate the youngest segment when it empties.

The main idea in the Folegnani and González work is that if a segment contributes very little to ILP, deactivating it would not harm performance. However, this holds only for ILP —not for MLP. In other words, even if the contribution of a segment in issued instructions is very small, it can still have a significant impact on performance, if *any* of its issued instructions is involved in MLP. It is exactly for such cases where MLP-awareness makes all the difference. Further, in the Folegnani and González work, once the IQ is downsized, there is no way to detect whether the situation changes —all we can see is the contribution to ILP of the *active* portion of the IQ. Thus, periodically, the IQ is brought back to its full size, and the downsizing process starts again. In our case, the existence of MLP automatically upsizes the IQ, to a size that does not harm MLP.

5 Practical Implementation

5.1 IQ Segmentation

To allow dynamic variation of the IQ size, we divide the IQ in a number of independent parts referred as segments. For example, we use an 128-entry IQ partitioned into eight, sixteen-entry segments. Bitline segmentation is used to implement the resizing of the structure [10]. The structure of the IQ follows the one in [2]. The IQ is a circular FIFO queue, with every new instruction inserted at the tail; retiring instructions are removed from the head. The difference in our case, is that individual segments can be deactivated. A segment is deactivated if instructions from the youngest segment contribute less than *threshold* instructions in a *quantum* of time (1000 cycles) and a segment is reactivated every 5 *quanta*. This inevitably leads to constraints that have to be met during the resizing process, similarly to those faced by Ponomarev et al. [11]:

downsizing of the IQ is only permitted if there are no instructions left to commit in the segment being removed and upsizing is constrained to activate segments that come after all the instructions currently residing in the IQ.

5.2 Superpaths

Our basic IQ management unit, the superpath, is characterized by its size and its first instruction. Sequential basic blocks are organized into superpaths at the dispatch stage and they contain at least as many instructions as the IQ size. Superpath creation ends when we encounter a basic block which is at the head of a previously created superpath, in order to reduce both the number and *the overlap* of superpaths. For each newly created superpath we allocate an entry in a small hardware cache which keeps information about the superpath, as well as information about its MLP-distance. After performing an exploration of the design space, we chose a 4-way, 16-set configuration (indexed by the lower-order bits of the start address of the superpath).

We store 28 bits per superpath entry: 20 bits of the starting address (lowest-order bits above the indexing bits), the MLP-distance prediction (4 bits, again quantized in multiples of 16) and its confidence counter (3 bits) and a valid bit. For our 4x16 cache, our storage requirements add up to 1792 bits. According to CACTI [14] this structure contributes 9.5 mW to the total power consumption of the processor, which is a reasonable power overhead compared to the power consumption of the IQ (8.9W for the configuration described in Section 6).

5.3 MLP-Distance Prediction

When all instructions of superpath commit, the stored superpath information is updated with the MLP-distance information of this particular execution. Different executions of a superpath are generally not identical in terms of MLP, so what we want to associate with the superpath is a dominant value for its MLP-distance. To manage this, in addition to keeping an MLP-distance prediction for each superpath entry, we employ a 3-bit saturating confidence counter which indicates our confidence that the stored MLP-distance is also the dominant value. The confidence counter is incremented for each MLP-distance update which agrees with the current prediction and decremented for each update which disagrees. When it reaches zero we replace it.

6 Evaluation

6.1 Experimental Setup

For our experiments we use a detailed cycle accurate simulator that supports a dynamic superscalar processor model and WATTCH power models [15]. The configuration of the simulated processor is described in Table 1. We use a subset of the SPEC2000 benchmarks, containing benchmarks with more than one long-latency load per 1K instructions for the smallest cache size we utilize. Benchmarks with even less misses present no interest for our work, since without misses our mechanism falls back to the baseline ILP-feedback technique. All benchmarks are run with their reference input. We simulate 300M instructions after skipping 1B instructions for all benchmarks except for vpr, twolf, mcf where we skip 2B instructions and ammp where we skip 3B instructions.

6.2 Results Overview

The experiments were performed for four different cache sizes. Three metrics are used in our evaluation: total processor energy, execution time and the energy × delay product. We first present the effects of MLP-awareness on the baseline ILP-oriented mechanism and then a direct comparison of the two mechanisms, the ILP-aware and the combination of the ILP-feedback and ILP/MLP techniques. All results (except otherwise noted) are normalized to the base case of an unmanaged IQ.

Table 1. Configuration of simulated system

Parameter	Configuration
Fetch/Issue/Commit width	4 instructions per cycle
BTB	1024 entries,4-way set-associative
Branch Predictor	Combining, bimodal + 2 Level, 2 cycle penalty
Instruction Queue (combined with ROB)	128 entries
Load/Store Queue	64 entries
L1 I-cache	16 KB, 4-way, 64 bytes block size
L1 D-cache	8 KB, 4-way, 32 bytes block size
Unified L2 cache	256 KB/512 KB/1MB/2MB, 8-way, 64 bytes block size
TLB	4096 entry (I), 4096 entry(D)
Memory	8 bytes wide, 120 cycles latency
Functional Units	4 int ALUs, 2 int multipliers, 4 FP ALUs, 2 FP multiplier

6.2.1 Effects of MLP-Awareness

Figure 4 depicts the EDP *geometric mean* of all benchmarks for two different thresholds: an aggressive threshold of 768 instructions and a conservative threshold of 256 instructions. Note how difficult it is to improve the geometric mean of the whole-processor EDP with IQ resizing. This depends a great deal on the portion of the total power budget taken up by the IQ. In our case, this is a rather conservative 13.7%, so in the ideal case —significant energy savings and no performance degradation— we expect to approach this percentage in EDP savings. Indeed, this is what we achieve with our proposal.

As shown in the graph, the ILP-feedback technique works marginally with a conservative threshold while its combination with the MLP-aware mechanism improves the situation only slightly. However, with much more aggressive resizing, the ILP-feedback technique seriously harms performance and despite the larger energy savings, yields a worse EDP across all cache configurations. In this case, the incorporation of the MLP-aware mechanism can readily improve the results and turn loss into significant benefit, approaching 10% EDP improvement.

As long as the ILP-feedback technique does not hurt the MLP, it yields benefits. When that changes, the performance loss is unacceptable. This hazard is avoided when the MLP mechanism is used because it works as a safety net. With the help of MLP mechanism, resizing can be pushed to the limit.

6.2.2 Direct Comparison of ILP- and ILP/MLP-Aware Techniques

In this section, we compare the behavior of the two approaches, each evaluated for a configuration which minimizes its average EDP. An additional consideration was to find configurations that do not harm EDP over the base case for any benchmark. This, however, is not possible for the ILP-feedback technique, lest we are content with marginal EDP improvements. Thus, for the ILP-feedback we remove this restriction and we simply select the threshold that minimizes average EDP, which is 256-instructions. The ILP-feedback with the MLP mechanism can be pushed much harder as it is evident from Fig.4 with the 768-instruction threshold. However, EDP worsens over the base case for two programs (applu and art), even though the average EDP is minimized. The threshold that gives the second best average EDP —giving up less than 2% over the previous best case— for the combined ILP/MLP mechanism is the 512-instruction threshold which satisfies our requirement for EDP improvement across all benchmarks.

Figures 5, 6, 7 illustrate the normalized EDP, execution time increase and energy savings respectively for the “best” thresholds for each mechanism. The end result is that the very aggressive resizing of the ILP/MLP technique harms performance comparably to the conservative ILP-feedback technique but at the same time manages to

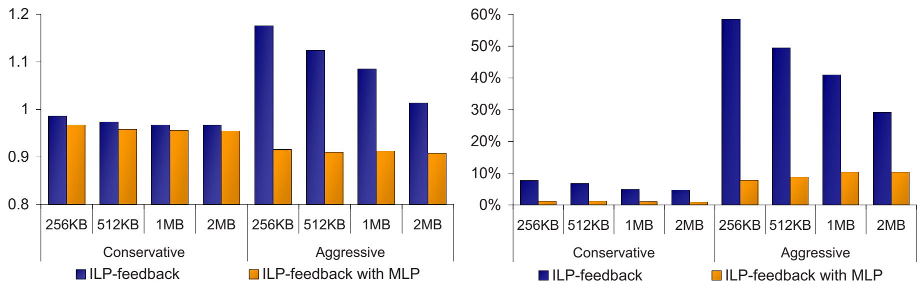


Fig. 4. Average (geometric mean) Normalized EDP (left) and Performance Degradation (right) for ILP-feedback and ILP-feedback with MLP-awareness

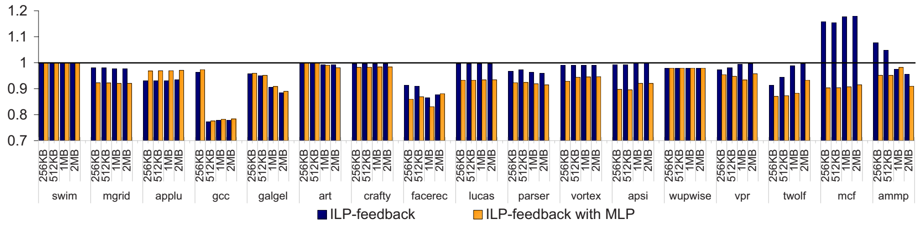


Fig. 5. Normalized Energy-Delay Product for ‘best’ configurations: ILPFeedback (256 threshold) and ILP-Feedback with MLP (512 threshold)

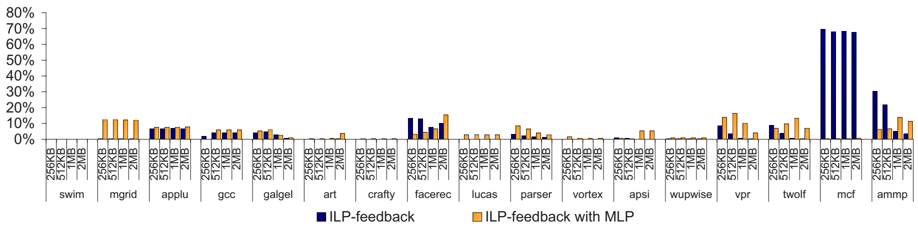


Fig. 6. Execution Time Increase for ILP-Feedback and ILP-Feedback with MLP (each for its best configuration)

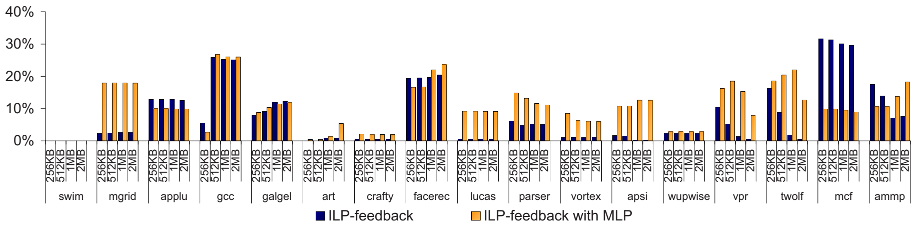


Fig. 7. Normalized Energy Savings for ILP-Feedback and ILP-Feedback with MLP (each for its best configuration)

reduce the IQ size more and produce significantly higher energy savings. This results in an EDP for the ILP/MLP technique that is consistently better than the EDP of the ILP-feedback technique, almost doubling the benefit on average (6.1-7.2% compared to 1.4-3.4% of the ILP-feedback technique). The added benefits of MLP-aware management diminish slightly with cache size, since with fewer misses we have less opportunities for MLP. Finally, note that the performance degradation of the ILP/MLP technique is kept at reasonable levels, while even for the conservative ILP-feedback it can vary considerably more (e.g., mcf execution time increases 67%-69%).

7 Conclusions

In this paper, we revisit techniques for resizing the instruction queue aiming to improve the power-efficiency of high-performance out-of-order cores. Prior approaches resized the IQ paying attention primarily to ILP. In many cases this results in considerable loss of performance while the energy gains from the IQ are bounded with respect to the energy of the whole processor. The result is that EDP improves in some cases but worsens in others making such techniques inconsistent.

The culprit for this is MLP —Memory-Level Parallelism. Resizing the IQ can reduce the amount of MLP in programs with serious consequences on performance. With this realization, we set out to provide a technique that can be applied on top of previous IQ resizing techniques. Our technique, detects possible MLP at runtime and uses prediction to guide IQ resizing decisions. Because we need to manage the whole IQ, our basic unit of management is a sequence of basic blocks, called superpath, comparable in the number of instructions to the maximum IQ size. MLP information is associated with superpaths and is used to override resizing decisions that might harm the MLP of the superpath. In absence of misses and MLP, resizing of the IQ is performed using already existing techniques.

Our results show that we can manage the IQ, considerably better than in prior approaches yielding consistently better EDP over the base case. At the same time, we can push the resizing of the IQ much more aggressively (to achieve better energy savings) knowing that our safety-net mechanism protects the MLP of the program and will not inordinately harm performance.

Acknowledgments. This work is supported by the EU FP6 Integrated Project - Future and Emerging Technologies, Scalable computer ARCitecture (SARC) Contract No. 27648, and the EU FP6 HiPEAC Network of Excellence IST-004408. The equipment used for this work is a donation by Intel Corporation under Intel Research Equipment Grant #15842.

References

[1] Buyuktosunoglu, A., Albonesi, D., Schuster, S., Brooks, D., Bose, P., Cook, P.: A Circuit Level Implementation of an Adaptive Issue Queue for Power–Aware Microprocessors. In: Proc. of Great Lakes Symposium on VLSI Design (2001)

[2] Folegnani, D., González, A.: Energy-effective issue logic. In: Proc. of the International Symposium on Computer Architecture (2001)

[3] Kucuk, G., Ghose, K., Ponomarev, D.V., Kogge, P.M.: Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors. In: Proc. of the International Symposium on Low Power Electronics and Design (2001)

[4] Eyerman, S., Eeckhout, L.: A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In: Proc. of the International Symposium on High Performance Computer Architecture (2007)

[5] Glew, A.: MLP yes! ILP no! In: ASPLOS Wild and Crazy Ideas Session 1998 (1998)

[6] Karkhanis, T.S., Smith, J.E.: A first-order uperscalar processor model. In: Proc. of the International Symposium on Computer Architecture (2004)

[7] Chou, Y., Fahs, B., Abraham, S.: Microarchitecture optimizations for exploiting memory-level parallelism. In: Proc. of the International Symposium on Computer Architecture (2004)

[8] Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y.N.: A Case for MLP-Aware Cache Replacement. In: Proc. of the International Symposium on Computer Architecture (2006)

[9] Sohi, G.S., Vajapeyam, S.: Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors. In: Proc. of the International Symposium on Computer Architecture (1987)

[10] Ghose, K., Kamble, M.B.: Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation. In: Proc. of the International Symposium on Low Power Electronics and Design (1999)

[11] Ponomarev, D., Kucuk, G., Ghose, K.: Dynamic Resizing of Superscalar Datapath Components for Energy Efficiency. IEEE Transactions on Computers (2006)

[12] Iyer, A., Marculescu, D.: Power aware microarchitecture resource scaling. Design, Automation and Test in Europe (2001)

[13] Rotenberg, E., Smith, J., Bennett, S.: Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In: Proc. of the International Symposium on Microarchitecture (1996)
[14] Tarjan, D., Thoziyoor, S., Jouppi, N.P.: CACTI 4.0. Hewlett-Packard Laboratories Technical Report #HPL-2006-86 (2006)
[15] Brooks, D.M., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: Proc. of the International Symposium Computer Architecture (2000)
[16] Gonzalez, R., Horowitz, M.: Energy Dissipation in General Purpose Microprocessors. IEEE J. Solid-State Circuits (1996)
[17] Zyuban, V., Brooks, D., Srinivasan, V., Gschwind, M., Bose, P., Strenski, P.N., Emma, P.G.: Integrated Analysis of Power and Performance of Pipelined Microprocessors. IEEE Transactions on Computers (2004)

Exploiting Inactive Rename Slots for Detecting Soft Errors

Mehmet Kayaalp[1], Oğuz Ergin[1], Osman S. Ünsal[3], and Mateo Valero[2,3]

[1] TOBB University of Economics and Technology
Department of Computer Engineering
Söğütözü Cad. No:43 Söğütözü, Ankara, Turkey
{mkayaalp,oergin}@etu.edu.tr
[2] Universitat Politecnica de Catalunya (UPC)
mateo@ac.upc.edu
[3] Barcelona Supercomputing Center (BSC)
osman.unsal@bsc.es

Abstract. Register renaming is a widely used technique to remove false data dependencies in superscalar datapaths. Rename logic consists of a table that holds a physical register mapping for each architectural register and a logic for checking intra-group dependencies. This logic checking consists of a number of comparators that compares the values of destination and source registers. Previous research has shown that the full capacity of the dependency checking logic is not used at each cycle. In this paper we propose some techniques that make use of the unused capacity of the dependency checking logic of the rename stage in order to detect soft errors that occur on the register tags while the instructions are passing through the frontend of the processor.

Keywords: Microprocessors, soft errors, register renaming, dependency checking logic.

1 Introduction

Soft errors caused by cosmic particles and radiation from packaging are an increasingly important problem in modern superscalar processors. Particle strikes that occur both on the memory structures and the computational logic may cause system crashes if these errors are not detected [1][12]. Parity bits and ECC are widely used in cache memories and some other important parts of the processors [12].

Modern microprocessors use aggressive techniques like out-of-order execution and dynamic scheduling for boosting performance. In order to feed these aggressive techniques that leverage instruction level parallelism, superscalar datapaths try to fetch more than one instruction each cycle. For example Intel's Pentium 4 processor [6] and each processor core in Intel's Core Duo architecture [5] fetches up to 3 microinstructions per cycle from the instruction cache, while the Alpha 21264 fetches 4 instructions per cycle [8]. In practice, the processor cannot fill the whole fetch width due to the speculative nature of branch instructions and the fetch stops after the first taken branch which leads to the underuse of processor resources.

Almost all contemporary processors use register renaming in order to cope with false data dependencies with the exception of Sun's Ultra Sparc [16]. Use of register renaming mandates the use of a mapping table and a renaming logic where a free register is assigned to each result-producing instruction and the dependent instructions get this information in the same cycle. This logic includes dependency checking logic, which contains a number of comparators to compare each and every destination register tag with the source register tags of the subsequent instructions that are renamed in the same cycle. Because of the fact that the processor pipeline is not filled to its capacity every cycle, the comparators of the dependency checking logic are not always utilized.

In this paper we propose techniques that leverage the inefficient utilization of the comparison logic in the renaming stage of the pipeline to detect transient errors that occur in the frontend of the processor. When the full pipeline width is not used it is possible to protect the register tags of the instructions by replicating the register tags into the unused fields of the subsequent instruction slots. We then use this redundant information by employing the already available comparator circuits of the rename logic to detect any errors that occur until the instruction reaches the rename stage. In order to improve the error coverage we also extend our scheme to replicate the tag data to subsequent cycles where rename stage resources are idle.

2 Register Renaming

Register renaming is a widely used technique to remove false data dependencies. The false data dependencies occur because of the insufficient number of architectural registers that the processor offers to the compiler. When the compiler runs out of registers, it uses the same architectural register multiple times in short intervals, which creates a false write-after-write (WAW) or write-after-read (WAR) dependency between the instructions that are in fact not related at all. Modern processors solve this problem by employing a large physical register file and mapping the logical register identifiers produced by the compiler to these physical registers. Consequently a processor that makes use of the register renaming technique needs more physical registers than the number of architectural registers to maintain forward progress [16].

A mapping table is maintained in order to point out the location of the value that belongs to each architectural register. This mapping table is called the "Register alias table (RAT)" or in short "rename table" and it contains an entry for each architectural register that holds the corresponding physical register that holds the last instance of the architectural register [6].

Each result-producing instruction that enters the renaming stage of the processor checks the availability of a physical register from a list of free registers. If a free register is available, the instruction grabs the register and updates the corresponding entry in the rename table. In some implementations of the register renaming, the instruction has to read and hold the previous mapping of its destination architectural register in order to recover from branch mispredictions or free the physical register that holds the previous value of the architectural register [6]. Each instruction also has to read the physical register identifiers that correspond to the architectural registers that it uses as source operands from the rename table.

In a superscalar processor, the rename table has multiple ports to allow the renaming of multiple instructions per cycle. Every instruction that is renamed together in the same cycle needs to acquire a free register from the free list, update the rename table and read the mappings for its source operands concurrently. Since some of the instructions that are renamed in the same cycle are dependent on each other with WAR or WAW hazards, instructions may try to write to the same entry of the rename table in the same cycle or an instruction may need to wait for a previous instruction to update the rename table before it can read the mappings for its source operands. This kind of sequential access to the rename table may either increase the cycle time or may not be even possible due to some design choices. Therefore register renaming stage of the pipeline includes a dependency checking logic to detect intragroup dependencies.

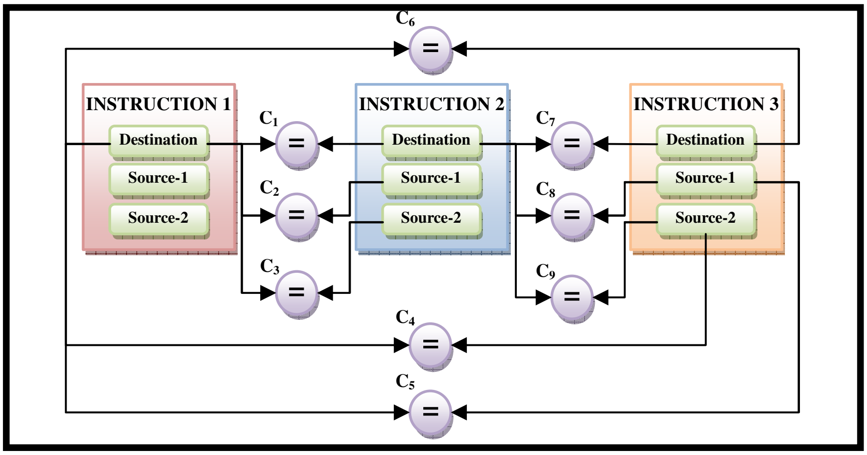


Fig. 1. Comparison circuitry of the rename logic

Fig. 1 shows the structure of the dependency checking logic for a machine that renames 3 instructions concurrently. There are multiple comparator circuits that compare the destination and source tags of all concurrently renamed instructions. Each instruction's destination architectural register tag is compared to the source operand and destination tags of all of the subsequent instructions. In case of a match, the physical register mapping corresponding to the source operand of the subsequent instruction is obtained from the destination physical register field of the preceding instruction rather than being read from the rename table. This way a serial write and read operation is avoided. Similarly, in order to avoid updating the same rename table entry multiple times in a single cycle, destinations of all of the instructions are compared against each other. If a match is detected, only the youngest instruction is allowed to update the rename table. The match/mismatch signals that are produced by the comparators $C_1 \ldots C_9$ are fed into the priority decoders to control the access of the instructions to the mapping table.

3 Using Dependency Checking Logic for Soft Error Detection

Although superscalar processors are designed for high throughput, pipeline width is not fully used from time to time. As the pipeline of the processor is not filled with instructions to its capacity, the number of simultaneously renamed instructions is reduced. This fact was previously observed by Moshovos and was exploited to reduce the complexity and the power dissipation of the rename table [9].

When the number of concurrently renamed instructions is below the processor rename width, dependency checking logic of the rename stage is not employed to its capacity. The comparators that are wired to the empty instruction slots during this period stay idle and generally are not used for any purpose. Therefore during the times when the processor is not using all of its rename slots, these comparators can be used to detect soft errors that occur on the register tags of the instructions when they are passing through the frontend of the processor.

Value replication is a long time known and used technique for reliability. Although it is possible to detect single bit errors in a value by adding a single parity bit, replicating the value can detect and possibly correct multiple errors if there are enough copies of the value. Errors can be detected with one redundant copy of a value and can be corrected with two redundant copies through simple voting [2]. More than three copies of the same data leads to a stronger protection as there will be more chances to recover from an error.

Instruction replication was proposed and implemented in different ways to cope with soft errors in the past. Redundant multithreading was proposed to replicate the whole thread running on the processor to detect any soft errors [11][13][19]. While it offers a system level protection, replicating each and every instruction in a program results in some performance degradation as processor resources are divided into two to execute instructions from both the leading and the trailing threads.

Selectively replicating most vulnerable instructions and processor structures have been proposed as a good tradeoff between performance degradation and fault coverage. However, most of the previous approaches have concentrated on protecting the backend structures of the processor such as the functional units, the issue queue or the reorder buffer [2][7][18].

In this paper we propose to replicate the register tags of the instructions into the register tag fields of the unused instruction slots and use the idle comparators of the dependency checking logic to detect and correct soft errors that occur in the frontend of the processor.

3.1 Protecting the Tags of a Single Instruction

When there is only one instruction flowing through the pipeline, all of the hardware resources can be used for this single instruction. It is possible to detect the errors on both the source tags and destination register identifier if the pipeline width is at least 4, without adding any additional comparator circuit. In order to maximize the error coverage, the tags of the single running instruction is copied to the empty fields of the unused slots as shown in Fig. 2 as early as possible in the pipeline. This copy operation is mostly likely to happen when the instruction is decoded. Also the copies may be ready right after fetching if a trace cache, where decoded instances of the

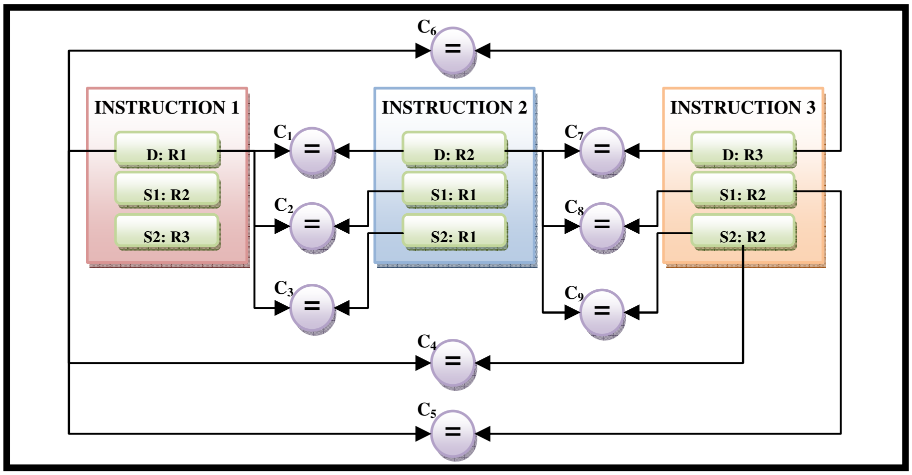


Fig. 2. Error detection example for single instruction renaming

instructions are stored [14], is employed. The instruction slots that hold the redundant information are marked as "bogus" in order to let the rename logic know that these tags do not belong to real instructions.

After the single instruction's tags are replicated in the empty slots, outputs of the already wired comparators at the rename stage are checked to see if an error occurred by the time instruction arrived at the rename stage. As seen in Fig. 2, if the outputs of the comparators C_2 and C_3 mismatch it can only be the result of an error on the destination register identifier of the instruction. Similarly, the outputs of the comparators C_8 and C_9 are checked in order to detect an error on the first source tag of the instruction. As for the destination tag, in fact even a mismatch signal in C_2 or C_3 indicates that an error has occurred. Using the output of the both comparators actually gives a chance to correct the error if one assumes a single event upset model.

In order to cover both source tags and the destination tag of the single instruction the processor needs to be capable of renaming at least 4 instructions each cycle. Although the Fig. 2 is shown for a 3-wide machine for simplicity, the second source tag of the first instruction is copied to the destination field of the third instruction as it would be done in a 4-wide machine.

3.1.1 Single Event Upset Model

If we assume that the fault-rate is sufficiently low, then we can statistically ignore the probability of a multiple-bit flip. This is typically called the Single Event Upset model. Under this assumption, the bit flip can not only be detected but corrected as well. Consider Fig. 2, and assume that there was a bit flip affecting R1. There are two possible cases:

a) The bit flip could occur in the destination field of instruction 1; then *both* C_2 and C_3 comparators would signal a mismatch, indicating that there was a strike to the destination field of instruction 1. The field is updated by copying the field of either S1 or S2 of instruction 2, which hold the copies of R1.

b) The bit flip could occur on the copies of R1 held in the S1 or the S2 fields of instruction 2. In this case a mismatch in either C_2 or C_3 would indicate a flip in S1 or S2 respectively. The error could then be corrected by updating the faulty field or would not be corrected at all since the actual tag value is free of errors.

3.1.2 Multiple Events Upset Model

If the fault-rate is high, then multiple bit-flips could occur. The fault-recovery for this model is slightly more complicated; however even for this case we can detect the fault. In this model, it is not for sure that the "correction" will lead to the actual value of the tag. For example if C_2 indicates a mismatch and or C_3 indicates a match it is logical to think that an error occurred on the first source field of instruction 2. However it is also possible that both the real tag and the copy residing in the second source field may be erroneous. Assuming that the actual tag is free of errors in this case may be wrong although the probability of actual tag being correct is high. Therefore it makes sense to flush the pipeline and refetch the instruction if any of the comparators give a mismatch signal.

It is possible to correct errors that occur on the tags of the single instruction. However this comes at the cost of sacrificing the protection on one of the tags since there are not enough comparators to check three tags in a 4-wide machine. In order to correct an error on R1 for the example shown in Fig. 2, the tag R1 has to be copied to the destination field of the instruction 2. This way there will be 3 comparisons for R1 and simple voting can be employed. Note that different from regular voting used to correct or detect errors, more copies of the same value are held here since we can only compare a destination tag to a source or destination tag of another instruction but we cannot compare two source tags with each other using only the already available hardware resources. Although not shown in Fig. 2, source register R3 is also copied to the source fields of the instruction slot 4 for protection in a processor with a 4-wide rename stage.

When the destination tag is replicated to all of the fields of the second instruction, an error in the actual tag will result in a mismatch signal at C_1, C_2 and C_3 at the same time. By copying one of the replicated values back into the actual tag area corrects the problem. However if there is even 1 match signal, it probably means that the replicas are corrupted since having two errors, one on the actual value and one on a replica, that will result to the same faulty value is a low probability. Yet again, it may be a good choice to flush the pipeline and refetch the instruction in such a case.

3.2 Protecting the Tags of Two Instructions Renamed in the Same Cycle

The processor renames one instruction at a cycle only if there is a problem. This problem may be a cache miss, a taken branch or a processor resource that causes a bottleneck temporarily. When the processor starts to execute the program faster, the number of instructions renamed per cycle increases. However as the throughput of the processor increases, the benefits of our technique decrease as more comparators start to be employed for their real purpose.

For a 4-wide processor, when only two instructions are renamed together, it is possible to correct an error that occurs on both of the destination tags of the instructions but we cannot detect or correct any errors occurring on the source tags at the same time. Alternatively we can copy the destination tag of one of the instructions to the source tag fields of instruction 3 and one source tag of the same instruction both to the destination tag of the third instruction and the source tags of the fourth instruction slot. This way it is possible to protect the destination tag and one of the source tags of one of the instructions.

3.3 Renaming More Than 2 Instructions Simultaneously

Our proposed technique can also provide partial soft error detection coverage if three instructions are renamed simultaneously. In this case only an error on the destination tag of one of the instructions can be detected by copying this tag to all of the fields (destination + sources) of the fourth instruction slot. In a 4-wide processor, the proposed technique won't be able to provide any soft error detection coverage since there won't be any empty slots or idle comparators.

3.4 Using the Bubbles in the Pipeline to Improve Soft Error Coverage

Our proposed scheme cannot offer any soft error detection coverage if the pipeline is fully employed. Also when there are more than one instruction flowing through the pipeline there are not enough storage slots to cover all of the source and destination tags. However the processor pipeline is occasionally not fully employed which offers a whole set of resources to check tags of one instruction. It is possible to copy the tag information of an instruction to a following empty stage to achieve error coverage. In this case there are different options that a designer may choose:

1. One of the instructions (the instruction that is the last in program order) falls behind to the previous stage where it can replicate all of its tags into the storage space for full soft error coverage. This solution may introduce a delay penalty as the instruction that falls back is delayed inside the pipeline, which will delay its entrance to the issue queue and may delay its issue to the function units. We evaluated this scheme but interestingly the soft error vulnerability increased while the performance dropped.
2. Tags of one of the instructions are copied to the next cycle and when these bogus tags arrive at the rename stage, they are compared against the original tag by holding onto this tag in the rename stage. This solution does not result in performance degradation but requires some hardware support. In this work we used this scheme.
3. All of the instructions copy their tags to the empty slot in the previous stage. However in this case there should be a hardware design that supports the checking of tags between stages. This may be accomplished by copying the entire stage and checking the outcomes of all comparators by latching the comparison outcomes. This mandates the use of a single-bit flip-flop (or a latch) for storing the outcomes of all comparators at the rename stage and an extra comparator should be employed to compare the contents of the latches with the outcomes of the comparators. In the examples depicted in Fig. 1 and Fig. 2 there are 9 comparators which means that there will be a 9 bit comparison outcome signature. If this outcome is latched and compared against the 9 bit comparator outcome signature of the replicated bogus tags in the next cycle any mismatch would mean a soft error. Therefore it is possible to protect the entire group of tags by using 9 latches and one 9 bit comparator. Note that this scheme shows an error if there is a mismatch in the signatures but a

signature match does not necessarily mean that the tags are error free since only the single bit outcomes of already existing comparators are compared. The comparators could have mismatched in the previous cycle and can mismatch again erroneously. Since this scheme does not offer full coverage we did not evaluate it.

4 Simulation Methodology

We used the PTLsim simulator that is capable of simulating x86 instruction set in order to get the percentage of the instructions whose tags are protected by the proposed technique. PTLsim simulates a Pentium 4-like microarchitecture and accurately models pipeline structures such as the issue queue, the reorder buffer, and physical register file which are implemented separately inside the simulator. For each benchmark, we simulated 1 billion committed instructions. Table 1 shows the simulated processor configuration.

Table 1. Configuration of the Simulated Processor

Parameter	**Configuration**
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	32 entry issue queue, 48 entry load/store queue, 64–entry ROB, 128-entry Register File
Function Units and Latency (total/issue)	Integer ALU (6/1), load/store unit (2/2), integer multiply (2/4), integer division (1/32), floating-point addition (2/6), floating-point multiplication (2/6), floating-point division (2/6). 6 integer, 2 floating point function units in total.
L1 I–cache	16 KB, 4–way set–associative, 64 byte line, 1 cycle hit time
L1 D–cache	32 KB, 4–way set–associative, 64 byte line, 2 cycles hit time
L2 Cache unified	256 KB , 16–way set–associative, 64 byte line, 6 cycles hit time
L3 Cache unified	4 MB, 32–way set– associative, 64 byte line, 14 cycles hit time
BTB	1024 entry, 4–way set–associative
Branch Predictor	64K entry bimodal and two level combined
Memory	140 cycles latency

5 Results and Discussions

In order to achieve high soft error detection coverage on the register tags with the proposed technique, the number of simultaneously renamed instructions per cycle needs to be as small as possible. This observation is against the general rule that the faster the processor gets the less empty the pipeline is. However since the invested hardware is minimal in our technique, detecting even the small number of errors would be beneficial.

Fig. 3 shows the number of concurrently renamed instructions for spec 2000 benchmarks. Having no instructions in the rename stage does not have any benefits for the initially proposed techniques. In fact the tag fields of the instructions are not vulnerable at all when they do not contain any valid information [10]. The figure shows that the simulated processor frequently uses the full rename width or the rename stage is empty on average. On the average across all spec 2000 benchmarks in more than 40% of the utilized cycles the full processor renaming capacity is not used. This result is consistent with the findings of Moshovos in [9] although he used a

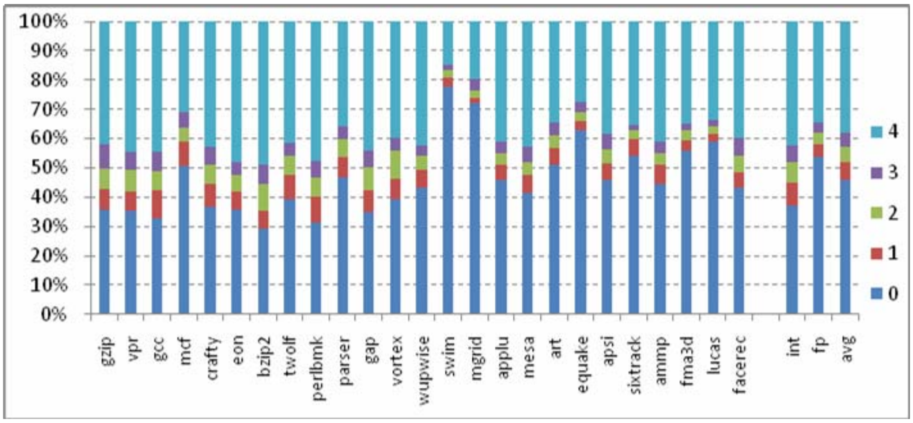


Fig. 3. Number of concurrently renamed instructions

different instruction set. These results show that the pipeline stages are full most of the cycles as opposed to our previous findings obtained by using M-Sim [16] for Alpha 21264 instruction set [3].

Fig. 4 shows how much the rename stage is employed in two consecutive cycles. Each horizontal set of bars represents a different benchmark and for each benchmark the statistics show the frequency of consecutive employment situation for the rename stage. For example, the value 44 on the x axis shows how frequent 4 instructions are renamed for two consecutive cycles. The results show that the processor usually switches between empty stage and full stage but does not usually switch from half-full to half-full. The frequency of 12 and 23 situations show this behavior. Also as it can be seen from the figure, 40, 30, 20 and 10 situations are quite common (more than 10% on average). Therefore it makes sense to copy a single tag to a following bubble slots at a full pipeline stage for soft error detection coverage.

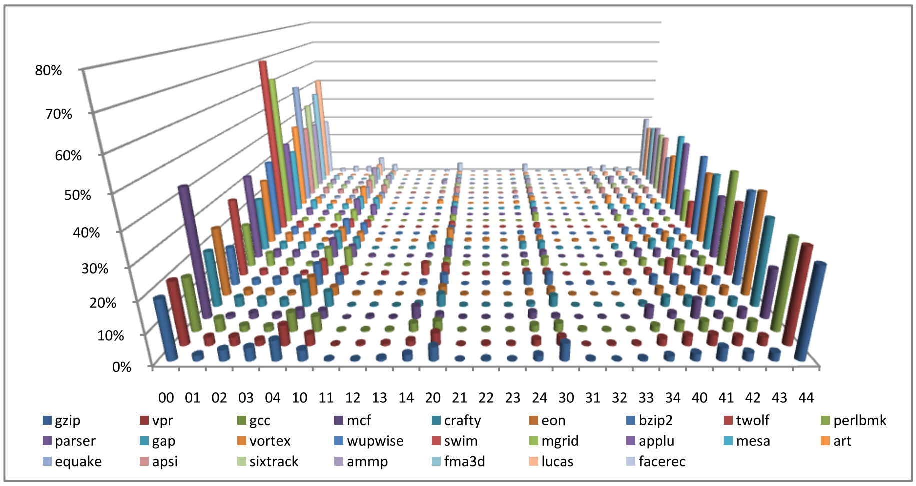


Fig. 4. Frequency of employment at the rename stage for two consecutive cycles

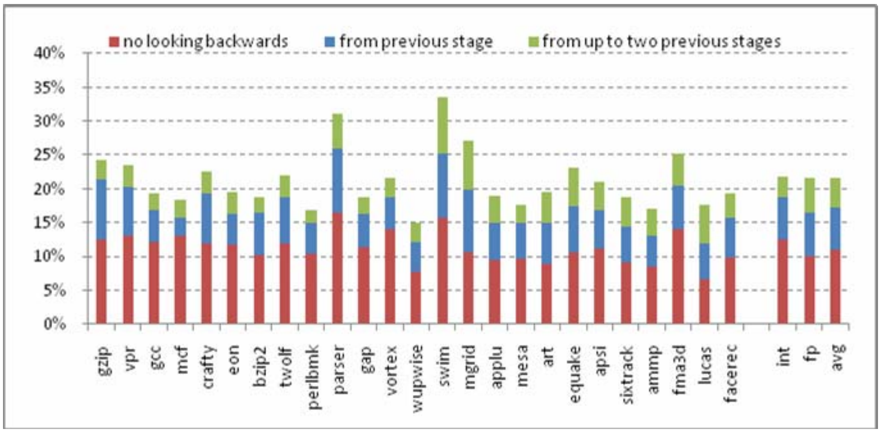


Fig. 5. Reduction of soft error vulnerability in the register tags

Fig. 5 shows the vulnerability reduction achieved in the register tags of the instructions by applying the proposed techniques. The bottom part of each bar shows the vulnerability reduction achieved by replicating the source and destination tags of 1 instruction or 2 instructions that are renamed in the same cycle. The middle part of the bar shows the vulnerability reduction achieved by employing the empty stages that come right after an employed rename stage cycle. In this case one of the tags is copied to the slots in the previous stage just after the fetching of the bubble into the pipeline. Finally the top part of the chart is the result when a tag is replicated for soft error detection to an empty slot that comes two cycles after a full stage. As the results reveal there are diminishing returns when we try to exploit farther empty slots while the hardware complexity to check for an error is increased.

6 Conclusion and Future Work

In superscalar processors multiple instructions are fetched, decoded, renamed and dispatched each and every cycle. In order to solve the intra group data dependencies, comparator circuits are employed in the rename stage of the pipeline which check if the destination of an older instruction is equal to the sources of a younger instruction. Frequently, because of control dependencies, full fetch width of the machine is not used and the comparators in the rename logic are kept idle until another set of instructions arrive. In this paper we proposed a scheme to detect and correct soft errors by replicating the instruction tags in the frontend of the processor and later checking the equality of these replicated tags at the rename stage of the pipeline by employing the unused comparator circuits. We also show that frequently no instructions are fetched from the memory and the unemployed processor resources during those cycles can be used for error detection for the previously fetched instructions. Our results show that, on the average across all spec benchmarks, more than 20% of the register tags can be protected against any soft errors in the frontend of the processors by employing the proposed techniques.

The results of our study shows an upper bound for the benefits that can be achieved by using the dependency checking logic of the rename stage since it is assumed that each and every instruction uses the destination and source tags. In reality, many instructions don't use some or all of these tag fields and this observation was used for different purposes by many researchers [3][15][20]. By using the unused space inside the instruction slots, it is possible to use the on-chip comparators for detecting more errors on the register tags. The investigation of how to use the unused register tag space for soft error detection is left for future work.

Acknowledgments

This work was supported in part by the Scientific and Technological Research Council of Turkey (TUBITAK) through the research grants 107E043 and 109E43, by the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) through the "Reliable Embedded Processors" research cluster and by the European Union (FEDER funds) under contract TIN2007-60625.

References

1. Baumann, R.: Soft Errors in Advanced Computer Systems. IEEE Design & Test of Computers 22(3), 258–266 (2005)
2. Ergin, O., Unsal, O., Vera, X., González, A.: Exploiting Narrow Values for Soft Error Tolerance. IEEE Computer Architecture Letters (CAL) 5, 12–15 (2006)
3. Ergin, O., Yalcin, G., Unsal, O., Valero, M.: Exploiting the Dependency Checking Logic of the Rename Stage for Soft Error Detection. In: 1st Workshop on Design for Reliability (DFR 2009) (January 2009)
4. Ernst, D., Austin, T.: Efficient Dynamic Scheduling Through Tag Elimination. In: ISCA, vol. 30, pp. 37–46 (2002)
5. Gochman, S., Mendelson, A., Naveh, A., Rotem, E.: Introduction to Intel Core Duo Processor Architecture. Intel Technology Journal 10(2) (May 2006)
6. Hinton, G., et al.: The Microarchitecture of the Pentium 4 Processor. Intel Technology Journal 5(1) (February 2001)
7. Hu, J.S., Link, G.M., John, J.K., Wang, S., Ziavras, S.G.: Resource-driven optimizations for transient fault detecting superscalar microarchitectures. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740, pp. 200–214. Springer, Heidelberg (2005)
8. Kessler, R.E.: The Alpha 21264 Microprocessor. IEEE Micro 19(2), 24–36 (1999)
9. Moshovos, A.: Power Aware Register Renaming, Computer Engineering Group Technical Report 01-08-2, University of Toronto (2002)
10. Mukherjee, S.S., et al.: A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: MICRO, pp. 29–40 (2003)
11. Mukherjee, S.S., et al.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: ISCA, vol. 30(2), pp. 99–110 (2002)
12. Phelan, R.: Addressing Soft Errors in ARM Core-based Designs, White Paper, ARM (December 2003)
13. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: ISCA, vol. 28(2), pp. 25–36 (2000)

14. Rotenberg, E., et al.: Trace cache: a low latency approach to high bandwidth instruction fetching. Tech Report 1310, CS Dept., Univ. of Wisc. - Madison (1996)
15. Sangireddy, R.: Reducing Rename Logic Complexity for High-Speed and Low-Power Front-End Architectures. IEEE Transactions on Computers 55(6), 672–685 (2006)
16. Sharkey, J.: M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. Technical Report CS-TR-05-DP01, Dept. of CS, SUNY — Binghamton (October 2005)
17. Sima, D.: The Design Space of Register Renaming Techniques. IEEE Micro 20(5), 70–83 (2000)
18. Sridharan, V., Kaeli, D., Biswas, A.: Reliability in the Shadow of Long-Stall Instructions. In: SELSE-3 (2007)
19. Vijaykumar, T.N., Pomeranz, I., Cheng, K.: Transient-Fault Recovery Using Simultaneous Multithreading. In: ISCA, pp. 87–98 (2002)
20. Yalçın, G., Ergin, O.: Using Tag-Match Comparators for Detecting Soft Errors. IEEE Computer Architecture Letters 6, 53–56 (2007)
21. Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: Performance Analysis of Systems & Software, pp. 23–34 (2007)

Efficient Transaction Nesting in Hardware Transactional Memory*

Yi Liu[1], Yangming Su[1], Cui Zhang[1], Mingyu Wu[1], Xin Zhang[2], He Li[2], and Depei Qian[1,2]

[1] Sino-German Joint Software Institute, Beihang University, Beijing 100191, China
[2] Department of Computer, Xi'an Jiaotong University, Xi'an 710049, China
yi.liu@jsi.buaa.edu.cn

Abstract. Efficient transaction nesting is one of the ongoing challenges for hardware transactional memory. To increase efficiency of closed nesting, this paper proposes a conditional partial rollback (CPR) scheme which supports conditional partial rollback without increasing hardware complexities significantly. In stead of rolling back to the outermost transaction as in commonly-used flattening model, the CPR scheme just rolls back to the conflicted transaction itself or one of its outer-level transactions if given conditions are satisfied. By recording access status of each nested transaction, the scheme uses one global data set for all of the nested transactions rather than independent data set for each nested transaction. Hardware transactional memory architecture with the support of CPR scheme is also proposed based on multi-core processor and current cache coherence mechanism. The system is implemented by simulation, and evaluated using seven benchmark applications. Evaluation results show that the CPR scheme achieves better performance and scalability than the flattening model which is commonly-used in hardware transactional memory.

Keywords: transactional memory, transaction nesting, multi-core processor, programming model, programmability.

1 Introduction

With the rapid development of multi-core and many-core processors, multithreading must be used in programs more than ever in order to utilize processing cores efficiently and promote performance of programs. However, in traditional lock-based programming paradigms, programmer must take many considerations in synchronizing among threads and processes, and even then, deadlock and poor performance still happen in case of inappropriate setting of locks.

Among research work to improve programmability of parallel systems, transactional memory(TM) is an attractive one. Compared to traditional

* This work is supported by National Science Foundation of China under grant No. 60873053 and National Hi-tech R&D Program (863 program) of China under grant No. 2009AA01Z107.

programming models, transactional memory can improve the programmability, avoid deadlock and furthermore, promote performance of concurrent programs.

Transaction nesting is one of the ongoing challenges for hardware transactional memory. To support efficient closed nesting, this paper proposes a conditional partial rollback (CPR) scheme which supports partial rollback conditionally without increasing hardware complexities significantly. Compared to the flattening model which is commonly-used in hardware transactional memory, the CPR scheme rolls back to the conflicted transaction itself or one of its outer-level transactions instead of the outermost if given conditions are satisfied. Transactional memory system architecture with support of the CPR scheme is also proposed based on multi-core processor and current cache coherent mechanisms. The system has been implemented by simulation. Its performance is evaluated by seven benchmark applications. Evaluation results show that the CPR scheme outperforms flattening model in performance and scalability.

The rest of this paper is organized as follows. Section 2 gives an overview of transactional memory. Section 3 discusses transaction nesting models and presents the CPR scheme which supports conditional partial rollback. Section 4 introduces the architecture of our hardware transactional memory. Section 5 evaluates the system with benchmark applications. And Section 6 concludes the paper.

2 Overview of Transactional Memory and Related Work

The concept of transactional memory is first proposed in [1]. It defines transaction as a sequence of instructions which is executed atomically, that is, a transactions is executed either completely (committed) or has no effect (aborted). The atomicity of a transaction is supported at the architecture level, either in hardware or software. During the execution of a transaction, all the modifications to the data are buffered and invisible to the system, if a conflict between two transactions occurs due to reading or writing to the same data, one of these transactions aborts its execution and rolls back. The transaction ends up with commit operation, which makes all the buffered modifications visible to the system.

Compared to traditional lock-based programming model, transactional memory has a series of advantages: firstly, programmer only need to partition the transactions without considering synchronization and mutual exclusion among threads, so the transactional program is easy to write, and programmability is improved; secondly, the synchronization among transactions is done by the system automatically, and a transaction can be aborted and rolled back at anytime, so deadlock can be avoided; and thirdly, multiple transactions can be speculative executed concurrently, and rollback only occurs on conflict, as the result, performance of transaction programs can be improved.

According to the implementation styles, current transactional memory system falls into two categories: hardware transactional memory (HTM) and software transactional memory (STM). HTM supports atomicity of transactions by hardware, and achieves high performance at the cost of resource limitations (e.g. size of a transaction); also the processor architecture must be modified. Typical

HTMs include TCC [2], UTM [3], LogTM [4], TokenTM [5], etc. Compared to HTM, STM [6–10] supports atomicity of transactions by software, and achieves benefits such as flexibility of transactions and various enhanced functions; moreover, STM can be implemented on current processors without hardware modifications. The disadvantages of STM are also obvious: low performance due to the cost of maintaining data consistency and transaction management. Beyond HTM and STM, mixed approaches are also proposed which is called hybrid TM [11–14].

Transaction nesting is one of the ongoing challenges for hardware transactional memory [15]. To achieve both performance and programming flexibility, efficient transaction nesting is needed in HTM. However, most of current HTMs either do not support transaction nesting or support closed nesting by using the inefficient flattening model. One work to achieve efficient nesting is the nested LogTM [16], which supports both open and closed nesting, and for closed nesting, it can rolls back to the beginning of conflicted transaction instead of the outermost transaction. However, this scheme requires that each nesting level maintains independent data set, and will increase complexities and costs of hardware significantly. Compared to nested LogTM, the scheme proposed in this paper just maintains one data set for all nesting levels; the cost is that partial rollback happens only in some specified conditions.

3 Efficient Transaction Nesting in HTM

3.1 Open and Closed Nesting

Transaction nesting occurs when a transaction is executed within another transaction, and commonly exists in calling a subroutine inside a transaction while subroutine itself contains one or more other transactions.

There are two types of semantic models for transaction nesting: closed and open nesting.

(1) Closed nesting

In closed nesting model, a transaction and all of its nested transactions are regarded as integral and its atomicity is guaranteed, that is, either all the nested transactions commit together, or abort and roll back as a whole. The common approach for closed nesting in HTM is the flattening model, in which a transaction and its nested transactions are treated as one transaction, and commit operation is executed at the end of the outermost transaction, on conflict, transaction rolls back to the beginning of the outermost transaction. Obviously, this mechanism is inefficient, especially in deep nesting.

(2) Open nesting

In open nesting model, a transaction and its nested transactions commit independently, and roll back to the beginning of itself instead of the outermost transaction on conflict. Compared with closed nesting, open nesting is more efficient, while on the other hand, it increases programmer's burden. Compensating actions are needed if an outer-level transaction aborts after its inner transaction commits successfully, and this involves complex coding.

3.2 The Scheme of Conditional Partial Rollback

The ideal method to improve performance of closed nesting is: when a transaction aborts due to a conflict with other transactions, it just rolls back to the beginning of itself instead of the outermost transaction. However, this requires that each nested transaction has its own data set, as in the nested LogTM, and will increase complexities and costs of hardware significantly.

To improve the performance of closed nesting in reasonable hardware cost, we propose a scheme which supports conditional partial rollback (CPR) of transactions. The idea of the scheme is: instead of maintaining independent data set for each nesting level, a global data set is used for all of the nested transactions, while the access status of each nesting level to the data set is recorded. When an inner level transaction aborts due to a conflict, it rolls back to the beginning of itself if the data set it accessed has no overlap with other transactions, otherwise it rolls back to the beginning of the outer-level transaction which accessed same data with it, in the worst case, it rolls back to the beginning of the outermost transaction.

In addition, the partial rollback can be further distinguished by read/write accesses. Only when the conflicted transaction has written to the same data with other transactions, it rolls back to the beginning of the outer-level transaction. The reason is that the data updated by the outer-level transactions has been overwritten by the conflicted transaction in the later, and cannot be restored to the beginning status of conflicted transaction, in that case, a rollback to the beginning of the outer-level transaction is needed. In other cases, the conflicted transaction just rolls back to the beginning of itself despite it has accessed the same data with other transactions.

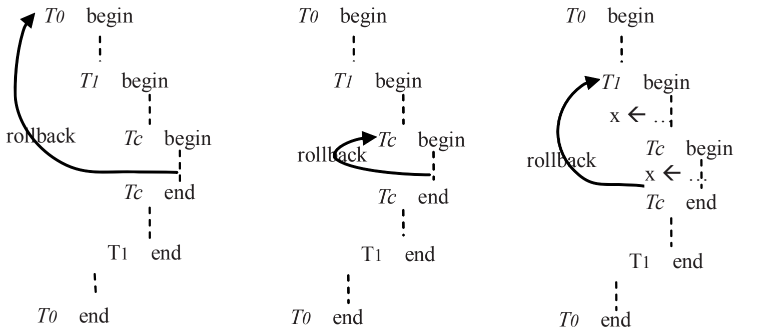
The above CPR scheme can be described formally as following: suppose the number of nesting level is n, the outermost transaction is T_0, the innermost transaction is T_{n-1}, and write set of each nested transaction are: $D_0, D_1, \ldots, D_{n-1}$.

When transaction T_c aborts due to a conflict, it rolls back to the beginning of transaction T_m, and:

$$m = \min\{\forall i \forall j[(D_c \cap D_i \neq \emptyset) \wedge (D_i \cap D_j = \emptyset)]\} \ . \qquad (1)$$
$$(i = 0, 1, 2, \ldots, c; j = 0, 1, 2, \ldots, i-1)$$

According to the above expression, the transaction T_c will roll back to the transaction that is the outermost among the transactions that have overlapped write set with T_c. If there is no overlapped write set between T_c and others, it will just roll back to the beginning of itself.

Fig. 1 gives an example of conditional partial rollback. According to the flattening model for closed nesting, an inner transaction Tc rolls back to the outermost transaction when conflict occurs, as shown in Fig. 1(a). For CPR scheme, the transaction rolls back to the beginning of itself if there is no overlap between its write set and others, as shown in Fig. 1(b); otherwise it rolls back to the outer transaction which has overlapped write set with Tc , as shown in Fig. 1(c).



(a) Flattening model (b) CPR: no overlapped data between T_c and outer transactions (c) CPR: overlapped data between T_c and T_1

Fig. 1. Example of conditional partial rollback

3.3 Hardware Support

Fig. 2 shows the structure of transactional buffer supporting conditional partial rollback, which is used to buffer the data accessed by transactions speculatively, and to store related status for committing or rollback. The structure of transactional buffer is similar to cache and data is also stored by line. The difference is that transactional buffer holds both old and new version of data for each line, where the old version is the data that the outermost transaction started with, and the new version is the current version updated by the transaction speculatively. In addition, there are n-bits read and write vector for each line to indicate whether the line has been read or written speculatively, where n is the maximum number of nesting levels supported by the scheme, more deeper levels will cause the scheme falls back to flattening model. Each bit in vector corresponds to one level of nested transactions.

The transactional buffer is in parallel with L1 data cache. Processor uses transactional buffer when executing transactions (i.e. in transaction state), while using L1 data cache when not in transaction state. In transaction state, L1 data cache is substituted by transactional buffer with the same cache coherence mechanism, while memory writes are limited only in transactional buffer. When

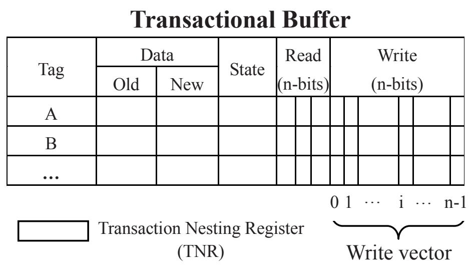


Fig. 2. Structure of transactional buffer

a transaction commits, read and write bits of those updated lines are cleared one by one, which makes the new version of data visible to cache coherence mechanism. When transaction rolls back, data is written back from “old” to “new” for those lines updated by the transaction and inner transactions.

The transaction nesting register (TNR) is used to count nesting levels of transactions. When the processor enters transaction state, which means the outermost transaction is started, the least significant bit of the register is set to 1. The register shifts 1-bit left each time a nested transaction is started, and shifts 1-bit right each time a nested transaction is ended. If the transaction is ended with least significant bit set, which means the outermost transaction is ended, the processor commits the transactional buffer. During the execution of transactions, the bitmap of TNR is also used to set the write-vector in transactional buffer when the transaction writes to a line. If the nesting level exceeds the width of TNR in some extreme cases, the system falls back to the original flattening model for the following inner transactions.

The scheme of conditional partial rollback described in previous section can be implemented in this architecture easily. The write set of a nested transaction corresponds to those lines updated by the transaction, and are labeled in the write-vector of transactional buffer.

4 System Architecture of Transactional Memory

4.1 System Architecture

The transactional memory architecture for the CPR scheme is shown in Fig. 3. The system is based on the multi-core processor architecture, and support execution of transactions by adding some hardware components which is shown in dashed line boxes. Other parts of the core are the same as conventional processors.

4.2 Execution of Transactions

(1) Conflict detection and resolving
Transactions belonging to different threads execute concurrently, and conflict occurs in the following two situations: a thread writes a line which has been read or written by another thread speculatively; or a thread reads a line which has been written by another thread speculatively.

Conflict detection is based on current cache coherence mechanism, such as snoopy or directory-based cache coherence protocol, with which conflicts in above situations can be detected. So, by tracing if a line has been read or written speculatively using read / write bits for each line, a conflict can be detected for sure.

Conflict needs to be resolved after it is detected. The rules to resolve a conflict are: if the conflict occurs between transactional and non-transactional code, the transaction is aborted; if the conflict occurs between two transactions, the hardware always decides in favor of the transaction currently requesting the data,

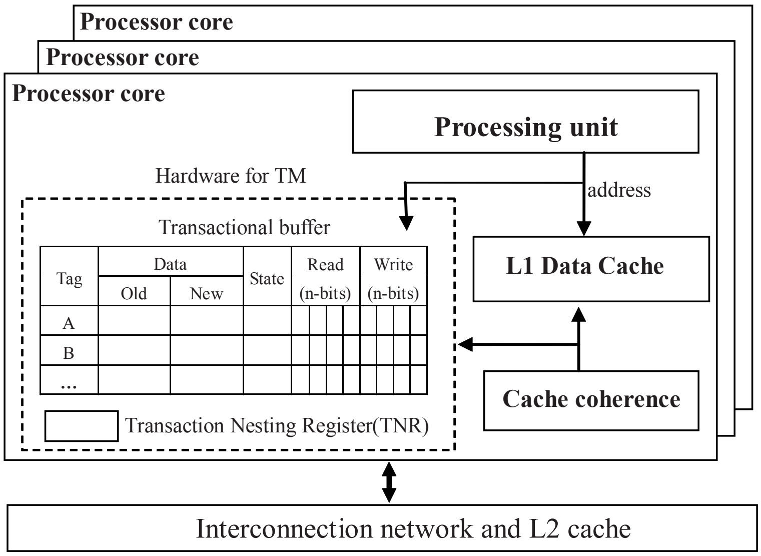


Fig. 3. System architecture

that is, the thread that requesting data currently can continue, while another transaction aborts.

(2) Abort and rollback
When a transaction aborts, firstly the rollback level is determined according to the write-vector in transactional buffer, and all the lines that have been updated speculatively by the rollback level and its inner levels are invalidated, and then an exception is raised and an exception handler is executed which clears related status and restart the transaction.

(3) Commit
The commit operation is executed at the end of the outermost transaction, which makes the buffered contents in transactional buffer valid to the system. To fulfill this, read / write bits in the transactional buffer are cleared line by line, which makes data in the buffer visible to the cache coherence mechanism.

4.3 Instruction Set Extension and Programming Interface

As a hardware transactional memory, the execution of transactions is transparent to programmers, and there is no restriction on programming languages. As shown in Table 1, only two instructions are extended to the instruction set with their corresponding programming interfaces respectively. Programmers just need to partition transactions in their programs, and insert corresponding APIs at the beginning and the end of each transaction.

Table 1. ISA Extensions and Programming Interface

Instruction	Description	Programming interface
XB	transaction start	BEGIN_TRANSACTION()
XC	transaction end	COMMIT_TRANSACTION()

Table 2. Configuration of Target System

Item	Configuration
cache size	L1: 64KB+64KB L2: 16MB transactional buffer: 64KB
cache line size	64 bytes
cache coherence protocol	MESI_CMP_filter_directory
interconnection network	hierachical switch topology

5 Evaluation and Analysis

5.1 Experimental Environment

The CPR scheme and system are evaluated on a simulation platform which is based on the full-system simulator Virtutech Simics [17] and GEMS [18]. By execution-driven simulating, the operating system and applications can be run on the platform.

The transactional memory system proposed in this paper is implemented by extending the simulator. The target system is based on Sparc processor with the extensions of hardware components and instruction set for transaction memory. The number of processor cores varies from 2 to 16. And the operating system is Solaris. Table 2 shows configurations of the target system.

The target system is evaluated using seven micro-benchmark applications as listed in Table 3. These applications fall into two categories: share-h, share-m and share-n have the same number of nesting levels but different densities of data sharing among nesting levels; nest-1 to nest-4 have different number of nesting levels but no data sharing among nesting levels.

Each application is executed in transactional memory with the support of CPR scheme and original flattening model, and performance data is obtained at the same time.

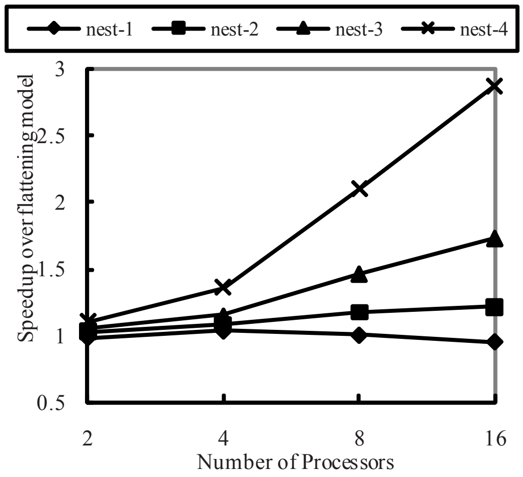
Table 3. Micro-benchmark Applications

Program name	Number of nesting level	Data sharing among nesting levels
share-h	4	high
share-m	4	middle
share-n	4	none
nest1-nest4	1, 2, 3, 4	none

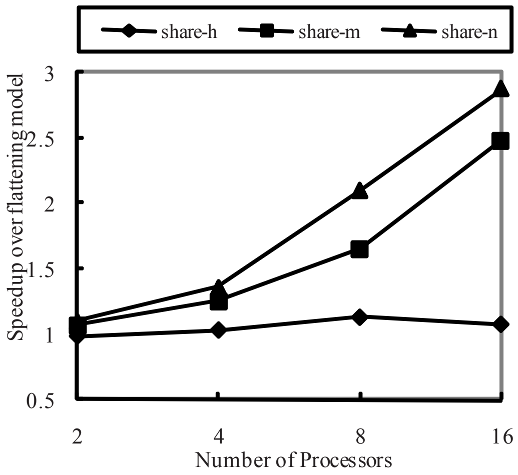
5.2 Results and Analysis

The speedup of CPR scheme over original flattening model is shown in Fig. 4, in which Fig. 4(a) shows the relationship between speedup and number of nesting levels, while Fig. 4(b) shows the relationship between speedup and density of data sharing among nesting levels.

As Fig. 4(a) shows, the speedup is approximate to 1 when nesting levels is 1, at this time, the CPR scheme equals to the flattening model. The speedup value increases with the increasing of nesting levels and processors. It reaches 2.8 for 4 nesting levels and 16 processors. The reason is that the more nesting levels



(a) Speedup of different nesting levels

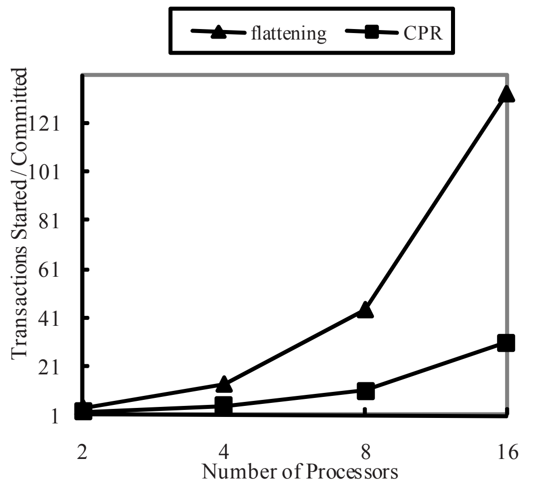


(b) Speedup of different densities of data sharing

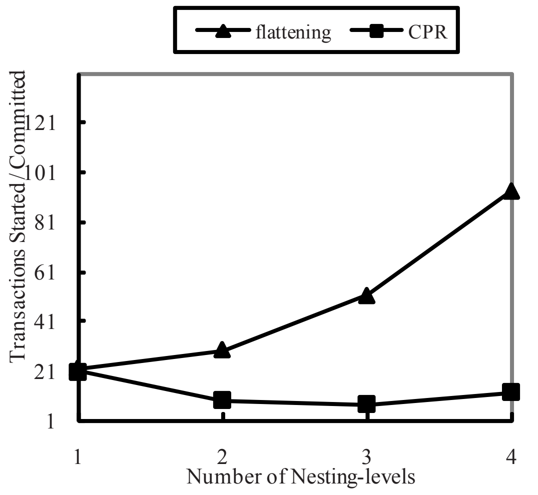
Fig. 4. Speedup of CPR over flattening model

the transactions have, the bigger the difference is between partial rollback and rolling back to the outermost transaction, and furthermore, the probability of conflict increases with the increasing of processors due to more parallel executed threads.

As Fig. 4(b) shows, the performance of CPR scheme is in inverse ratio to the density of data sharing among nesting levels. The higher the density is, the fewer the partial rollback occurs for the CPR scheme, and the speedup over flattening model is more approximate to 1; in contrast, with the density of data sharing



(a) CPR vs. flattening for different processor-numbers



(b) CPR vs. flattening for different nesting levels

Fig. 5. Ratio of started transactions to committed transactions

decreases, partial rollback occurs more frequently, and leading to the increasing of speedup.

Fig. 5 shows the ratio of started transactions to committed transactions. A transaction needs to be restarted when rollback occurs due to a conflict; therefore this ratio reflects the frequency of rollbacks. The ratio value 1 indicates that there is no rollback occurs, and a bigger value means more rollbacks.

As Fig. 5(a) shows, the ratio of flattening model increases rapidly with the increasing of processors, while the ratio of CPR scheme increases slowly, this is because that the conflict occurs more frequently when the processor number increases, and compared to rolling back to the outermost transaction, CPR rolls back partially whenever it is possible, therefore, the number of restarted transactions for partial rollback is far fewer than rolling back to the outermost transaction.

Fig. 5(b) gives relationship between the ratio and nesting levels. It shows that the ratio of flattening model increases rapidly with the increasing of nesting levels, while the ratio of CPR keeps stable and stays at a low level.

Two conclusions can be drawn from the above experiment results: firstly, CPR scheme outperforms flattening model in performance and scalability, in other words, with the increasing of processor number and nesting level, CPR scheme achieves better performance than flattening model; secondly, the performance of CPR scheme is correlative with the density of data sharing among nesting levels.

6 Conclusion

Transactional memory can improve programmability of multi-core processors, avoid deadlock and furthermore, promote performance of concurrent programs. However, it also faces a series of challenges including transaction nesting.

To support efficient closed nesting in hardware transactional memory, this paper proposes a CPR scheme which support conditional partial rollback on conflict without increasing hardware complexities significantly. Compared to the flattening model which is commonly-used in hardware transactional memory, the CPR scheme rolls back to the conflicted transaction itself or one of its outer-level transactions instead of the outermost if given conditions are satisfied. Transactional memory system architecture with support of the CPR scheme is also proposed based on multi-core processor and current cache coherent mechanisms. The system has been implemented by simulation. Its performance is evaluated by seven benchmark applications. Evaluation results show that the CPR scheme outperforms flattening model in performance and scalability.

References

1. Herlihy, M., Eliot, J., Moss, B.: Transactional memory: Architectural support for lock-free data structures. In: 20th International Symposium on Computer Architecture, pp. 289–300. IEEE Computer Society, San Diego (1993)

2. Hammond, L., Wong, V., Chen, M.: Transactional memory coherence and consistency. In: 31st International Symposium on Computer Architecture, pp. 102–113. IEEE Computer Society, Munich (2004)
3. Scott Ananian, C., Asanovic, K., Kuszmaul, B.C.: Unbounded transactional memory. IEEE Micro 26(1), 59–69 (2006)
4. Moore, K.E., Bobba, J., Moravan, M.J.: LogTM: log-based transactional memory. In: 12th International Symposium on High-Performance Computer Architecture, pp. 258–269. IEEE Computer Society, Austin (2006)
5. Bobba, J., Goyal, N., Hill, M.D.: TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In: 35th International Symposium on Computer Architecture (ISCA 2008), pp. 127–138. IEEE Computer Society, Beijing (2008)
6. Shavit, N., Touitou, D.: Software transactional memory. In: 14th annual ACM symposium on Principles of distributed computing, pp. 204–213. ACM Press, Ottawa (1995)
7. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: 11th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 187–197. ACM, New York (2006)
8. Rajwar, Ravi, Goodman, James, R.: Transactional lock-free execution of lock-based programs. ACM Operating Systems Review 36(5), 5–17 (2002)
9. Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V.: Compiler and runtime support for efficient software transactional memory. In: 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 26–37. ACM, Ottawa (2006)
10. Harris, T., Marlow, S., Jones, S.P.: Composable memory transactions. In: 2005 ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming, pp. 48–60. ACM, Chicago (2005)
11. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: 32nd International Symposium on Computer Architecture, pp. 494–505. IEEE Computer Society, Madison (2005)
12. Kumar, S., Chu, M., Hughes, C.J.: Hybrid transactional memory. In: 11th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 209–220. ACM, New York (2006)
13. McDonald, A., Carlstrom, B., Chung, J.W.: Transactional memory: The hardware-software interface. IEEE Micro 27(1), 67–76 (2007)
14. Baugh, L., Neelakantam, N., Zilles, C.: Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In: 35th International Symposium on Computer Architecture (ISCA 2008), pp. 115–126. IEEE Computer Society, Beijing (2008)
15. Harris, T., Cristal, A., Unsal, O.: Transactional memory: An overview. IEEE Micro 27(3), 8–20 (2007)
16. Moravan, M.J., Bobba, J., Moore, K.E.: Supporting Nested Transactional Memory in LogTM. In: 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (October 2006)
17. Magnusson, P.S.: Simics: A Full System Simulation Platform. IEEE Computer (February 2002)
18. Martin, M.M.K., Sorin, D.J., Beckmann, B.M.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News, CAN (September 2005)

Decentralized Energy-Management to Control Smart-Home Architectures

Birger Becker[1], Florian Allerding[1], Ulrich Reiner[2], Mattias Kahl[2], Urban Richter[1], Daniel Pathmaperuma[1], Hartmut Schmeck[1], and Thomas Leibfried[2]

[1] Karlsruhe Institute of Technology – Institute AIFB
76128 Karlsruhe, Germany
[2] Karlsruhe Institute of Technology – Institute IEH
76128 Karlsruhe, Germany
{birger.becker,florian.allerding,ulrich.reiner,
mattias.kahl,urban.richter,daniel.pathmaperuma,
hartmut.schmeck,thomas.leibfried}@kit.edu
http://www.kit.edu

Abstract. In this paper, we focus on a real world scenario of managing electrical demand sets of a smart-home. External signals, reflecting the low voltage grid's state, are used to support the challenge of balancing energy demand and generation. To manage the smart-home's appliances and to integrate electric vehicles as energy storages decentralized control systems are investigated.

1 Introduction

Today, around 80% of the world's primary energy is produced from fossil fuels; renewable energy sources add up to around 4.3% [1]. However, the world is coming to realize, that keeping on burning fossil fuels is not an option in the long run. This affects not only electrical energy production in power plants but also the fuel demand of usual combustion engines in vehicles. Integrating *renewable energy sources* as well as *electric vehicles* into today's energy grid confronts us with a number of problems, especially concerning the variable and hardly predictable nature of the most renewable energy sources and the high demand peaks caused by simultaneous charging of electric vehicles.

In this paper, we introduce self-organizing techniques based on concepts of *organic computing* [2], which support the process of balancing energy production and its demand. For that purpose, we built a *smart-home* in laboratory scale to test the concepts on real hardware. The electrical load profile of several intelligent home appliances can be observed and controlled to a certain degree by a centralized management device. Additionally, a major benefit is the reduction of expensive balancing power [3] by re-scheduling home appliances without cutting down the user's freedom which efficiently improves energy consumption in general.

Furthermore, we examine the integration of an electric vehicle into the *smart-home*. Its energy storage can be used to feed-back electrical energy into the *smart-home* during times of high energy demand in the grid and to reduce load peaks in general.

A selection of prospectively expected issues with the energy grid is pointed out in Sect. 2. In Sect. 3, we describe our demonstrating environment representing the *smart-home*, while in Sect. 4 the generic observer/controller architecture used in our proposal is introduced. Our solution of managing a *smart-home's* architecture is introduced in Sect. 5 and some major experimental results are presented in Sect. 6. The paper concludes with a summary and a discussion of future work in Sect. 7.

2 The Electrical Grid

The average load curve of a typical German residential area grid comprising about 100 households is depicted in Fig. 1. The simulation scenario, presented in [4], examines the uncontrolled charging of 20 electric vehicles with a maximum power of 10 kW additionally to the usual load curve. A very high simultaneity of charging during the evening is assumed. The load of the electric vehicle occurs in addition to the high evening *peak demand*. The *load peak* is increased by 50%, the *load spreading* even by 72%.

Due to uncontrolled interactions between the grid and electric vehicles several bottlenecks might occur: An overload of the transformer, an overload of the power lines, and voltage dips. Depending on the grid topology and the transformer, nominal power load limits could be exceeded. Since each *low voltage grid* is unique, every one of them has to be checked separately for its capabilities. Reinforcement of the grid infrastructure would cause tremendous costs.

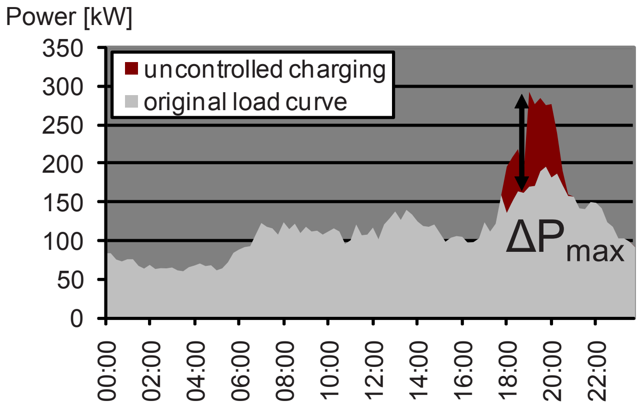


Fig. 1. Load curve in a residential area with uncontrolled electric vehicle charging

Furthermore, the extent of renewable energy sources, like solar power and wind turbines, becomes more and more significant. The former centralized power plant structure will partially be transformed into a distributed generation system. This development is not fully compatible with the current grid structure.

A typical suburban *low voltage grid* structure consists of a transformer, several loads, and generators. Local demand is partly satisfied by *renewable energy sources*. Unfortunately, demand does not comply with the power generation. Renewable power is quite often provided in a period of low demand. This results in increased grid losses, because power has to be transformed to the upper voltage level.

To avoid the afore mentioned bottlenecks and to efficiently integrate *renewable energy sources* into the grid, an intelligent load management is mandatory. Furthermore, it is necessary to evolve from a demand depending generation to a generation depending demand. To reduce the *peak demand*, shown in Fig. 1, control of demand and generation is necessary.

In [3], an external management structure of the *low voltage grid* in a residential area is addressed, whereas this paper has its focus on managing the in-house appliances with respect to energy efficiency. Increasing the resident's comfort is not the central point of this approach. Therefore, we introduce a hierarchical structure; alternatively a peer-to-peer approach is presented in [5]. In this approach, several pools of appliances are formed to balance demand sets and power generation. Such a pool contains a set of decentralized power suppliers and consumers. The needed balance can be achieved, if the consumers' demand set profiles are adjusted to the power generation in the pool.

3 Demonstrating Environment

To analyse the suggested improvements on real hardware components, a demonstrating environment has been developed. It represents some significant electrical components of modern *smart-home* architectures, as is evident from Fig. 2. The entire assembly is connected via electric powerline and supports a maximum of nine electrical consumers, of which six are constructed to behave like *intelligent appliances*.

The in-house-installation is controlled by the *smart-home management device* (SHMD). The intelligent appliances are connected through six relays so that each of them can be switched on or off independently. To be called *intelligent*, they should be aware of their current situation and reasonably respond to it. Each intelligent consumer has its own sensory equipment to get electrical data like voltage, current, and active power.

The intelligent responses of each appliance are realized by an embedded system, which periodically polls sensors and provides averages of the measured values via HTTP-interface in XML format separately for each intelligent appliance. The sum of all consumers' power consumption in the *smart-home* can also be requested at the smart power meter, continuously. Additionally, the local controller unit of each appliance receives control-signals from the SHMD to toggle

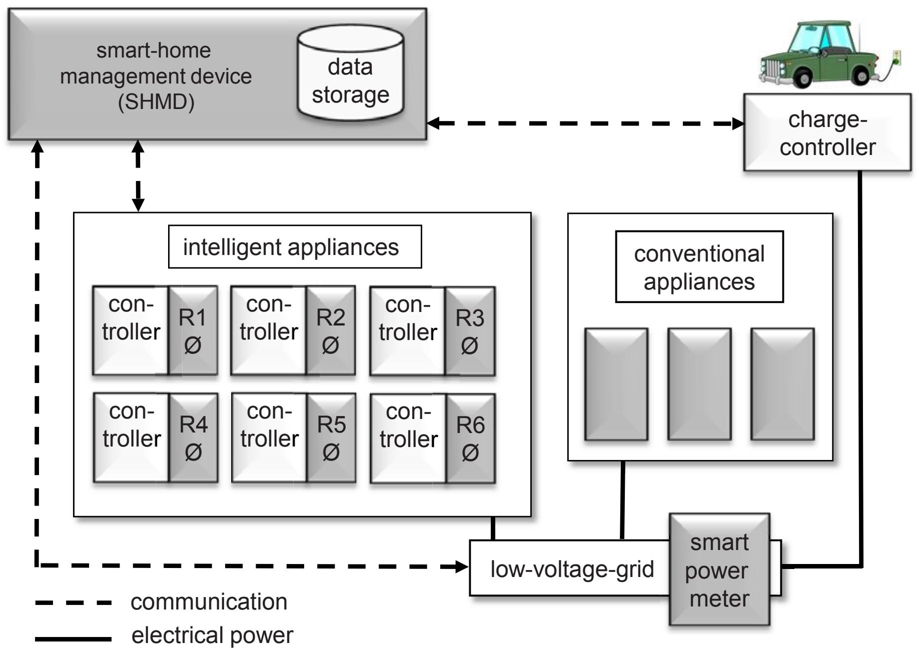


Fig. 2. Demonstrating environment

the on-state of its connected intelligent appliance. In this way, a hierarchically structured observer/controller architecture, cf. Sect. 4, is used.

In the demonstrating environment, it is assumed that an electric vehicle is present in the *smart-home*. For that purpose, a lead-acid storage battery with a capacity of 4 kWh is connected to the *smart-home* demonstrator. It represents the *mobile energy storage* of the electric vehicle. The charging- and discharging process is controlled and observed by a special charge controller which is connected via *powerline* to the SHMD.

The SHMD is the central device in the scenario presented here. Any relevant data, like current power-requirement of each intelligent appliance, the system's voltage, or the current total power of the *smart-home*, are collected in its database.

Based on the content of the database, a user interface is generated by the SHMD's web-server to inform the *smart-home's* resident about its current condition and optimization potential. It allows displaying and analyzing the power-data of elapsed time-slots. Furthermore, the web-interface permits the resident to define the degrees of freedom for each intelligent appliance. This value describes the potential of controlling the device's on-state automatically.

The main challenge of future energy management is balancing supply and demand in energy grids. Therefore, smart algorithms are needed to schedule the electrical consumers and especially the car's charging process. A large number of decentralized batteries permits to feed electrical energy back into the grid.

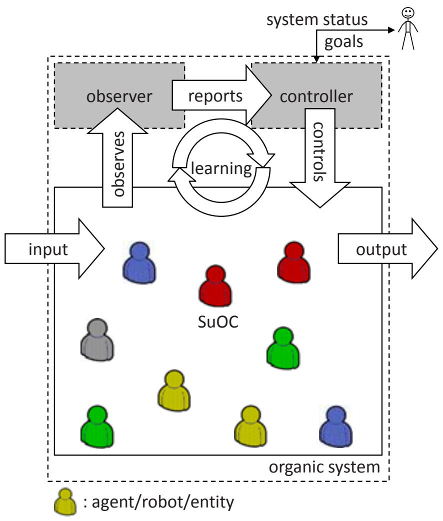
An optimizing algorithm is used on the SHMD to calculate the best schedule for the *smart-home's* electrical consumers for the next 24 hours. In the following, an approach is described to meet this challenge.

4 The Generic Observer/Controller Architecture

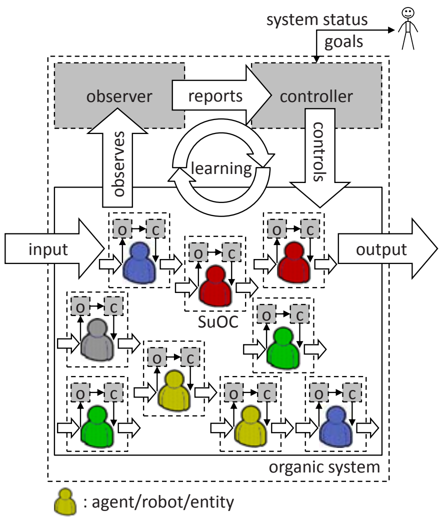
The complexity of technical systems is constantly increasing. Breakdowns and fatal errors occur quite often, respectively. Therefore, the mission of *organic computing* is to restrain these challenges in technical systems by providing appropriate degrees of freedom for self-organized behavior. These technical systems should adapt to changing requirements of their execution environment, in particular with respect to human needs. According to this vision an organic computer system should be aware of its own capabilities, the requirements of the environment, and it should also be equipped with a number of so-called *self-x-properties* [2].

Thus, technical systems are equipped with an observation and control layer called observer/controller architecture, as proposed in [6]. The intention of this design paradigm is to be able to observe and potentially control the systems in order to comply with the objectives given by the user or the developer, cf. Fig. 3(a).

The observer/controller uses a set of sensors and actuators to measure system variables and to influence the system. Together with the *system under observation and control* (SuOC), the observer/controller forms the so-called *organic*



(a) Simplified view of the centralized observer/controller architecture



(b) Hierarchically structured view

Fig. 3. Variants of the generic observer/controller architecture

system. An observer/controller loop enables adequate reactions to control the – sometimes completely unexpected – emerging global behavior resulting from interactions between local agents.

In other words, a closed control loop is defined to keep the properties of the self-organizing SuOC within preferred boundaries. The observer monitors certain (raw) attributes of the system and aggregates them to situation parameters, which concisely characterize the observed situation from a global point of view, and passes them to the controller. The controller acts according to an evaluation of the observation (which might include the prediction of future behavior). If the current situation does not satisfy the requirements, it will take action(s) to direct the system back into its desired range, will observe the effect of the intervention(s), and will take further actions, if necessary. Using this control loop, an organic system will over time adapt to its changing environment. It is obvious that the controller could benefit from learning capabilities to tackle these challenges. The observing and controlling process is continuously executed, and the SuOC is assumed to run autonomously, even if the observer/controller architecture is not present (which might result in suboptimal behavior).

The *centralized* generic observer/controller architecture has been extended in [7]. Here, we specially focus on a *hierarchically structured* observer/controller architecture, where an observer/controller is installed on each system element as well as one for the whole technical system, as depicted in Fig. 3(b).

In particular, e. g., in larger and more complex systems (where the objective space drastically increases) it will be necessary to build hierarchically structured organic computing systems instead of trying to manage the whole system with one centralized observer/controller. In the case of multiple observer/controller levels, the SuOC at the lowest level will consist of simple elements like single software or hardware modules.

Because of the capability of the observer/controller architecture to support the adaptation of a system to changing environmental requirements it has been chosen as a design pattern that is well-suited to cope with the management problems of the *smart-home* scenario addressed in this paper.

5 Observer/Controller Architecture for Smart-Homes

In our approach of controlling a *smart-home*, we decided to implement a hierarchically structured observer/controller architecture, as shown in Fig. 4. Each household appliance is equipped with a local *observer/controller unit* (o/c unit) to observe its current state and to interact with the appliance by turning it on or off. Based on the measured data the local observer generates a specific demand set[1] for each appliance. This data is communicated to the central component, the SHMD.

The observer of the SHMD collects the demand set data from every appliance and generates a global demand set prediction for the *smart-home*. Thereby, the controller of the SHMD decides to re-schedule the demand sets of each appliance

[1] Which characterizes an appliance's power consumption profile.

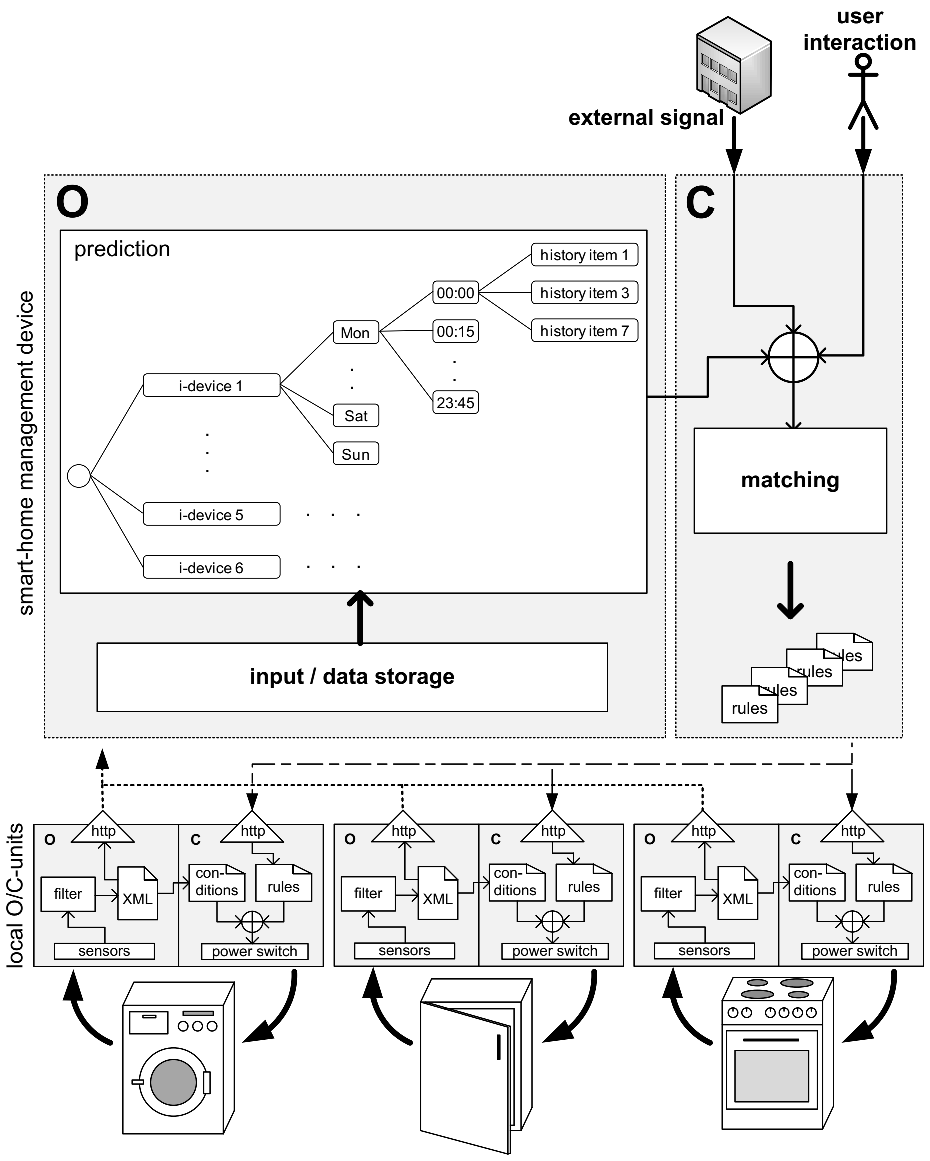


Fig. 4. Overview of the implemented architecture

based on the demand-prediction from the observer and the load-prediction of the global grid, which is communicated by an external 24h-signal. The external 24h-signal is periodically sent by the grid operator to the SHMD. It contains a behavior request for the next 24 hours individually for each *smart-home* and rates all timeslots during the considered period to request the SHMD to re-schedule the appliances' demand sets. A high value of the signal represents a high demand in the grid during a specific timeslot. The aim of the SHMD is to re-schedule demand sets to timeslots rated with the lowest possible signal value. Thus, the grid's balancing process of supply and demand is supported.

In addition to the 24h-signal, the SHMD is able to receive a short-term signal from the grid operator. In this way, the grid operator is able to communicate the condition of a short-term imbalance in the grid to the SHMD. The *smart-home* can react to this signal e. g. by interrupting the charging process of the car in case of requiring positive balancing power, or by starting the charging process in case of the demand of negative balancing power. The SHMD calculates a scheduling of every device to satisfy the given signals on the one hand and to fit the appliance's degree of freedom on the other hand. The signals reflect the predicted load and short-term imbalances in the grid. In this way the SHMD is able to schedule its appliances related to the grid's condition.

Thus, the controller unit of the SHMD tries to match the demand set of each appliance with a certain degree of freedom with the received signals, referred as *matching* in Fig. 4. After this a set of rules can be generated and sent to each appliance. These rules contain instructions for the appliances, in which timeslot they should start or break its operation. The controller unit of each appliance's observer/controller unit contains a simple set of static rules to interact with the appliance. It allows the local observer/controller unit to adjudicate finally, combined with the given rule from the SHMD, whether to switch on the appliance or not. This decision is based on the capability to operate; a washing machine for example should only operate, if it is filled up with laundry.

In [8], a related approach is introduced. Any smart-home is equipped with a *Bedirectional Energy Management Interface (BEMI)*, which is able to control smart appliances and to measure electrical values indicating the grid's condition locally. The BEMI is able to optimize the operation of locally connected controllable appliances. The BEMI-approach focuses the communication between the *Energy Service Provider* and his low voltage grid connection points, while the present approach concentrates on in-house monitoring and controlling based on external signals.

5.1 The Local Observer/Controller Unit

Most of the modern household appliances are complex devices. Thus, it is not possible to interfere directly in the program of the appliance's native controller. For this reason, the appliance is monitored by a local observer, as shown in Fig. 4, and will be managed by its local controller only with the simple command either to operate or not. Each intelligent appliance is fitted with a set of sensors to detect its current state. Depending on the household appliance, different parameters may be measured, and a parameter common to all parameter sets is the power supply. Based on the assumption that for every household appliance there is an approximated typical demand set, we can draw conclusions from the current state of the appliance. The potential of re-scheduling depends on the degree of freedom each appliance has. A set of classes regarding the degree of freedom can be defined. The operation of these appliances with a high degree of freedom like washing machines or thermal storages can be re-scheduled several times in compliance with its local constraints. A stove or a multimedia device, on the other hand, has an extremely poor degree of freedom, so their demand profile can be observed, but there is no possibility of re-scheduling the operation. In our scenario, the degree of freedom is determined by each appliance or by the *smart-home's* resident using the web-interface. Some appliances have to execute a specific program before they can be switched off. Thus, the specific profile for these appliances must be respected in the SHMD while generating the re-scheduling of the appliances. Therefore, the local controller of each appliance has a set of static rules. The decision to change the operation state depends on these rules combined with the rule set from the SHMD.

5.2 Central Observer

The *smart-home's* observer unit captures power changes for each intelligent appliance, the charge condition of the electric vehicle's battery, the devices' degree of freedom, and the known *device profiles*. The latter is received from the intelligent appliance itself or calculated from the past power data. The whole set of values is stored in the SHMD's database. Reduced to the essential, the database contains a list of the attributed set of intelligent appliances and the data history of the power demand for each of them.

Using a sufficient quantity of history data, it is possible to build a prediction, described in Fig. 4, for each day of the week. The history items are classified in data sets by the day of week and a certain timeslot. An appropriate data structure is a tree containing for each intelligent appliance a node on the first level. Each of these nodes branches into seven nodes on the second level, representing the seven days of the week. The weekday node has subnodes for a fixed set of timeslots (e. g., blocks à 5 minutes) of the day. Finally, every data set from the intelligent appliances's history is associated to one of the timeslot nodes.

This part's challenge is to get timeslots, where an appliance is frequently switched on. Each timeslot is represented by the corresponding daytime and weekday. To find these frequent item sets in the tree structure, the timeslot nodes can be easily scanned for such nodes having more than a certain number of leaves in the tree. Having found these timeslots, it is obvious that the associated device will probably be powered up on the same weekday's timeslot during the next weeks. To increase the rate of adaption a digressive weighting depending on the items' age is used. Furthermore, this knowledge can be transformed into a *power forecast* for each intelligent appliance for the next days.

5.3 Central Controller

Reverting to an extensive pool of data collected by the observer of the SHMD, the main controller's basic task is to send adequate rule sets to the subordinated local controllers of the intelligent appliances. The final decision is made by the intelligent appliance's controller, but it should be mostly consistent with the rule sets of the main controller. To simplify our initial approach to the *smart-home* controlling system, we are leaving out the aspect of *learning* on the controller. In further work, we will investigate the controller's aspect of learning, as proposed in Sect. 3. The observer/controller-architecture is an appropriate framework to implement learning techniques.

Depending on the appliance's degree of freedom and the planned schedule, an optimized schedule can be computed, which is based on the 24h-signal of the energy supplier. E. g., a dishwasher is often switched on after lunch at 12:10 p. m. on a certain weekday. It will probably be switched on during the same weekday's timeslot in the next week. Since this timeslot is rated with a high demand by the 24h-signal, it is beneficial to re-schedule the starting time of the dishwasher to a timeslot, where the demand for energy is as low as possible. The appliance's degree of freedom has to be respected. The optimizing process has to determine the best starting timeslot of each appliance. For that purpose, the function of the demand-set and the 24h-signal's rating are considered for each possible starting timeslot. The appliance's start is scheduled to the timeslot with the lowest integral value which is consistent with the operating constraints.

5.4 Integration of an Electric Vehicle

Additionally, the mobile storage in the electric vehicle is used to support the balancing process of demand and generation. Via the battery's charge controller, the SHMD is able to manage the charge- and feed-back-process. Some appliances in the house with a poor degree of freedom cannot be re-scheduled into a period of low demand. E. g., a stove is often switched on for cooking before lunchtime, but this is a period with an extremely high demand. In this case, a vehicle's battery can be used to feed-back electrical energy into the grid. The SHMD can send a control signal to the controller of the electrical vehicle to request the feed-back of a certain amount of electrical power. The start of the charging process of the car's battery can also be scheduled by the SHMD. The charging of a car battery causes a high electrical load. Thus, it is beneficial to re-schedule the charging process into periods of low demand.

6 Results

In the upper diagram of Fig. 5, the *forecast* for the next 24 hours based on the history data's analysis is visualized. Each of the intelligent appliances is scheduled in the time slot interval, where it has been typically powered on during the considered period. The lower diagram represents the optimized schedule by the SHMD. The gray curve shows the behavior request from the grid operator (motivated in Sect. 5). It is shown that, e. g., devices like the breadmaker, the dishwasher, and the stove are located in periods of high demand, hence they should be re-scheduled.

The breadmaker and the dishwasher have a sufficient degree of freedom. Thus, they can be re-scheduled to periods of lower demand, as shown in the lower diagram of Fig. 5. The stove has a poor degree of freedom. Therefore, it cannot

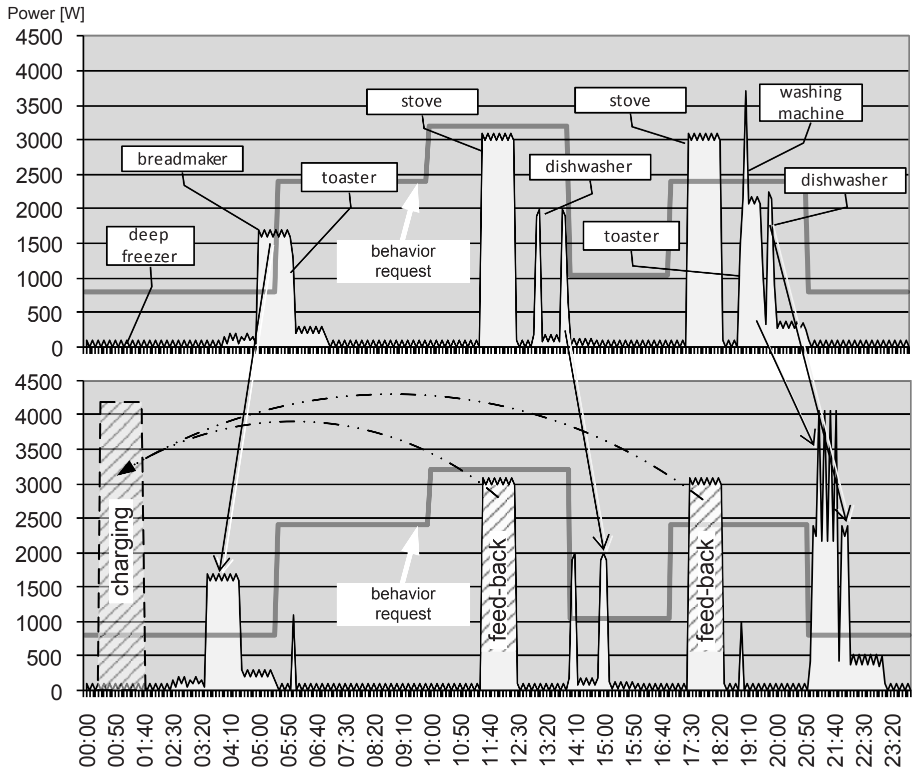


Fig. 5. Re-scheduling based on the behavior request

be re-scheduled. In this case, the power demand of the stove can partially be balanced[2] by feeding-back from the mobile energy storage. This potential is highlighted by the shaded area in the lower diagram. The car's battery has been discharged during the feed-back process. The charging process is scheduled to the early morning by the SHMD. This is a period of low demand. Thus, the demand of the house can, to a certain degree, comply with the given behavior request, and, accordingly, to the predicted load of the local grid.

7 Conclusion

In this paper, a hierarchically structured observer/controller architecture is applied to control a *smart-home*, to increase the energy efficiency, and to avoid overloads of the low voltage grid. The integration of electric vehicles raises the potential of re-scheduling the energy demand of a *smart-home*. This is achieved

[2] The power demand of the stove is higher than the max. feed-back power of the car's battery.

by controlling the charging process and feeding-back energy into the grid during times of high demand. The experimental results have shown the potential of automatic scheduling the *smart-home's* appliances to comply with the external signals given from the grid operator. In this way, the challenge to balance demand and generation can be tackled by each *smart-home*.

In future work, the currently applied static rules for determining the demand of control can be improved by extended learning algorithms to achieve a higher degree of autonomy and to increase the robustness of the management system. Consequently, less user interaction would be required, improving the systems' usability and comfort.

To get the capability of analyzing a more representative user behavior in a *smart-home*, it is planned to extend the environment in a long-running experiment in a real house with a significant number of intelligent appliances and distributed energy generators.

References

1. Schiffer, H.W., Czakainski, M., Gerling, P., Hareiner, H.R., zu Schwabedissen, C.M., Rath, H., Schmid, C., Schupan, J., Seeligmüller, I., Wiemann, M.: Energie für Deutschland. Deutsches nationales Komitee des Weltenergierates, DNK (2006)
2. Schmeck, H.: Organic computing: A new vision for distributed embedded systems. In: Proceedings Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pp. 201–203. IEEE Computer Society, Los Alamitos (2005)
3. UCTE: Operation Handbook. European network of transmission system operators for electricity (2004)
4. Reiner, U., Leibfried, T., Allerding, F., Schmeck, H.: Potential of electrical vehicles with feed-back capabilities and controllable loads in electrical grids under the use of decentralized energy management systems. In: ETG-Kongress (2009)
5. Kamper, A., Eßer, A.: Strategies for decentralized balancing power. In: Biologically-inspired optimization methods. Studies in Computational Intelligence, vol. 210, pp. 261–289. Springer, Heidelberg (2009)
6. Müller-Schloer, C.: Organic computing: On the feasibility of controlled emergence. In: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS 2004), pp. 2–5. ACM Press, New York (2004)
7. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: INFORMATIK 2006 – Informatik für Menschen! LNI, vol. P-93, pp. 112–119. Bonner Köllen Verlag (2006)
8. Nestle, D., Ringelstein, J.: Application of bidirectional energy management interfaces for distribution grid services. In: 20th Int. Conf. on Electricity Distribution, CIRED (2009)

EnergySaving Cluster Roll: Power Saving System for Clusters*

Manuel F. Dolz, Juan C. Fernández, Rafael Mayo, and Enrique S. Quintana-Ortí

Depto. de Ingeniería y Ciencia de los Computadores, Universidad Jaime I, 12071-Castellón, Spain
dolzm@guest.uji.es, {jfernand,mayo,quintana}@icc.uji.es

Abstract. With the increasing number of nodes in high performance clusters as well as the high frequency of the processors in these systems, energy consumption has become a key factor which limits their large-scale deployment. In response, this paper presents a software module that, taking into consideration past and future users' requests, implements energy saving policies, powering on and shutting down the system nodes. This tool employs Roll for clusters with Rocks® as operating system and Sun® Grid Engine as queue system.

Keywords: High performance computing, data centers, green computing, power consumption, queue system.

1 Introduction

High Performance Computing (HPC) clusters have been widely adopted by companies and research institutions for their data processing centers because of their parallel performance, high scalability, and low acquisition cost. On the other hand, further deployment of HPC clusters is limited due to their high maintenance costs in terms of energy consumption, required both by the system hardware and the air cooling equipment. In particular, some large-scale data processing centers consume the same energy power as 40,000 homes, and studies by the U.S. Environmental Protection Agency show that, in 2007, the power consumption of data centers in the United States was around 70 billion KWatt-hour, representing 5,000 million euros and the emission of 50 million tonnes of CO_2 [1].

Since the benefits of HPC clusters are clear, scientists and technicians are currently showing special interest in all types of solutions and ideas to minimize energy costs in data processing centers: Energy-awareness has spread among researchers from organizations like IEEE, which have analyzed HPC clusters, concluding that a significant part of the energy consumed by these systems is due to the interconnection of its components (switches, network cards, links, etc.); following this result, energy-aware algorithms have been developed which can disable

* This research was supported by projects CICYT TIN2008-06570-C04 and FEDER, and P1-1B2009 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

idle interconnections in the cluster [2]. Microsoft inspects the problem from a different viewpoint and one solution proposed is to share highly efficient power supplies among several nodes of the system, achieving significant energy savings [3].

In this context a well-known energy management technique is DVFS (Dynamic Voltage and Frequency Scaling). DVFS entails reducing the system energy consumption by decreasing the CPU supply voltage and the clock frequency (CPU speed) simultaneously. This technique has a great impact on the development of work aimed at reducing consumption in this research context [4,5,6]. The authors in [7] present an energy-aware method in order to partition the workload and reduce energy consumption in multiprocessor systems with support for DVS. Freeh et al. [8] analyze the energy-time tradeoff of a wide range of applications using high-performance cluster nodes that have several power-performance states to lowering energy and power, so that the energy-time tradeoff can be dynamically adjusted. In [9], the authors use economic criteria and energy to dispatch jobs to a small set of active servers, while other servers are transitioned to a low energy state.

Alternative strategies to limit power consumption and required cooling of HPC clusters are based on switching on and shutting down the nodes, according to the needs of the users' applications. An algorithm that makes load balancing and unbalancing decisions by considering both the total load imposed on the cluster and the power and performance implications of turning nodes off is described in [10]. Several policies to combine dynamic voltage scaling and turning on or off nodes to reduce the aggregate power consumption of a server cluster during periods of reduced workload are presented in [11]. Rock-solid [12] and PowerSaving [13] are prototype examples of this strategy which provide little functionality or are still under development. This paper presents a new application which follows the same strategy, but with a much more complete functionality. In particular, our tool allows the definition of different conditions to activate and deactivate and nodes for a full adaption to the requirements of the system administrator and/or the end user. The tool has been designed and implemented as a module (Roll) for Rocks® [14] and employs the Sun® Grid Engine (SGE) [15]. A simulation of the module under real conditions shows that its use combined with a reasonable policy deliver considerable energy savings compared with a conventional cluster in which all nodes are permanently active.

The article is organized as follows: Section 2 presents the hardware and software tools used in the development of the energy saving module, and Section 3 reviews its implementation. Section 4 evaluates the performance of the energy saving module, in terms of impact on the power consumption and execution time of the applications. Finally, Section 5 describes some future lines of work and summarizes the conclusions.

2 Basic Hardware and Software Tools

The target hardware platform used in this work is an HPC cluster that employs the NPACI Rocks® Linux distribution as operating system, equipped with a front-end node that is responsible of the queue system and the new energy saving module (Roll).

The module queries the SGE queue system to collect information on the actual jobs, nodes and queues (`qstat`, `qmod` and `qhost` commands). This is then used to compel the necessary statistics, and apply the power saving policy defined by the system administrator. The module also runs several daemons implemented in Python [16]. These daemons maintain a MySQL database that contains all the information and statistics; they also query the "cluster" database used by Rocks® to extract information about the nodes (e.g., their MAC addresses) to remotely power them on using WakeOnLAN (WOL) [17].

The nodes of the cluster have their BIOS configured with WOL (WakeUp events). Systems that support the PCI 2.2 standard in conjunction with a compatible PCI network card usually do not require a WOL cable because energy is provided through the PCI bus.

3 Implementation of the Energy Saving Roll

In this section, we describe the energy saving module in detail; see Figure 1. The module includes the following major components:

– Three daemons in charge of managing the database, collecting statistics, and executing the commands that power on and shut down the nodes.
– The database that stores all information necessary to make decisions.
– The website interface to configure and administer users' groups as well as set the threshold triggers that define the power saving policy.

We have chosen a modular design, mapping the main functions of the system to daemons (control of queue system, collect statistics, and apply policies of activation and deactivation nodes). Moreover, we have decided to employ a database to ease data mining via SQL. The user interface is web-oriented, seamlessly integrates with Rocks®, and facilitates remote access and administration.

3.1 Daemons

Daemon for epilogue requests. A node of the cluster runs a *epilogue script* provided by the SGE queue system when a job completes its execution, and therefore leaves the queue. This script receives parameters from the SGE executor daemon which are essential for monitoring the cluster and, therefore, for implementing the energy saving policy. As the database is located in the front-end node, it is necessary to send this set of parameters through the network. For this purpose, the node that executes the epilogue script opens a connection via a TCP socket with the epilogue daemon that runs on the front-end node to pass the necessary information.

The epilogue daemon employs this information to perform a series of updates in the energy saving module database, extracting data from the accounting file maintained by the queue system. Updated data comprise the number of jobs for the user responsible of this job, this user's average execution time, and the queue average waiting and execution times.

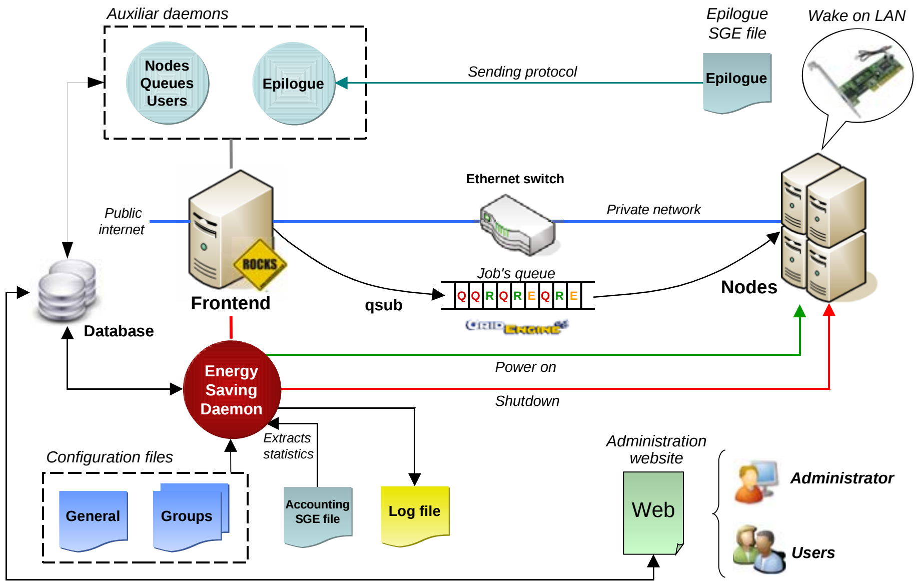


Fig. 1. Diagram of the energy saving module

Daemon for the queues, users and nodes. This daemon is responsible for ensuring that all information on users, nodes and queues that are actually in operation in the SGE queue system is correctly reflected by the database. To achieve this, the daemon queries the queue system about queues and nodes (with the `qhost` command), and the OS about users (checking the file `/etc/passwd`). With these data, it ensures that the database is consistent. The daemon is also in charge of enabling nodes that were marked as disabled.

Daemon for the activation/deactivation actions and statistics. This daemon, the most important of the module, activates and deactivates the nodes according to the needs queue system's user. The daemon compares the threshold parameters set by the system administrator and the current values of these parameters from the database to test if any of the activation or deactivation conditions is satisfied. If certain nodes have to be shut down, it suspends all queues to prevent the execution of new jobs, makes the transaction, updates the database and log file, and finally resumes the queues.

The threshold conditions are not necessarily equal for all users, as the system administrator can create multiple user groups with different values for the threshold parameters. Thus, a priority system can be defined for groups of users.

The daemon is divided into several functions, and the administrator can specify the order in which activation and deactivation conditions are checked. It is therefore very simple to enable or disable one or several of these conditions.

This daemon also updates the waiting time (both per user and per queue) of all enqueued jobs and ensures that the current database size does not exceed the specified maximum size.

3.2 Activation and Deactivation Conditions

Node activation. This operation is performed using the `ether-wake` command [18] which sends the magic packet WOL. Nodes can be turned on if any of the following conditions are met:

– *There are not enough appropriate active resources to run a job.* That is, as soon as the system detects that a job does not have enough resources, because all the nodes that contain the appropriate type of resource are turned off, nodes are powered on to serve the request.
– *The average waiting time of an enqueued job exceeds a given threshold.* The administrator must define a maximum average waiting time in queue for the jobs of each group. When the average waiting time of an enqueued job exceeds the maximum value assigned to the corresponding user's group, the system will turn on nodes which contain resources of the same type as those usually requested by the same user.
– *The number of enqueued jobs for a user exceeds the maximum value for its group.* In this case, the daemon selects and switches on nodes which feature the properties required by most of the enqueued jobs.

When the magic packet is sent, the daemon for activation/deactivation actions starts a timer. If this daemon does not detect that the node is active after the timer expires, the node is automatically marked as unavailable.

The system administrator can also use the following options to select the (candidate) nodes that will be activated:

– *Ordered*: The list of candidate nodes is sorted in alphabetical order using the name of the node (hostname).
– *Randomize*: The list of candidate nodes is sorted randomly.
– *Balanced*: The list of candidate nodes is sorted according to the period that the nodes were active during the last t hours (with t sets by the administrator). The nodes that are selected to be powered on are among those which have been inactive a longer period.
– *Prioritized*: The list of candidate nodes is ordered using a priority assigned by the system administrator. This priority can be defined, e.g., according to the location of the node with respect to the flow of cool air [19].

In the context of the SGE queue system, the slots of a queue instance for a given a node indicate the maximum number of jobs that can be executed concurrently in that node. When an exclusive execution is required (as, e.g., is usual in HPC clusters), the number of slots equals the number of processors. The daemon can also specify a strict threshold to power on nodes to serve job requirements:

– *No strict*: The nodes are turned on to serve job requests if there are not enough free slots on current active nodes. This option yields low queue waiting times but saves little energy.
– *Strict*: Nodes are only turned on when the current active nodes do not provide enough slots (free or occupied at the moment) to serve the requirements of the new job. This option produces longer queue waiting times than the previous policy but may provide fair energy saving.
– *Strict and sequential*: Nodes are only turned on to serve the job requests when all current active nodes have their slots in free state. This option simulates a sequential execution of currently enqueued jobs, likely delivering the longest queue waiting times and attaining high energy savings.

Node deactivation. Nodes are shut down using the `shutdown` command [20]. The following parameters define when a node is turned off:

– *The time that a node has been idle.* If this time is greater than a threshold set by the administrator, the node is turned off to save power.
– *The average time waiting for users' jobs is less than a threshold set by the administrator.* The administrator must define a minimum value for the queue waiting time of the jobs of each group of users. In case the average waiting time of a user's job is lower than the threshold assigned to its group, the daemon turns off a node (among those which exhibit the properties that were more rarely requested in the near past).
– *Current jobs can be served by a smaller number of active nodes.* The administrator can enable this condition to run the enqueued jobs using a smaller number of nodes than are switched on at a specified moment. In such case, the system turns off one of the nodes to save energy. Although this condition can significantly increase the average waiting time of the user's jobs, it may also reduce power consumption significantly.

When one of these three conditions is satisfied, the daemon executes the command "`shutdown -h now`" in a remote `ssh` session in the nodes that were selected to be shut down and suspends all associated queues to prevent the execution of new jobs on those nodes.

3.3 Database

The database contains the information needed to manage the system, as briefly described next.

The database stores information about each node of the cluster, like the node hostname, the time when the last job finishes its execution, the "active" bit (set if the administrator decides to maintain this node always turned on), the "unavailable" bit (for crashed nodes), the number of jobs executed on the node, and the assigned priority (only applicable if the selection node mode is "prioritized").

The database also maintains details of the queues (their name, number of enqueued jobs, and the average waiting and execution times of jobs that have been submitted to each queue) and the users (as, e.g., the name, the user's group, the number of jobs submitted to each queue by this user, and the average waiting and execution times of the jobs which have been submitted by this user).

Finally, the database records a history of the actions in each node (activations/deactivations), the action timestamp, and the cause (e.g., if the node was deactivated because a certain condition was satisfied).

3.4 Website Interface

The module has an interface that eases the administration of the energy saving module. Possible operations using this website include:

– Check and modify configuration parameters of the energy saving system.
– Look up, delete, and modify users' groups in the system.
– Look up and modify the parameters that define the activation/deactivation policies of each group of users.
– Read the full database in HTML format, with the possibility to generate reports (as PDF documents) with the required information.
– Monitor the operation of the cluster through a series of diagrams which illustrate the active/inactive node times, the average waiting time and execution time of jobs, etc.
– Monitor the energy savings in terms of power consumption and economic cost.
– Set "active" bits for those nodes that the administrator will always keep running, and declare unavailable nodes when they are crashed or being repaired.
– Turn on, reset or shut down the three daemons, and reset counters and timers of the database.

All changes done on the system configuration parameters are updated both in the database and the configuration files. Users without administrative privileges can only view reports, diagrams, and configuration parameters.

4 Experimental Results

To evaluate the benefits of the system we have developed a flexible simulator, named EnergySaving-SIM, that provides information on the system behavior for various platform configurations and under realistic workloads. EnergySaving-SIM applies the desired activation and deactivation policies to a given input workload and can be easily adapted to reflect different types of platforms. Among many other statistics, the simulator reports the percentage of the time that each node in the cluster will be turned on/off and, therefore, offers an estimation of the energy consumption.

4.1 Implementation of EnergySaving-SIM

The design of the EnergySaving-SIM consists of the following modules:

– *Queuing system:* This module simulates a queuing system and submits, executes, and terminates jobs in a specified cluster according to the availability of the nodes (active or inactive) from an input workload. This workload can be obtained from a parsed log file resulting from a real queuing system that contains the time that a job was submitted, the execution time and its requirements (such as specific nodes, number of processors, etc.). This module provides the appropriate interfaces to interact with the energy saving module and uses a database to store the data of the jobs.
– *Energy saving*: This module is the same used in the real system, but employs the interfaces provided by the queuing system module to check and query the status of jobs and make decisions to activate and deactivate nodes. By using the same code simulation we can obtain realistic simulations.
– *Results, statistics and graphs*: When the simulation finishes, all the submit, start and finish times of jobs and activation/deactivation actions of nodes with its specific time are stored in the database. This application extracts various statistics such as total active and inactive time per node, job's latency, etc. and prepares tables and graphs to evaluate energy consumption.

4.2 Simulation Results

We have configured the simulator to emulate the system of queues of the HPC computing service at the Universidad Jaime I (UJI). This facility is composed of the following nodes:

– **Front-end**: HP Proliant DL360 G5 with 2 dual core Intel Xeon 5160 processors, running at 3.00 GHz and with 14 GB of RAM.
– **Group 1**: 26 nodes, Fujitsu Siemens RX200 with 2 Intel Xeon processors running at 3.06 GHz and with 4 GB of RAM.
– **Group 2**: 27 nodes, HP Proliant DL360 G5 with 2 dual core Intel Xeon 5160 processors at 3.00 GHz, with 14 GB of RAM.
– **Group 3**: 11 nodes, HP Proliant BL460C with 2 Quadcore Intel Xeon E5450 processors at 3.00 GHz, with 32 GB of RAM.
– **Altix**: An SGI Altix 3700 server with 48 Itanium2 processors at 1.5 GHz, with 96 GB of RAM.

The job benchmark was obtained from the real queue system logs of the computing facility at UJI. This benchmark features the following properties:

– Composed by 10,415 jobs corresponding to the load submitted to the HPC computing facilities during three full months of 2009:
  99.87% of the jobs required one processor, 0.12% required four processors, and 0.01% required eight processors.

On the other hand, 73.3% of the jobs was executed on group 1, 0% requested group 2, 16.99% requested group 3, and 9.7% requested the Altix server.
– The average execution time of the jobs was 1 day, 2 hours and 53 minutes.

We evaluated the following policies:

– **No Policy (NP)**: This configuration represents a conventional cluster without the energy saving module on which nodes are permanently active.
– **Policy 1 (P1)**:
  - **Activation/deactivation conditions**:
    1. *Job without resources (Turn on).*
    2. *Idle time of a node (Turn off)*: Max. idle time: 60 seconds.
  - **Node selection algorithm**: Ordered.
  - **Strict level**: No strict.
– **Policy 2 (P2)**: This is the same configuration as **P1**, except for the *Strict level* which is changed from "No strict" to "Strict" level.
– **Policy 3 (P3)**: This is the same configuration as **P1**, except for the *Strict level* which is changed from "No strict" to "Strict and sequential" level.

Table 1 reports the following results obtained with the simulator:

– **Latency**: Average time since jobs are submitted till their execution is completed. This value thus includes the time a job is enqueued as well as its execution time.
– **Power on time (%)**: Average fraction of the total time that the nodes of the cluster remain turned on.
– **Total time**: Elapsed time since the first job is submitted till the last job completes its execution.
– **Total consumption**: Total consumption in MWatts-hour (MWh). (We have considered that a node consumes on average 250 Watts/hour.)

The results in the table show that the **NP** policy, where all the nodes are powered on all the time, yields an average response time (latency) per job over 339 h, and consumes consumes 65.37 MWh to execute the 10,415 jobs. The application of policy **P1** ("No strict" level) roughly increases the job latency to 461 h, but now the percentage of time that nodes are powered on is only 42.9%, and the power consumption is reduced to 29.51 MWh.

Table 1. Execution time and energy savings obtained for different policies

Policy	Latency	Power on time	Total time	Total consumption
NP	339 h, 44 m, 18 s	100.0%	4,022 h, 39 m, 50 s	65.37
P1	461 h, 54 m, 0 s	42.9%	4,022 h, 49 m, 15 s	29.51
P2	12,387 h, 56 m, 34 s	5.8%	29,962 h, 2 m, 41 s	46.50
P3	36,556 h, 28 m, 9 s	2.2%	86,712 h, 51 m, 31 s	85.73

Policy **P2** produces worse results than **P1**. As most of the jobs require a single processor, with the "Strict" level new nodes are not turned on until a job requests more processors. Thus, when most jobs require the same number of nodes, this is basically equivalent to a sequential execution. In particular, the job latency is increased to more than 12,387 h, the time the nodes are active is reduced to only 5.8%, and the time to complete the benchmark is now close to 30,000 h; moreover, the power consumption is increased to 46.50 MWh. In summary, this policy is not appropriate for this type of jobs, though it can be potentially interesting in other cases; e.g., when the batch of jobs requests very different number of processors.

Policy **P3** presents a long response time, and although the consumption is reduced to only 2.2%, the time to complete the benchmark raises to almost 86,713 h. Interestingly, the power consumption of this policy exceeds that of **NP**. Policy **P3** is more restrictive than **P2**, because it uses the "Strict and sequential" level, and therefore, no inactive nodes are powered on until all processors in turned on nodes are idle. For this particular benchmark, this policy is not appropriate. However, as was the case with policy **P2**, **P3** can deliver better best results in case the jobs request multiple processors. We will only discuss policy **P1** hereafter, as policies **P2** and **P3** were not competitive for this workload.

Table 2 reports more detailed results for **P1**. The first row in the table shows the number of node shut-downs happened in about 4,000 h: 206 shutdowns for a 65-node cluster is a reduced number. This means that, in average, a node was activated and deactivated slightly more than 3 times in 4,000 h. The largest number of active nodes at any given moment is 37 (out of 65), with the reason for this being that no job of the workload required nodes of group 2. The following rows show the active and inactive average time for all nodes in the cluster. To obtain these metrics, we collected the active and inactive time per node to calculate the total average. This metric illustrates the time that the nodes are powered on: basically 1,723 h of a total of 4,000 h, or 42.9% of the time. The last two rows of results in the table display the total active/inactive average time of nodes from the local averages of active and inactive intervals per node. Looking

Table 2. Detailed results for policy **P1**

Measure	**Total**	**Per node**
Number of shutdowns	206	3.17
Maximum active nodes	37 of 65	-
Minimum active nodes	1 of 65	-
Active time	112,056 h, 24 m, 28 s	1,723 h, 56 m, 41 s
Inactive time	149,424 h, 46 m, 47 s	2,298 h, 50 m, 33 s
Active time with average of active intervals per node	25,462 h, 29 m, 0 s	391 h, 43 m, 49 s
Inactive time with average of inactive intervals per node	120,678 h, 57 m, 53 s	1,856 h, 35 m, 58 s

at the inactive time with average of inactive intervals (roughly 1,856 h), we can conclude that nodes were down a considerable period of time for this particular workload. This high value indicates that nodes have been deactivated for long periods of time and, therefore, the decision of keeping them down is feasible. The result also demonstrates that, for this particular workload, it is more convenient to turn nodes off than to keep them active using, e.g., DVFS, as the time needed to reactivate a node is negligible compared with the period of time they remain inactive.

5 Summary and Conclusions

This paper describes the components of an energy saving module for the Sun® Grid Engine queue system and Rocks® Clusters operating system. By turning on only those nodes that are actually needed at a given time during the execution of a batch of jobs, the module may yield substantial energy savings.

The module is highly configurable: Specifically, a node can be turned on if a lack of resources for a particular job is detected, the average waiting time of the jobs is greater than a threshold, or the number of enqueued jobs exceeds a threshold. On the other hand, a node can be turned off if the idle time exceeds a threshold, the waiting time of enqueued jobs is lower than a threshold, or the current jobs can be served using a smaller number of nodes. In addition, there are also options to select candidate nodes to be powered on, and strict levels which can produce a considerable energy savings.

As expected, choosing the best policy depends on the type of jobs that are submitted to the system and the configuration of the cluster. Thus, the energy saving module is just a tool easily configurable but its performance will ultimately depend on the system administrator's expertise.

As future work we plan to adapt the module to the Portable Batch System (PBS), developing a Roll for the Rocks® system, and extend the module to extract real energy consumption on each node using *Watts Up?* power meters [21] attached at the private cluster network.

The energy saving system is currently in operation in the HPC clusters of the *High Performance Computing & Architectures* research group of UJI.

References

1. U.S. Environmental Protection. Report to congress on server and data center energy efficiency, public law 109-431. Agency ENERGY STAR Program (August 2007), `http://www.energystar.gov/ia/partners/prod_development/EPA_Datacenter_Report_Congress_Final1.pdf`
2. Kim, E.-J., Link, G., Yum, K.H., Narayanan, V., Kandemir, M., Irwin, M.J., Das, C.R.: A holistic approach to designing energy-efficient cluster interconnects. IEEE Trans. on Computers 54(6), 660–671 (2005)

3. Supermicro Computer Inc. Supermicro ofrece HPC de alta densidad con Microsoft Windows compute cluster server 2003. PR Newswire Europe Ltd. (March 2007), `http://www.prnewswire.co.uk/cgi/news/release?id=192954`
4. Liu, Y., Yang, H., Dick, R.P., Wang, H., Shang, L.: Thermal vs energy optimization for DVFS-enabled processors in embedded systems. In: International 8th Symposium on Quality Electronic Design (ISQED 2007), March 2007, pp. 204–209 (2007)
5. Ge, R., Feng, X., Cameron, K.W.: Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In: Conference on High Performance Networking and Computing Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p. 34 (2005)
6. Rountree, B., Lownenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.: Adagio: making DVS practical for complex HPC applications export. In: Proceedings of the 23rd international conference on Supercomputing (ICS 2009), pp. 460–469 (2009)
7. Xian, C., Lu, Y.-H., Li, Z.: Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In: Proc. 44th Annual Conf. Design Automation, San Diego, CA, USA, pp. 664–669. ACM, New York (2007)
8. Freeh, V.W., Pan, F., Lowenthal, D.K., Kappiah, N., Springer, R., Rountree, B.L., Femal, M.E.: Analyzing the energy-time tradeoff in high-performance computing applications. IEEE Transactions on Parallel and Distributed Systems 18(6), 835–848 (2007)
9. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centres. In: Proc. 18th ACM Symp. Operating System Principles, Banff, Canada, pp. 103–116 (2001)
10. Pinheiro, E., Bianchini, R., Carrera, E.V., Heath, T.: Load balancing and unbalancing for power and performance in cluster-based systems. In: Workshop on Compilers and Operating Systems for Low Power (September 2001)
11. Elnozahy, E.N., Kistler, M., Rajamony, R.: Energy-efficient server clusters. In: Workshop on Mobile Computing Systems and Applications (February 2002)
12. Rocks-solid: Extension to ROCKS cluster that make your cluster more solid!, `http://code.google.com/p/rocks-solid`
13. Grid Engine project. PowerSaving, `http://wiki.gridengine.info/wiki/index.php/PowerSaving`
14. University of California. Rocks® Clusters, `http://www.rocksclusters.org`
15. Inc. Sun Microsystems. Sun® Grid Engine, `http://gridengine.sunsource.net`
16. van Rossum, G.: Python Reference Manual, `http://docs.python.org/ref/ref.html`
17. Wikipedia. WakeOnLAN, `http://en.wikipedia.org/wiki/Wake-on-LAN`
18. Linux man page ether-wake, `http://linux.die.net/man/8/ether-wake`
19. Bash, C., Forman, G.: Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. HP Laboratories Palo Alto (August 2007)
20. Linux man page shutdown, `http://linux.die.net/man/8/shutdown`
21. Watts Up? Meters. Watts Up? .Net, `https://www.wattsupmeters.com`

Effect of the Degree of Neighborhood on Resource Discovery in Ad Hoc Grids

Tariq Abdullah[1], Koen Bertels[1], Luc Onana Alima[2], and Zubair Nawaz[1]

[1] Computer Engineering Laboratory, EEMCS, Delft University of Technology, Mekelweg 4, 2624 CD, Delft, The Netherlands
{m.t.abdullah,k.l.m.bertels,z.nawaz}@tudelft.nl
[2] LOA DEPENDABLE IT SYSTEMS S.A., 4, Rue d'Arlon, Luxemburg
luc.onana.alima@loa-dits.com

Abstract. Resource management is one of the important issues in the efficient use of grid computing, in general, and poses specific challenges in the context of ad hoc grids due to the heterogeneity, dynamism, and intermittent participation of participating nodes in the ad hoc grid. In this paper, we consider three different kinds of organizations in an ad hoc grid ranging from completely centralized to completely decentralized (P2P). On the basis of self organization mechanisms, we study the effect of the neighborhood degree of a node for finding resources on the efficiency of resource allocation. We investigate the message complexity of each organization and its corresponding efficiency in terms of task/resource matching and the response time. We show that the intermediate state of the ad hoc grid with multiple adaptive matchmakers outperforms both a completely centralized and a completely decentralized (P2P) infrastructure.

1 Introduction

Recent advances in personal computer processing power and Internet bandwidth has enabled achieving tremendous computing power via opportunistic resource sharing [1,2,3]. Opportunistic resource sharing is done in very dynamic environments where the addition of new nodes, system/network failures or variation in resource availability is expected. Therefore, in this context, resource management becomes one of the most important and complex part of grid middleware.

Resource discovery approaches for grids in general, and especially for ad hoc grids, can be categorized as completely centralized [1,3,4,5] and completely/ partially decentralized [6,7,8,9,10]. Generally, completely centralized resource discovery systems and peer-to-peer (P2P) systems are often considered to be mutually exclusive and residing on the two extremes of the infrastructural spectrum. In the GRAPPA project [11], we consider them to be a part of a continuum and study the effect of either of the extremes or any intermediate state between the two extremes using a micro-economic based resource discovery mechanism. This paper is based on our earlier work [12,13], where we presented the mechanisms and algorithms that enable the ad hoc grid to self-organize according to the workload of the ad hoc grid. In this paper, we look at the impact of adoption of a particular infrastructure, taken from the infrastructural continuum.

The contributions of this paper are as follow: First, we define the degree of neighborhood of a node for resource discovery in completely centralized, multiple adaptive matchmakers and in completely decentralized (P2P) environment in an ad hoc grid. Secondly, we analyze the effect of varying the degree of neighborhood in completely decentralized (P2P) ad hoc grid. Thirdly, we compare the results of varying the degree of neighborhood in completely decentralized approach with completely centralized approach and with multiple adaptive matchmakers approach. Fourthly, we perform the message complexity analysis of the above mentioned resource discovery approaches in order to understand the communication cost of a particular resource discovery approach. Finally, we give recommendations for trade offs in resource discovery on an infrastructural spectrum ranging from completely centralized to completely decentralized approaches in the ad hoc grids.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 describes the required background knowledge to understand the proposed model. Section 4 explains the proposed model. Section 5 provides message complexity analysis. The experimental setup and results discussion are presented in Section 6, While section 7 concludes the paper and briefs about the future work.

2 Related Work

Different approaches are used for resource discovery in the ad hoc grids. These approaches vary from completely centralized to completely decentralized ones. The completely centralized approaches [1,2,3,5] for the ad hoc grids employ a client-server architecture. A trusted server distributes the jobs to clients. The clients request jobs, the centralized server allocates the jobs to the clients, the clients run the jobs, and the server collects the results. The completely centralized approaches provide high throughput. However, robustness and reliability is maintained by the server. Furthermore, the above mentioned approaches have a single point of failure and the complete system becomes unavailable in case of network or server failure.

In completely/semi decentralized approaches, each node or group of nodes negotiates for its required resources with other nodes. Iamnitchi et al. [8] proposed a resource discovery approach in completely decentralized grid environments and evaluated different request forwarding algorithms. Their approach employs time to live (TTL) for resource discovery. TTL represents the maximum hop count for forwarding a request to the neighboring nodes. The TTL approach is simple but may fail to find a resource, even though that resource exists somewhere in the grid. Attribute encoding [6,7] is used for resource discovery in structured overlay network. The available resources are mapped to the nodes of a P2P structured overlay network in the attribute encoding approach. There can be a load imbalance due to attribute encoding, when the majority of encoded attributes are mapped to a small set of nodes in the overlay network.

A zone based hybrid resource/service discovery approach using Zone Routing Protocol is presented in [9]. This work is closely related to our work. The main differences from our work are the use of micro-economic approach for resource discovery and the extension of a structured overlay network [12] for ad hoc segmentation/desegmentation. The reasons for using a micro-economic approach for resource discovery in ad hoc grid are described in Section 3.1. Zhou et al. [10] exploited blocks of idle processing cycles and grouped them into geographic and night time aware overlay networks. Unfinished tasks are migrated to another night time zone when the current night time zone ends. The main drawback of this work is that the host availability model is not based on the resource requirements of a job.

This paper defines and implements a micro-economic based resource discovery approach with varying the degree of neighborhood of nodes in an ad hoc grid. Secondly, the paper analyzes the effect of the degree of neighborhood on resource discovery. Thirdly, the results are compared with the completely centralized approach and with multiple adaptive matchmakers approaches for resource discovery. Finally, the paper provides recommendations to define trade-offs for a micro-economic based resource discovery mechanism on an infrastructural spectrum ranging from the completely centralized to the completely decentralized environments.

3 Background Knowledge

Before presenting the proposed model, first we explain the necessary concepts needed to understand the proposed model and the experimental results.

3.1 Micro-economic Based Resource Discovery

An overview of Continuous Double Auction (CDA) based resource discovery mechanism is provided in this section. CDA is one of the many-to-many auctions in micro-economic theory. CDA supports simultaneous participation of producer/consumer, observes resource offer/request deadlines and can accommodate variations in resource availability.

Our ad hoc grid consists of autonomous nodes. Each node has *resource consumer*, *resource producer* and *matchmaker* agents. A node can be a consumer/producer of resources (such as CPU, memory, disk space or bandwidth) and/or a matchmaker at the same time. A *producer node* offers its available resources (such as CPU, memory, disk space or bandwidth). A *consumer node* requests the desired resources in order to execute its jobs. The node playing the role of a mediator between the consumer and the producer nodes is named the resource allocator or a *matchmaker* in this work. These three kinds of agents are also three main participants in CDA based resource discovery mechanism. The resource provider agent submits resource offer (called *ask*) and the resource consumer submits resource request (called *bid*) to the matchmaker agent. A resource request (bid) is specified by number of constraints such as requested resource quantity, job execution duration, job validity period (denoted by Time to Live (TTL) and represents the time duration during which a request can be processed), and bid price. Similarly, a resource offer (ask) is also specified by a number of parameters such as offered resource quantity, offer validity period (TTL, represents the time duration during which the offer can be availed), and ask price.

The matchmaker stores all received bids/asks in its request/offer repositories. The matchmaker is responsible for finding the *matched bid/ask pairs* from received bids and asks of the consumer and producer agents respectively. A matched bid/ask pair represents a pair where the resource request constraints are satisfied by the matching resource offer. The matchmaker finds the matches between the consumers and producers by matching asks (starting with lowest price and moving up) with bids (starting with highest price and moving down). The matchmaker searches all available asks (resource offers), for finding a matched bid/ask pair, on receiving a bid (resource request). A bid/ask is stored in the matchmaker repository until a match is found or its TTL is expired. The details of CDA based matchmaking mechanism and ask/bid price calculation formula can be found in [14].

3.2 Resource Discovery with Multiple Adaptive Matchmakers

In multiple adaptive matchmakers resource discovery approach in an ad hoc grid, a new matchmaker(s) is introduced or removed according to the workload of the matchmaker [12]. There can be n nodes in our experiments. There can be a maximum of m ($m < n$), out of n nodes, matchmakers in the multiple adaptive matchmakers approach.

Each joining consumer/producer/matchmaker node is provided a zone number to which the node belongs. The whole identifier space is divided into zones. Each zone has a responsible matchmaker. It is ensured that each consumer/ producer node is under the responsibility of a matchmaker. When a matchmaker becomes overloaded then it promotes its predecessor matchmaker node to perform matchmaking. The consumer/producer nodes under the responsibility of an overloaded matchmaker are now under the responsibility of the predecessor matchmaker. In the case that the predecessor matchmaker is already performing matchmaking (i.e. active) then the excess workload is forwarded to the successor matchmaker of the overloaded matchmaker.

Conversely, when a matchmaker is underloaded then it demotes itself and informs its predecessor and successor matchmakers about the change in its matchmaking status. The successor matchmaker of the demoted matchmaker becomes the responsible matchmaker for consumer/producer nodes that were previously under the responsibility of the demoted matchmaker. After demoting itself, the demoted matchmaker will forward the request/offer messages to its successor matchmaker. The demoted matchmaker also informs the consumer/producer node under its responsibility, about its matchmaking status change and about the new matchmaker.

A consumer/producer node finds its responsible matchmaker node with the provided information after joining the ad hoc grid. In case there is only one matchmaker in the ad hoc grid then it becomes the responsible matchmaker for all the consumer/producer nodes. The consumer/producer node can submit request/offer to the matchmaker node after finding the responsible matchmaker node. Each matchmaker node maintains matchmaking status information (active/inactive) about its predecessor and successor matchmaker nodes, after joining. The matchmaker does so by exchanging matchmaking status information with its successor and predecessor nodes.

4 The Neighborhood on the Infrastructural Continuum

In this section, we explain the degree of neighborhood of a node on the following points of an infrastructural spectrum that ranges from completely centralized to completely decentralized extremes.

In order to explore the difference in resource allocation efficiency between the completely centralized and the completely decentralized (P2P) approaches, we introduce the notion of neighborhood. The degree of neighborhood of a node defines the visibility region of a node by defining the number of nodes accessible from that node. We explain the degree of neighborhood of node on the following points on an infrastructural spectrum:

- Completely Centralized Approach
- Multiple Adaptive Matchmakers Approach
- Completely Decentralized (P2P) Approach

In the **completely centralized approach,** with a single matchmaker, all consumer/producer nodes (see Section-3.1) send their resource requests or resource

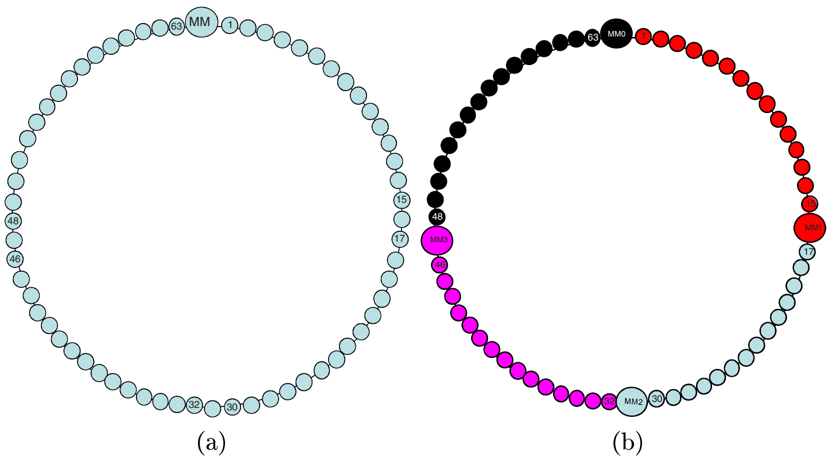


Fig. 1. Neighborhood on the infrastructural spectrum. **(a)** Completely centralized. **(b)** Multiple adaptive matchmakers.

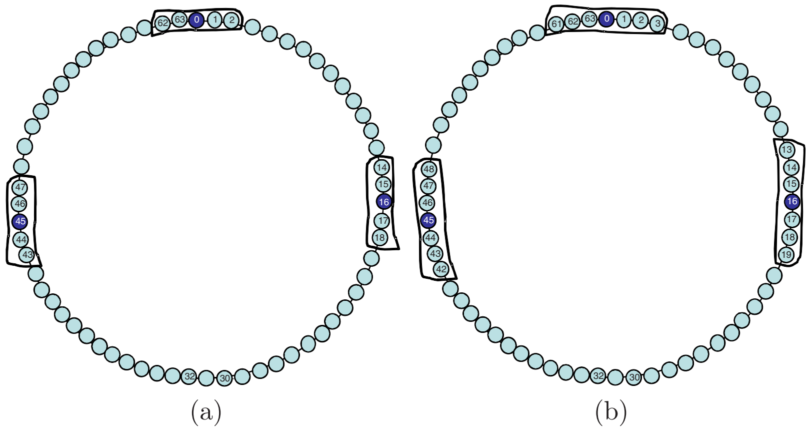


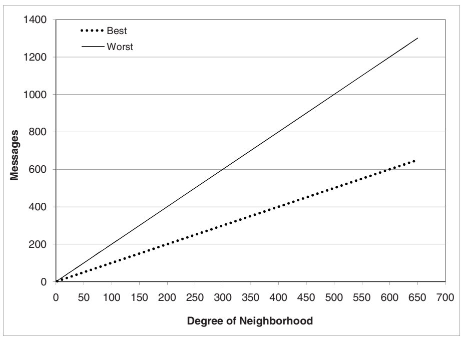
Fig. 2. Neighborhood on the infrastructural spectrum. **(a)** Completely decentralized degree=4. **(b)** Completely decentralized degree=6.

offers to the matchmaker. The matchmaker finds matches for resource requests from received resource offers and informs the matched consumer/producer nodes. As all participating consumer/producer nodes can send their request/offer message to the matchmaker only, therefore the neighborhood of a consumer/producer node is n (n being the total number of the nodes in the ad hoc grid). This is represented in Figure-1a, where there is only one matchmaker.

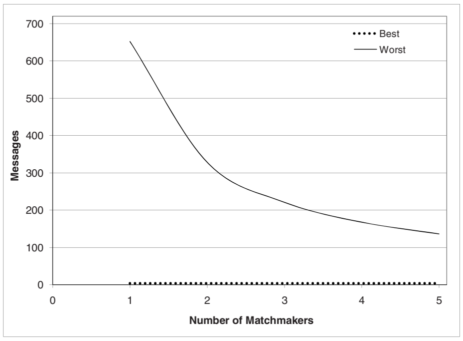
In the **multiple adaptive matchmakers approach**, an intermediate centralized approach using multiple adaptive matchmakers, each consumer/producer node is under the responsibility of one matchmaker at any given point in time. The matchmaker is demoted or promoted according to the workload of the matchmaker(s) in the ad hoc grid. Then number of matchmaker(s) and the responsible matchmaker of a consumer/producer node may also change by the promotion/demotion of the matchmaker(s) [12]. As each consumer/producer node is under the responsibility of only one matchmaker at any given point in time, therefore the neighborhood of a consumer/producer nodes is $^{n}/_{m}$ (n being the total number of the participating nodes and m being the number of matchmakers). This is represented in Figure 1b, where multiple matchmakers are represented with different colors and the consumer/producer nodes in each zone are represented by the color of their responsible matchmaker.

In the **completely decentralized (P2P) approach**, where every node is its own matchmaker, each node looks for the appropriate resources from all the nodes in its degree of neighborhood. The ad hoc grid is implemented on top of Pastry [15], a structured P2P overlay network. The degree of neighborhood of a node is implemented and varied with the help of Pastry node's leaf set [15] in our ad hoc grid, which is explained below.

We consider a Pastry node with nodeID x for explaining the degree of neighborhood in the ad hoc grid. Each node in Pastry is assigned a 128 bits unique



(a)



(b)

Fig. 3. (a) Number of messages exchanged when varying the degree of neighborhood in completely decentralized (P2P) approach. **(b)** Number of messages exchanged in centralized and in multiple adaptive matchmakers approach.

node identifier (referred to as *nodeID* hereafter). A Pastry node's leaf set contains L closest nodeIDs to the nodeID x. The leaf set, L, comprises of $|L|/2$ numerically closest larger nodeIDs and $|L|/2$ numerically closest smaller nodeIDs, relative to any node's nodeID in a Pastry overlay network. Here $|L|$ represents the cardinality of the leaf set L. The visibility of a node in the ad hoc grid increases with an increase in its degree of neighborhood. The neighborhood degree 4 and 6 of different arbitrary nodes (with nodeIDs 0, 16 and 45) in a completely decentralized ad hoc grid are represented in Figures 2a and in 2b respectively.

Typically, a Pastry node can route a message to another Pastry node in less than $\log_{2^b} N$ steps [15]. A Pastry node directly sends a message to its leaf set members. As the neighborhood is implemented as the leaf set, therefore, all the message exchange between our ad hoc grid nodes take only *one* hop instead of $\log_{2^b} N$ hops.

5 Message Complexity Analysis for Finding a Match

It is important to understand the cost implications of a particular organization of the ad hoc grid. To this purpose, we analyze the number of messages exchanged for finding a matched pair in the completely centralized, multiple adaptive matchmakers and in completely decentralized (P2P) resource discovery approaches.

First, we analyze the **completely centralized approach**. Let n be the total number of participating nodes. These nodes can play the role of a consumer or a producer at any given time. There is only one matchmaker in the centralized resource discovery approach. In the best case, a consumer node sends a request to the matchmaker and a producer node sends a resource offer to the matchmaker. The matchmaker finds a match and a reply message is sent to the consumer and producer nodes. In the worst case, $n - 1$ nodes will send their offers to

Table 1. Messages exchanged to find a match

	Best Case	Worst Case
Total Centralized (One Matchmaker)	4	$n + 2$
Multiple Matchmakers	4	$m + n_i + 1$
Varying the degree of Neighborhood	$d + 1$	$2d + 1$

the matchmaker. Only then the matchmaker can find a suitable offer for the received request and a matched message is sent to both matching consumer and producer nodes. Hence, only 4 messages are required in the best case and $n + 2$ messages are required in the worst case to find a matched request/offer pair in the centralized resource discovery approach.

In case of the **multiple adaptive matchmakers approach**, each matchmaker is responsible for certain number of nodes out of all the participating nodes. An overloaded matchmaker forwards its excess workload to its neighboring matchmaker. The details of matchmaker(s) promotion/demotion and excess workload forwarding are discussed in [12]. Let n be the total number of participating nodes, m be the number of matchmakers, where $m < n$, and n_i be the number of nodes under the responsibility of matchmaker m_i, where $i = 1, 2, 3, ..., m$, in the ad hoc grid, such that: $n = \sum_{i=1}^{m} n_i$.

The best case for a matchmaker in the multiple adaptive matchmaker approach is the same as that of the centralized approach. However, in the worst case, a request/offer message may be forwarded to at most $m - 1$ matchmakers [12]. Therefore, the maximum number of messages to find a match will be $(m - 1) + n_i + 2$, where n_i is the number of nodes under the responsibility of $(m - 1)^{th}$ matchmaker and 2 represents the matched message sent to both matched consumer and producer.

For the **completely decentralized (P2P) approach** with varying the degree of neighborhood**,** let n be the total number of nodes and d be the degree of neighborhood, such that d = 2, 4, 6, 8, ..., n in the ad hoc grid. In the best case, all the neighboring nodes will send offers to the current node, for its resource request, and one matched message will be sent to the matching producer node. Hence, the number of messages will be $d + 1$. The worst case scenario of this protocol, varying the degree of neighborhood, was explained in the start of this section. A node will send its resource request/offer to all neighboring nodes, and all neighboring nodes will send a resource offer/request to the sender node. The sender node will send a confirmation message to the selected producer/consumer node. Total number of exchanged messages to find a matched pair will be $2d + 1$.

It is important to point out that the differentiating point in the analyzed resource discovery approaches, is the matchmaker's ability to search for a required resource from the nodes under its responsibility or in its degree of neighborhood. The matchmaker agent can look at the submitted offers of the nodes under its responsibility in the completely centralized and in multiple adaptive matchmakers approach. The matchmaker agent is limited by the degree of neighborhood (except when $d = n$) and cannot search the resources of all participating nodes. Figure 3a and 3b compare the number of messages required to find a match in the varying the degree of neighborhood approach with centralized and multiple adaptive matchmakers approach, respectively, in the ad hoc grid.

Although the distribution of ad hoc grid nodes among two or more matchmakers vary according to the workload of the matchmakers [12], we assume $n_i = {}^{n}/_{m}$ for the case of multiple adaptive matchmakers, while comparing the message complexity for varying the degree of neighborhood in completely decentralized approach with other two approaches in Figure 3a and 3b. $1MM$ represents one matchmaker of the centralized approach, whereas $2MM, ..., 5MM$ represent two or multiple matchmakers of the multiple adaptive matchmakers approach in Figure 3b. The number of messages exchanged in different resource discovery approaches are summarized in Table 1.

6 Experimental Setup and Results

We developed our ad hoc grid experimental platform on top of Pastry [15]. Although we used Pastry, in principle any other structured overlay network can be used. Pastry is a self-organizing and adaptive overlay network. Pastry is used for node arrival/departure, node failure handling, and for message routing in this work. Node join/leave and Pastry message routing is explained in [15].

The experiments are executed on PlanetLab [16]. PlanetLab is a global, community-based effort and is used mostly for network related experiments. The PlanetLab nodes are connected through the Internet. Research institutions/ organizations contribute a minimum of 2 computing machines. The researchers of the corresponding institute/organization are granted access to a pool of more than 1000 PlanetLab nodes.

The experiments are executed to answer the questions discussed in Section 1. The first set of experiments are executed to analyze the effect of varying the degree of neighborhood. In the second set of experiments, the experimental results with varying the degree of neighborhood are compared with total centralized approach and with multiple adaptive matchmakers approach for resource discovery in an ad hoc grid.

The number of participating nodes varies from 15 to 650. The number of matchmakers varies from 1 to 5 in the experiments with multiple adaptive matchmakers. TTL of the request/offer messages is set to 10000 milliseconds in order to cater the delays observed in PlanetLab. In this work, we have only considered computational power (CPU cycle) as a resource. However, other resources like memory, bandwidth and disk storage can also be incorporated in this model. The job execution time, job deadline, budget, and request/offer computational resource amount are randomly generated from a predefined range. The request/offer resource quantity varies for each request/offer message. Data presented is obtained after the system reaches a steady state, when ${}^{1}/_{4}th$ of the experiment time is elapsed.

Matchmaking efficiency, response time and the message complexity are analyzed in these experiments. Message complexity analysis is explained in Section 5. The matchmaking efficiency in time interval $T = [T_{start}, T_{end}]$ is defined as:

$$\left(\sum_{T_{start}}^{T_{end}} Matched\ Message \Big/ \sum_{T_{start}}^{T_{end}} Total\ Message\right) * 100$$

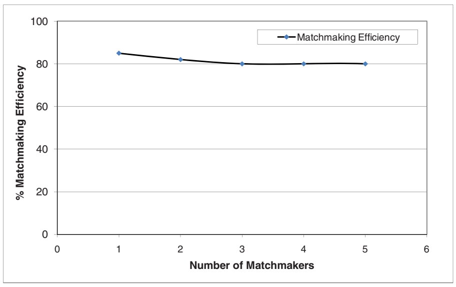
Where T_{start} and T_{end} represent the start and end time of the time interval $T = [T_{start}, T_{end}]$. The response time denotes the time interval, starting from the time a message is received, and ends at the moment when a match is found for the received message. The response time is calculated as: $RT = T_{match} - T_{receive}$, where RT represents the response time, T_{match} is the time when the matchmaker agent found a matching offer/request for the received request/offer message and $T_{receive}$ is the receiving time of the received request/offer message.

All the experiments are executed in different network conditions, including task intensive ($tasks >> resources$), balanced ($tasks \approx resources$) and resource intensive ($tasks << resources$) network conditions. The task intensive network condition in our experiments is the case when approximately 80% of the participating nodes act as resource consumers and 20% as resource producers. The consumer-to-producer ratio is $50\% - 50\%$ in the balanced network condition and the consumer-to-producer ratio is $20\% - 80\%$ in resource intensive network condition. The experimental results of balanced network condition are presented and explained in the next section.

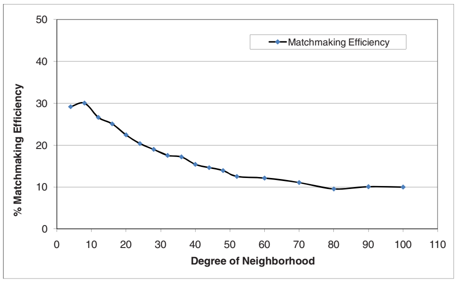
6.1 Experimental Results

First, we look at the matchmaking efficiency in the completely centralized resource discovery approach (number of matchmaker as 1 in Figure 4a) and with multiple adaptive matchmakers resource discovery approach (number of matchmakers > 1 in Figure 4a). The **completely centralized approach** shows higher matchmaking efficiency for small workloads. However, one matchmaker cannot maintain its matchmaking efficiency with the increasing work load. The matchmaking efficiency keeps on decreasing with increasing work load of the matchmaker. This phenomenon can be understood with the following explanation. With the increasing workload, the matchmaker has to process more messages, so it takes more time to find matched pairs. This results in an increased response time of the matchmaker. Since each request/offer message has a validity period (TTL), therefore the TTL of the request/offer messages start expiring with increased processing time of the matchmaker and consequently the matchmaking efficiency of the matchmaker decreases with increasing workload of the matchmaker. The work load threshold for one matchmaker system, decreasing matchmaking efficiency with increasing workload of a matchmaker are explained in our earlier work [13].

The matchmaking efficiency of **multiple adaptive matchmakers approach** is not affected by the increasing workload. The adaptive mechanism introduces



(a)



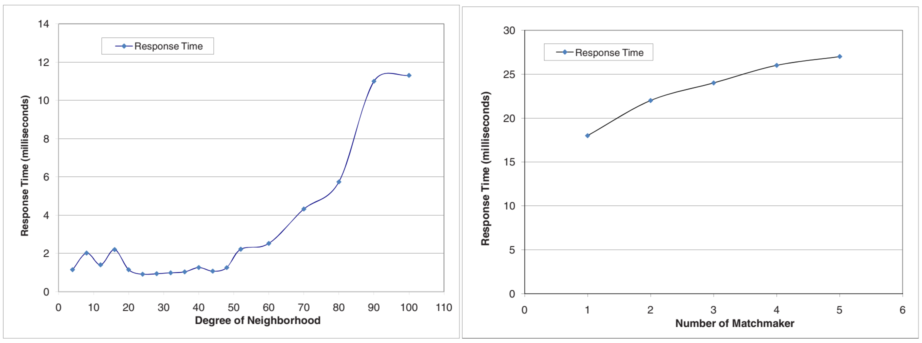
(b)

Fig. 4. (a) Matchmaking efficiency of centralized and multiple adaptive matchmakers approach. **(b)** Matchmaking efficiency with varying the degree of neighborhood in completely decentralized (P2P).

more matchmaker(s) when needed by an overloaded matchmaker(s). Hence, the matchmaking efficiency remains the same with the increased number of matchmakers. The matchmaking efficiency of completely centralized resource discovery approach is slightly higher than that of multiple adaptive matchmakers approach (Figure 4a). The matchmakers in multiple adaptive matchmakers approach communicate with other matchmakers in order to promote/demote matchmakers and for sharing their access workload with the other matchmakers [12]. Some of the request/offer messages expire during this process. Since, there is no communication or work load sharing with other matchmakers in the completely centralized approach, the maximum matchmaking efficiency of the completely centralized system is slightly higher than that of the multiple adaptive matchmakers system. However, the completely centralized approach is not scalable and can have a single point of failure [12,13].

Figure 4b shows the matchmaking efficiency of the resource discovery approach with varying the degree of neighborhood in a **completely decentralized (P2P) ad hoc grid**. The matchmaking efficiency initially increases with an increased degree of neighborhood. This seems logical as with an increased degree of neighborhood, the chances for finding a required resource/offer also increase. However, this trend starts decreasing with further increase in the degree of neighborhood due to the increased number of request/offer messages (refer to Figure 3a). The matchmaker agent of each node has to process more messages. The increased processing time results in TTL expiry of request/offer messages and consequently a drop in the matchmaking efficiency. The experiments were repeated for $n = 100$, 250 and d was varied from $d = 2, 4, 6, 8, ..., n$. The same matchmaking pattern as in Figure 4b was observed.

An alternative view of the above discussed phenomenon is to consider the average response time. Figure 5a shows the average response time to find a match with varying the degree of neighborhood in a **completely decentralized (P2P) ad hoc grid**. In our experiments, the response time stays stable up to 50 hops in the P2P case. The response time increases more than proportional



(a) (b)

Fig. 5. (a) Response time with varying the degree of neighborhood in the completely decentralized (P2P) approach. **(b)** Response time of centralized and multiple adaptive matchmakers approach.

once the number of hops goes beyond 60 (Figure 5a). The over proportional increase in response time is due to the communication overhead incurred with the increased degree of neighborhood in the completely decentralized (P2P) ad hoc gird.

We observe an increase in the response time of **multiple adaptive matchmakers approach** with increased number of matchmakers (Figure 5b). This increase is due to the segmentation of the ad hoc grid and due to increased communication as explained in Section 5. The experiments were also executed under resource intensive and task intensive network conditions. We observed the same trend of the matchmaking efficiency and response time as discussed above.

It can be concluded from the above discussion that neither a completely centralized nor a completely decentralized (P2P) is generally a suitable infrastructures for resource discovery in an ad hoc grid. A completely centralized infrastructure is not scalable and can have a single point of failure. On the other hand, a completely decentralized (P2P) infrastructure incurs excessive communication overhead that results in an increased response time and decreased matchmaking efficiency. An intermediate infrastructure having multiple adaptive matchmakers seems most efficient in terms of response time and in finding matches. The intermediate infrastructure with multiple adaptive matchmakers should be preferred whenever possible in the ad hoc grid.

7 Conclusions

In this paper, we analyzed the effect of varying the degree of neighborhood on resource discovery in a local ad hoc grid. For this purpose we defined and implemented the degree of neighborhood for participating nodes. Results were obtained for completely centralized, multiple adaptive matchmakers and for completely decentralized resource discovery approaches in an ad hoc grid. Results show that the ad hoc grid becomes less efficient with increased degree of neighborhood in completely decentralized approach, due to the excessive messages being exchanged. The results also confirmed that an intermediate ad hoc grid infrastructure with multiple adaptive matchmakers is preferable in a local ad hoc grid. In future, we will investigate the resource discovery approaches in hybrid environments for multiple adaptive matchmakers approach, where both the centralized matchmaking and P2P matchmaking will occur.

References

1. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: 5th IEEE/ACM International Workshop on Grid Computing (2004)
2. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: Architecture and performance of an enterprise desktop grid system. JPDC 63(5), 597–610 (2003)
3. Germain, C., Néri, V., Fedak, G., Cappello, F.: XtremWeb: Building an experimental platform for global computing. In: Buyya, R., Baker, M. (eds.) GRID 2000. LNCS, vol. 1971, pp. 91–101. Springer, Heidelberg (2000)
4. Chien, A., Marlin, S., Elbert, S.T.: Resource management in the entropia system, pp. 431–450. Kluwer Academic Publishers, Dordrecht (2004)
5. Patil, A., Power, D.A., Morrison, J.P.: Economy-based computing with webcom. Journal of Computer and Information Science and Engineering 1(2), 82–89 (2007)
6. Cheema, A.S., Muhammad, M., Gupta, I.: Peer-to-peer discovery of computational resources for grid applications. In: Proceddings of GRID 2005 (2005)
7. Gupta, R., Sekhri, V., Somani, A.K.: CompuP2P: An architecture for internet computing using peer-to-peer networks. IEEE Transactions on Parallel and Distributed Systems 17(11), 1306–1320 (2006)
8. Iamnitchi, A., Foster, I.: On fully decentralized resource discovery in grid environments. In: Lee, C.A. (ed.) GRID 2001. LNCS, vol. 2242, pp. 51–62. Springer, Heidelberg (2001)
9. Moreno-Vozmediano, R.: A hybrid mechanism for resource/service discovery in ad-hoc grids. Future Generation Computer Systems (2008) (article in press)
10. Zhou, D., Lo, V.: Wavegrid: A scalable fast-turnaround heterogeneous peer-based desktop grid system. In: Proceedings of IPDPS 2006 (2006)
11. GRAPPA Project, `http://ce.et.tudelft.nl/grappa/`
12. Abdullah, T., Alima, L.O., Sokolov, V., Calomme, D., Bertels, K.: Hybrid resource discovery mechanism in ad hoc grid using strucutred overlay. In: ARCS 2009 (2009)
13. Abdullah, T., Sokolov, V., Pourebrahimi, B., Bertels, K.: Self-organizing dynamic ad hoc grids. In: SASO 2008 Workshops (2008)
14. Pourebrahimi, B., Bertels, K., Kandru, G., Vassiliadis, S.: Market-based resource allocation in grids. In: Proceedings of e-Science 2006 (2006)
15. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
16. PlanetLab Online, `https://www.planet-lab.org/`

Compiler-Directed Performance Model Construction for Parallel Programs

Martin Schindewolf[1], David Kramer[1], and Marcelo Cintra[2]

[1] Institute of Computer Science & Engineering
Karlsruhe Institute of Technology (KIT)
Haid-und-Neu-Straße 7
76131 Karlsruhe, Germany
{schindewolf,kramer}@kit.edu
[2] School of Informatics
University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB, United Kingdom
mc@inf.ed.ac.uk

Abstract. During the last decade, performance prediction for industrial and scientific workloads on massively parallel high-performance computing systems has been and still is an active research area. Due to the complexity of applications, the approach to deriving an analytical performance model from current workloads becomes increasingly challenging: automatically generated models often suffer from inaccurate performance prediction; manually constructed analytical models show better prediction, but are very labor-intensive. Our approach aims at closing the gap between compiler-supported automatic model construction and the manual analytical modeling of workloads. Commonly, performance-counter values are used to validate the model, so that prediction errors can be determined and quantified. Instead of manually instrumenting the executable for accessing performance counters, we modified the GCC compiler to insert calls to run-time system functions. Added compiler options enable the user to control the instrumentation process. Subsequently, the instrumentation focuses on frequently executed code parts. Similar to established frameworks, a run-time system is used to track the application behavior: traces are generated at run-time, enabling the construction of architecture independent models (using quadratic programming) and, thus, the prediction of larger workloads. In this paper, we introduce our framework and demonstrate its applicability to benchmarks as well as real world numerical workloads. The experiments reveal an average error rate of 9% for the prediction of larger workloads.

1 Introduction

For many years, research in computer system architecture has aimed at modeling the performance of computer architectures and workloads. Performance modeling is widely applicable during a computing life-cycle: design, integration, installation, and tuning use results derived from performance models. Although the models deliver great benefits, the construction of performance models mostly is a manual and time-consuming task [1,2,3]. Therefore, the semi-automation of the model construction process for workloads is an alleviation for the modeler and, hence, a major contribution of this work. This paper presents a model construction process directed by compiler information derived from static analysis of the source code. The compiler recognizes frequently executed parts of an application and instruments these. This helps to focus the modelling process as only selected parts of the code are instrumented. The instrumented executable calls a library tracing the application execution and collecting hardware performance counter statistics. The gathered performance counter data is used for the construction and refinement of the performance model. The validation against this empirical data is necessary for improving the quality of the predicted parameters and to allow for incremental refinements to the model.

The paper contributes the first enhancement of an open source compiler with performance modeling capabilities (GCC-PME), demonstrates the seamless integration of the GCC-PME into a flexible and extensible set of post-processing tools, and presents results derived from modeling benchmarks and a numerical workload in order to demonstrate the applicability of the tool chain.

The presented results can be viewed as a first step towards compiler-based model construction. Previous approaches rely on hand-instrumentation of applications to derive and construct workload models such as the modeling assertions approach [4]. Manually instrumenting the application with calls to a run time system, requires detailed knowledge of the application. Involving the compiler to instrument the application has two benefits: first, the user does not need any application-level knowledge which makes the process amenable for literally everyone. Second, the compiler instruments the executable on the user's behalf. Thus, the user still has control over the instrumentation process and may add complementary instrumentation if this is desired.

A survey of related work is found in Section 2. Design and implementation of the proposed approach are presented in Section 3. Starting to describe the compiler enhancements, this section also contains a description of the runtime library, an introduction to the applied models, and the post-processing tools. Section 4 shows models from the NAS benchmarks and real world numerical workloads. This work concludes in Section 5.

2 Related Work

This section gives an overview of the related work in the area of performance prediction for parallel applications.

The Modeling Assertions approach (MA) towards model construction was introduced by Alam and Vetter at the Oak Ridge National Laboratory [4,5]. Their framework consists of parts for construction, validation and refinement of a performance model based on the application input parameters. The symbolic representation enables efficient exploration of the application's parameter space.

The approach is based on the manual instrumentation of the application with MA API calls. The models for floating point and load/store operations are created at run-time of the application. The used MA library starts and stops the performance counters (using PAPI), which are used to validate the model. The MPI communication patterns are profiled. Further, an incremental refinement of the model is supported through adding more variables to the input parameters. The MA library functions generate trace files. These traces are used to validate the model. The CG and SP benchmarks of the NPB-suite were evaluated and reveal a maximum error rate less than 30% (typical error rate less than 10%) for floating point operations.

Ipek et. al. propose a neural network based approach towards performance prediction of parallel applications [6]. Through training neural networks on performance data, this approach benefits from automated model construction, modeling full system complexity without the need to add architectural details to the model to get precise predictions.

Marin describes an architecture independent model construction process [7]. The object code of an application is analyzed in two ways. First, a static analysis is performed, identifying the loop nests, instruction mixes in basic blocks and delivering a control flow graph. Second, the binary is instrumented to measure the basic block counts, the communication volume and frequency and the memory utilization at runtime. The post-processing tool set generates an architecture neutral model, which serves as input for the scheduler and is merged with an architecture description leading to an overall performance model. The scheduler maps the specific instructions to generic classes, assuring the architecture independence. The resulting models are capable of predicting floating point, load and store operations. This approach is extended for cross-platform prediction [8]. The effects of restricting the execution time on cross-platform performance prediction are studied by Yang et. al [9].

The convolution method was developed at the Performance Modeling and Characterization Lab at the San Diego Supercomputer Center [10]. The performance prediction of a parallel application is obtained from single processor performance and network utilization.

Lee et al. propose to apply statistical methods to combine single processor and contention models to predict multiprocessor behavior [11,12].

The Performance Oriented End-to-end Modeling System (POEMS) is an environment for end-to-end performance modeling of parallel systems [13]. System components are modeled on different layers of abstraction: applications, runtime, operating system, and hardware. Different modeling paradigms, such as simulation, analysis, and direct measurement, are employed to model the components. Sweep3D is used to evaluate and predict parallel architectures.

TAU is a set of tools addressing instrumentation, measurement and analysis of parallel applications [14]. However, their approach does not involve a compiler but instead relies on the combination of a preprocessor and binary instrumentation or dynamic instrumentation to realize an instrumented binary. Binary instrumentation has the advantage to work even in the absence of the source code.

On the other hand application level knowledge that is still available at compile time (names of loop variables, trip counts, origin of code fragments (source file and line), etc.) may be lost at binary level or is difficult to reconstruct.

Compared with the surveyed related work based on simulation, statistical methods or incorporating an array of different models, our approach contributes compiler-directed models. In contrast to the complex model construction process favored in the related work, we restrict ourselves to present simplistic models based on quadratic programming to show the validity of the compiler-based approach. Our approach is expected to integrate with the modeling assertions approach and, thus, may greatly simplify the construction of more sophisticated performance models (e.g. symbolic models) of unknown workloads.

3 Framework

The following section describes the implementation aspects of our work. Figure 1 shows the tool chain and the section is organized top-down. First, a description of the compiler passes is given. Afterwards, this section describes the implementation of the MA library sketching the actions taken at run-time. At last, the post-processing tools are covered, addressing the tool to validate predictions against information retrieved from performance counters as well as aspects of the model construction.

3.1 GCC-PME

This section shows the basic design decisions for the implementation of the compiler extension. The "PME" extension to the name GCC underlines the program modeling enhancements to the tree structure of GCC (version 4.2.0). Since the user needs some means to influence the modeling process, new compiler options are introduced. These options influence the instrumentation process so that only selected parts are instrumented. These parts are determined by an operation count pass, which visits all basic blocks and gathers information. After processing this information, the user-specified code parts remain and are instrumented by a separate instrumentation pass.

Compiler Options. This section briefly introduces the additional compiler options. The optimization switch `-O1` enables GCC's loop analysis and additional PME passes rely on the results of this analysis. The options `ma-float-cov`, `ma-int-cov`, and `ma-ls-cov` are used to specify the relative amount of floating point, integer, or load/store operations to be annotated in the executable. Additionally, the `ma-call-cov` annotates the function calls with a call to the MA-library containing the function name. To provide an intuitive interface for the programmer, command line values are given in per cent of the estimated total amount of operations. The amount of operations a program performs, depends on the control flow. Loops, for instance, are prominent programming constructs introducing control flow and, thus, influencing the amount of operations. Depending on the exit condition of a loop, loop bounds may be derived at compile

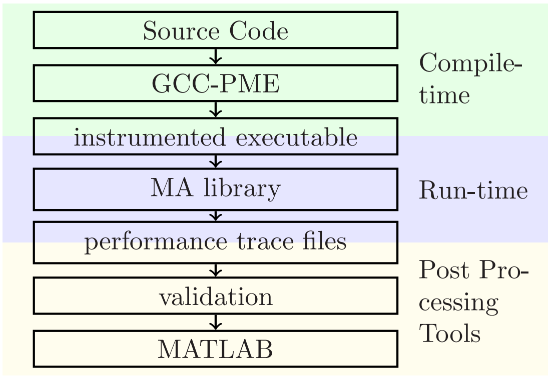


Fig. 1. Developed tool chain to create the performance model

time. To benefit from this fact, the results of GCC's loop analysis are incorporated in the PME process. This includes the results of scalar evolutions (SCEV) and induction variables analysis (IV). In case these analysis passes fail to provide a loop bound, we fall back to a user-specified value: The `ma-loop` switch defines the default value that is assumed for loops whose number of iterations was not derivable from static analysis. In order to avoid over instrumentation, steer the instrumentation process and, hence, decrease time and space overhead, the user may set a threshold value (called `ma-barrier`). The estimated amount of operations is divided by the total estimated amount of operations and compared to the threshold value. Every function or basic block must exceed the threshhold to qualify for annotation. GCC's infrastructure delivers information from the added passes to the user. Specific switches trigger the writing of the internal program representation into a file. These files contain the modifications carried out by the corresponding passes, enabling the user to examine the results of the instrumentation process. Taking a look at these files, the user will quickly and easily learn to adjust the additional compiler parameters to gain decent instrumentation.

Loop Annotation. The first PME pass annotates every loop with a call to the MA library. Herewith the identification of the loop nesting level and the frequently executed parts of an application at run time is straightforward. The arrangement of loop components in GCC's intermediate representation is shown in Figure 2. The preheader edge leads to the loop header and is mandatory for every loop. Loops are strongly connected parts of the control flow graph, which have exactly one entry block (header) [15]. The loop header holds the condition deciding to enter the loop body. After executing the loop body the latch edge is taken to the loop header. If the loop has multiple exits, the loop body may be left through an additional exit edge as well. The annotation of the loop adds the `ma_loop_start` call at the preheader edge and the `ma_loop_end` call to all exit edges. The *start* call takes three parameters: an unique identifier, variable name of the loop bound (if derivable), expected loop count (passing the corresponding variable). To complement this information the *end* call takes two arguments:

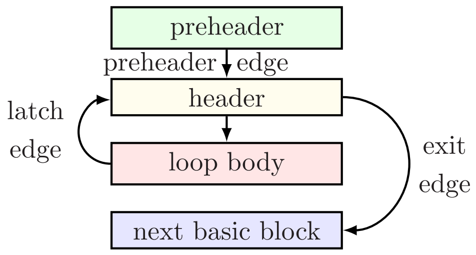


Fig. 2. GCC internal representation of a single exit loop

the already described identifier and the value of the loop count (if available from static analysis).

Operation Count Analysis. The `ma_insert` pass investigates the basic blocks. Each statement is examined whether it contains: function calls, floating point or integer operations, or load/store operations. The obtained values are added to operation count data structures. Additionally variable `est_cnt` holds the estimated number of times a basic block is executed. The results from the iteration analysis of the loops (as far as available) are stored into this variable. Nested loops lead to a multiplication of the trip counts. Loops with indeterminable number of iterations are assumed to be executed `ma_loop` times. Hence, the user setting the `ma_loop` parameter influences the effect of such loops on the model. Further, our approach relies on the fact that functions are compiled one at a time. We take advantage of the fact that functions are compiled first, hence, the results from the operation count analysis are made available to later passes. This operation count data gathered by visiting the basic blocks are saved in a per function statistic. This allows the computation of an overall statistic including all compiled functions. The `main` function is processed last. When the `main` function is compiled, the global statistic is complete as far as the data is available[1]. The processing of the data evaluates the relative contribution of the functions to the statistics and inserts them into a sorted array. This array is pruned until all remaining functions or basic blocks meet the user constraints passed with the compiler options. The `ma_insert` pass traverses the array and annotates the remaining parts.

Naming Scheme. A naming scheme (using unique names) ensures that annotated call expressions are distinguishable at run time. This naming scheme enables correlating measured values to certain regions of source code, simplifying the implementation of the post-processing tools. Besides, the naming scheme enables an optimized generation of traces, by omitting redundant actions inside the runtime library.

[1] Externally linked libraries and third party object files are not subject to the compiler's operation count analysis.

3.2 MA Library

At run-time the executable, instrumented by the GCC-PME passes, calls the MA library (which is similar regarding the API and the use of PAPI performance counters to the one developed at the Oak Ridge National Laboratory [4]). The library serves two major purposes. First, event traces for the post-processing tools are generated. This includes calling the PAPI library, that starts, resets, reads, and stops hardware event counters. Second, it writes the annotated parts into a file. In order to enable the user to complement the compiler annotated source with manual annotations, all basic functions come with wrapper functions. These wrapper functions are necessary since FORTRAN and C follow different calling conventions.

3.3 Post-processing Tools

The post-processing tools evaluate the data provided by the run time library. First, trace files generated by the library have to be prepared. Finally, MATLAB is used to visualize the data and to construct an analytical model.

Validation. The *validate* tool prepares the data obtained from the MA library. A trace file, holding the compiler assumed values from the *start* calls, is merged with a file containing performance counter values from the *end* calls. Matching these pairs of calls is simplified by the naming scheme. The outcome of the matching and aggregation process, is written to a file and serves as input for the model construction and visualization using MATLAB.

Model construction. A MATLAB program visualizes the data derived from the post-processing tool *validate* and creates an analytical model. The analytical model enables to predict runs with larger input classes. Floating point, integer or load/store operations are displayed in terms of predicted and measured values.

This section presents the construction of the performance model using the method of quadratic programming. The intention is to run a small number of program instances with different input data sets and measure the desired values. Afterwards the quadratic programming method is used to incorporate the values into a model capable of predicting unknown program runs with larger data sets. Both, model construction as well as the prediction, rely on the knowledge of the program input parameters. The model bases on the predicted and measured values for floating point, load/store or integer operations and the input parameters of the benchmark. Since the construction does not depend on the type of information measured, we will refer to the values simply as p for predicted and m for measured. The index i for p_i and m_i represents the different input classes. The separate program phases are analyzed one at a time. This aims at a fine grained modelling of the phases, correlating the predicted and measured values. The annotated program parts are evaluated with different parameters. Hence, the overall prediction is more accurate. The following mathematical equations outline the problem transformation for one measured program phase. Program parameters are represented as n_j with $n_j \in \{1, \ldots, m\}$. r is the number of different input parameter sets with m parameters each. Hence, there are r potentially different assignments for n_j. The primary goal is to transform the optimization problem in a way that MATLAB's `quadprog`-function can solve the problem. The problem is to find the optimal values for the vector x so that

$$\left| \begin{pmatrix} n_{11} & \dots & n_{m1} \\ \vdots & \ddots & \vdots \\ n_{1r} & \dots & n_{mr} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} - \begin{pmatrix} p_1 - m_1 \\ \vdots \\ p_r - m_r \end{pmatrix} \right|^2 \tag{1}$$

is minimal. Every solution for x is related to a separate program phase. For `quadprog` the term above has to be transformed into

$$\min_x \left(\frac{1}{2} x^T \mathbf{H} x + f^T x \right). \tag{2}$$

Then, MATLAB can solve the problem using quadratic programming such that the value for x is minimal. The problem transformation is outlined in the following. First, the left part of the equation is written as a sum:

$$\left| \begin{pmatrix} \sum_{i=1}^m x_i n_{i1} \\ \vdots \\ \sum_{i=1}^m x_i n_{ir} \end{pmatrix} - \begin{pmatrix} p_1 - m_1 \\ \vdots \\ p_r - m_r \end{pmatrix} \right|^2 .$$

Second, the quadratic part is calculated and written as a sum:

$$\begin{aligned} &\sum_{j=1}^r \left(\sum_{i=1}^m x_i n_{ij} - (p_j - m_j) \right)^2 \\ =& \underbrace{\sum_{j=1}^r \left[\left(\sum_{i=1}^m x_i n_{ij} \right)^2 \right]}_{=:\frac{1}{2} x^T \mathbf{H} x} + \underbrace{\sum_{j=1}^r \left[-2 * \left(\sum_{i=1}^m x_i n_{ij} \left(p_j - m_j\right) \right) + \left(p_j - m_j\right)^2 \right]}_{=:\mathbf{f}^T x + \mathbf{c}} \end{aligned} \tag{3}$$

This step leads to the structure of $\mathbf{H}$:

$$\sum_{j=1}^r \left[\left(\sum_{i=1}^m x_i n_{ij} \right)^2 \right] = \sum_{i=1}^m \left(x_i^2 \sum_{j=1}^r n_{ij}^2 \right) + 2 \left(\sum_{u=1}^m \sum_{\substack{v=1 \\ v>u}}^m x_u x_v \left(\sum_{j=1}^r n_{uj} n_{vj} \right) \right)$$

$\mathbf{H}$ is derived from comparing the coefficients as matrix:

$$\mathbf{H} = \begin{pmatrix} \sum_{j=1}^r n_{1j}^2 & \sum_{j=1}^r n_{1j} n_{2j} & \dots & \sum_{j=1}^r n_{1j} n_{mj} \\ \sum_{j=1}^r n_{2j} n_{1j} & \sum_{j=1}^r n_{2j}^2 & & \vdots \\ \vdots & & \ddots & \vdots \\ \sum_{j=1}^r n_{mj} n_{1j} & \dots & \dots & \sum_{j=1}^r n_{mj}^2 \end{pmatrix},$$

To verify the entries of matrix $\mathbf{H}$, the left hand side of equation 3 must be considered. Hence, the entries of matrix $\mathbf{H}$ are correct, if the following equation holds: $x^T\mathbf{H}x = \sum_{j=1}^{r}\left[(\sum_{i=1}^{m} x_i n_{ij})^2\right]$. Some basic mathematical calculations prove the transformation of $\mathbf{H}$ to be correct. Now, the only missing part of equation 2 is the vector $\mathbf{f}$. $\mathbf{f}$ is derived from the right hand side of equation 3 so that $\mathbf{f} = -2\begin{pmatrix} \sum_{j=1}^{r} n_{1j}(p_j - m_j) \\ \vdots \\ \sum_{j=1}^{r} n_{mj}(p_j - m_j) \end{pmatrix}$. The right hand side of equation 3 reveals a remaining part: $\mathbf{c} = \left(\sum_{j=1}^{r}(p_j - m_j)^2\right)$. The remaining part $\mathbf{c}$ does not depend on x, therefore, the optimization process is not influenced by $\mathbf{c}$. Hence, $\mathbf{c}$ can be ignored without altering the results of the optimization.

4 Results

This section contains the results obtained with the tool chain described in Section 3.1. The results are measured on an HP XC6000 cluster of Intel Itanium II processors operated at the Steinbuch Centre for Computing in Karlsruhe. The cluster comprises 108 small nodes with 2 processors (1.5 GHz and 16 GB local memory) and 12 large nodes with 8 processors (1.6 GHz and 64 GB local memory). All nodes have local disks and network adapters and are connected via a Quadrics QsNet II interconnect. Large jobs are submitted through a batch system which was instructed to allocate as many processors as MPI tasks available. The compiler options introduced in Section 3.1 are set as follows: ma-float-cov=80, ma-int-cov=0, ma-ls-cov=0, ma-call-cov[2]=100, ma-loop=100 and ma-barrier[3]=40. The goal is to omit the infrequently executed parts and annotate the frequently executed regions computing floating point operations. The results are derived with the quadratic programming method for the programs CG and IS of the NAS parallel benchmark suite as well as the numerical Lattice-Boltzmann implementation (LB). LB is a numerical method for simulating viscous fluid flow – the MPI-based implementation models a 3-D lid-driven cavity problem. In order to create an analytical model of these applications, the method for quadratic programming is employed. Section 3.3 details the transformation. Then, MATLAB's `quadprog` function solves the optimization problem. This process delivers a vector which is used to weigh the input parameters of the largest data input set. In Figure 3 the y-axis represents the number of floating point operations and the x-axis the different input classes. NPB's input parameter sets are grouped in classes to benchmark computer systems differing in computational power. In order to comply with the different problem sizes,

[2] For Lattice-Boltzmann the switch `ma-call-cov` is set to 50.

[3] For Lattice-Boltzmann the switch `ma-barrier` is set to 0.

Table 1. Average relative error predicting input class C for IS, CG, and LB

	Number of MPI Tasks			
Benchmark	2	8	16	32
IS	0%	0%	0%	0%
CG	19.8%	21.1%	23.3%	29%
LB	0.2%	0%	−1.3%	−12.1%

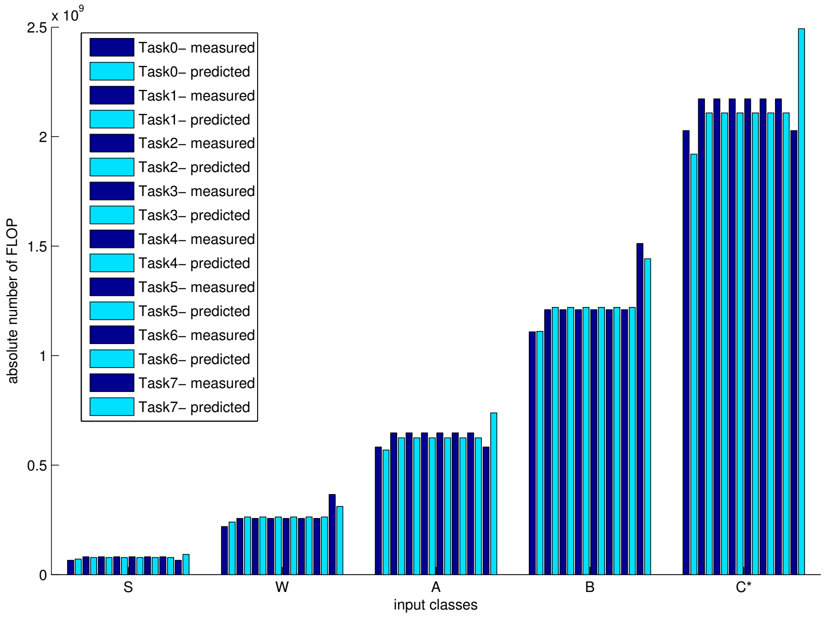


Fig. 3. Quadratic programming model predicting the Lattice Boltzmann numerical workload executed with eight MPI tasks on two large nodes

the following input parameter classes were defined for LB – each one specifies the volume of a cube:

	S	W	A	B	C
`volume`	64000	216000	512000	1000000	1728000

In Figure 3 problem sizes are increasing along the x-axis. Two adjacent bars in this figure correspond to the same task. The left bar (in dark blue) represents the measured amount of floating point operations, whereas the right bar (in light blue) shows the amount predicted by the model. The difference of these bars shows the model's accuracy. Input classes S, W, A, and B were used to construct the model. Input class C did not contribute to the modeling process. Thus, the bar with the measured values enables us to specify the quality of the prediction. All runs for all input parameter sets are performed once, but only the obtained values with classes S, W, A, and B are used to construct the model. In Figure 3 class C is marked with a star to indicate the prediction. The prediction for program CG, in Table 1, shows that the modeling of class C yields an error between 19% and 23% with 2 to 16 MPI tasks and increases to 29% measured with 32 tasks. However, CG offers potential for refined modeling. Due to its regular structure, the IS program is predicted with an error less than 1% for all numbers of tasks (see Table 1). Lattice-Boltzmann shows an error of less than 1% for 2 and 8 tasks. Figure 3 depicts the reason for the increasing error rate for larger task numbers: the unbalance of the last task inhibits the modeling process and contributes large parts to the error. Anyhow, an average error rate of 12% for 32 tasks is acceptable.

5 Conclusion

This work presents a compiler-based approach towards the construction of performance models for parallel applications. The GCC-PME adds compiler passes for gathering operation count data and instrumenting the executable. A library, called while the program executes, accesses hardware performance counters. In addition a chain of post-processing tools is designed, to relate the compiler predicted values to the performance counters. The models are verified using hardware event counts, which enables to quantify the prediction accuracy. Quadratic programming models are applied for the prediction of large problem sizes. The achieved precision in predicting the number of floating point operations for bigger problem sizes shows that this modeling approach is promising. In particular the programs LB and IS with emphasis on computational aspects are shown to be predictable.

Some improvements of the tool chain became obvious. First, the compiler should support a fine-grained level modeling. The results presented for the CG benchmark reveal that modeling on a per function level should be replaced by a more detailed modeling of functions. Collecting operation counts for smaller code fragments than functions and use those as input for the model construction is one promising approach. Here, the contradicting requirements of modeling in greater detail and limiting the overhead caused by instrumentation are challenging. Second, the fast and straightforwardly constructed quadratic programming models could be complemented by symbolic models. The tool chain is designed to also support models relying on information available at compile time. For instance, the propagation of variable names acting as loop bounds is particularly important in symbolic models. Thus, a combination of PME compiler passes and symbolic models is promising.

Acknowledgment

The authors would like to thank Matthias Bartelt for expert assistance and skilled implementation work. Further, this work was supported in part by the EC under grant HiPEAC IST-004408.

References

1. Frank, M.I., Agarwal, A., Vernon, M.K.: LoPC: modeling contention in parallel algorithms. SIGPLAN Not. 32(7), 276–287 (1997)
2. Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.: LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. Technical report, Santa Barbara, CA, USA (1995)
3. Culler, D.E., Karp, R.M., Patterson, D., Sahay, A., Santos, E.E., Schauser, K.E., Subramonian, R., von Eicken, T.: LogP: a practical model of parallel computation. Commun. ACM 39(11), 78–85 (1996)
4. Alam, S., Vetter, J.: A framework to develop symbolic performance models of parallel applications. In: IEEE International Parallel & Distributed Processing Symposium, p. 368 (2006)
5. Bhatia, N., Alam, S.R., Vetter, J.S.: Performance modeling of emerging HPC architectures. In: HPCMP Users Group Conference, pp. 367–373 (2006)
6. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005)
7. Marin, G.: Semi-automatic synthesis of parameterized performance models for scientific programs. Master's thesis, Rice University, Houston, Texas (April 2003)
8. Marin, G., Mellor-Crummey, J.: Cross-architecture performance predictions for scientific applications using parameterized models. In: SIGMETRICS 2004/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems, pp. 2–13. ACM, New York (2004)
9. Yang, L.T., Ma, X., Mueller, F.: Cross-platform performance prediction of parallel applications using partial execution. In: SC 2005: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p. 40. IEEE Computer Society, Los Alamitos (2005)
10. Carrington, L., Snavely, A., Wolter, N.: A performance prediction framework for scientific applications. Future Generation Computer Systems 22(3), 336–346 (2006)
11. Lee, B.C., Collins, J., Wang, H., Brooks, D.: CPR: Composable performance regression for scalable multiprocessor models. In: MICRO: 41st International Symposium on Microarchitecture (2008)
12. Lee, B.C., Brooks, D.: Efficiency trends and limits from comprehensive microarchitectural adaptivity. SIGOPS Oper. Syst. Rev. 42(2), 36–47 (2008)
13. Deelman, E., Dube, A., Hoisie, A., Luo, Y., Oliver, R.L., Sundaram-Stukel, D., Wasserman, H.J., Adve, V.S., Bagrodia, R., Browne, J.C., Houstis, E.N., Lubeck, O.M., Rice, J.R., Teller, P.J., Vernon, M.K.: POEMS: end-to-end performance design of large parallel adaptive computational systems. In: WOSP 1998: Proceedings of the 1st International Workshop on Software and Performance, pp. 18–30 (1998)
14. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
15. Dvořák, Z.: Loop optimizer cheatsheet, `http://gcc.gnu.org/wiki/GettingStarted?action=AttachFile&do=get&target=loopcheat.ps`

A Method for Accurate High-Level Performance Evaluation of MPSoC Architectures Using Fine-Grained Generated Traces

Roman Plyaskin and Andreas Herkersdorf

Institute for Integrated Systems, Technische Universität München,
Arcisstr. 21, 80290 Munich, Germany
{roman.plyaskin,herkersdorf}@tum.de
http://www.lis.ei.tum.de

Abstract. Performance evaluation at system level has become a prerequisite in the design process of modern System-on-Chip (SoC) architectures. This fact resulted in many simulative methods proposed by the research community. In trace-based simulations, the performance of SoC architectures is evaluated using abstracted traces. This paper presents an approach for the generation of the traces at the instruction level from a target SW code executed on a cycle accurate CPU simulator. We showed that the use of fine-grained traces provides accuracy above 95% with an increase of simulation performance by factor of 1.3 to 3.8 compared to the reference cycle accurate simulator. The resulting traces are used during high-level explorations in our trace-driven SystemC TLM simulator, in which performance of MPSoC (Multiprocessor SoC) architectures with a variable number of CPUs, diverse memory hierarchies and on-chip interconnect can be evaluated.

1 Introduction

Constantly increasing complexity of System-on-Chip (SoC) architectures, stimulated by the rising amount of transistors on a single chip, faces new challenges in the design of integrated circuits. In order to shorten product development cycles under high time-to-market pressure, early system-level modeling and simulation has become a necessary part of the design process. Due to higher complexity and many low-level details, cycle accurate system-level simulations are not feasible from the perspective of simulation time. At system level, components are typically modeled at a high level of abstraction allowing faster and more flexible design space exploration. Therefore, the main challenge is to perform system-level simulations fast and accurately at the same time.

For the performance evaluation, which is addressed in this paper, trace-based simulations have been widely used for general purpose computer systems as well as for systems-on-chip [7,8,10,11,13,15]. The trace-based approach represents hardware components as black-box modules that either perform internal processing or make read or write requests on the communication infrastructure.

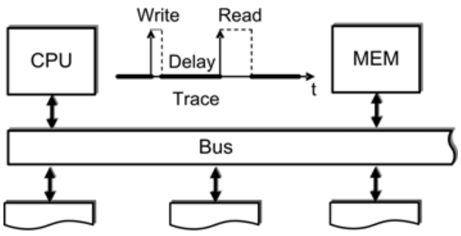


Fig. 1. Trace-based simulation

Thus, the workload generated by the component can be captured in the form of a so called *trace* (Fig. 1).

A trace is a list of pseudocode constructs or *trace primitives*. Each primitive defines an action which the abstracted module should perform. For example, when executing a *delay* primitive (denoted as Delay in the figure) the abstracted model of the CPU waits for a certain amount of time simulating a processing latency. On execution of *read* or *write* primitives (denoted as Write and Read), the CPU model issues correspondingly a read or write transaction to the memory on the bus. The overall time required to access the memory module will depend on the bus arbitration, data transfer latencies on the bus, and read/write latency of the memory. Thus, the execution of the next delay primitive will be postponed correspondingly. In [15], the authors demonstrate how trace-based simulations can be applied for abstract application modeling and performance evaluation of network processor architectures.

Since the internal functionality of the components is abstracted, performance of trace simulations can be significantly increased compared to fully functional and cycle accurate simulations, e.g. performed using an instruction set simulator (ISS). In case of MPSoC architectures, the performance becomes a vital factor since simultaneous instantiation of multiple cycle accurate CPU simulators can significantly decrease the simulation time.

The trace representation allows system designers to describe a workload at various levels of granularity. In a coarse-grained workload, traces contain approximated processing latencies of a CPU without a detailed pattern of memory accesses. This level of granularity may be of a great interest when the target SW code is not available, but the designer can have an idea how the components could interact with each other in the desired application [15].

In this paper, we propose to use abstract traces for accurate performance estimation during high-level design space explorations of MPSoC architectures, and address the challenge of how to derive accurate fine-grained traces. In our approach, the traces are defined at the instruction level which allows achieving exact bus access patterns and precise processing latencies of a CPU. We present a complete workflow in which traces generated from a cycle accurate CPU simulator are used for rapid evaluations of MPSoC architectures in our trace-based SystemC TLM simulator. We demonstrate how the trace-based approach can be applied during a design space exploration phase, when functional repartitioning between dedicated HW accelerators and CPUs is performed. The paper shows examples of the performance estimation using traces and discusses origins of the errors produced by the proposed method.

2 Related Work

Performance evaluation of SoC architectures using high level models has been addressed in many research works. One of possible solutions for obtaining accurate timing information at system level is the integration of instruction set simulators into high-level SystemC models. MPARM [2] is a simulation platform for MPSoC architectures, in which a cycle accurate model of an ARM processor is integrated into SystemC environment. In [4], an ISS is co-simulated with SystemC using the GDB-kernel. In this method, the high level model receives timing information via commands sent through the debugging interfaces. However, the integration of the instruction set simulators significantly decreases the simulation performance.

Along with HW/SW co-simulation using an ISS, there is another modeling principle in which precise timing information is back-annotated to abstracted high-level simulations. This method is widely used in software code instrumentation techniques. In [12], a target C code is used to generate a SystemC code in which the execution time is partially determined at the compilation time. In turn, dynamic timing information, e.g. effects of caches and branch predictors, is obtained at simulation runtime using the models of the architecture components. In [6], a target code is instrumented with additional functional calls for keeping a count of the executed cycles and for accessing the TLM communication infrastructure.

In [7], the authors show how traces can be employed for generation of communication analysis graphs which are later used for evaluation of on-chip communication architectures. In Sesame [10] and Spade [8] frameworks, traces are used to represent the workload of multimedia application models constructed using Kahn Processing Networks (KPN). However, in these approaches the range of target applications is restricted to those that are adaptable to the KPN model of computation. Moreover, processing latencies and communication transactions specified by these traces are very coarse-grained, i.e. delay entries represent large blocks of computation. At this granularity level, exact memory traffic, which can significantly influence the application performance, is not addressed. For example, during the computation of a KPN process, contention of multiple CPUs on the shared on-chip interconnect during cache misses cannot be considered at this abstraction level. In our method, traces specify precise patterns of memory accesses and processing latencies allowing for more accurate analysis of the application workload.

S. Mahadevan et al. demonstrated a technique in which traces, obtained from a cycle accurate CPU simulator, are used to represent the component's behavior exactly captured at its bus interface [9]. Contrarily, in our method we define the traces at the instruction level and explicitly model the cache component which can be reconfigured during design space explorations. Consideration of the cache effects in high-level models is necessary since caching significantly alters the application performance. In our method, communication latencies are defined at simulation runtime depending on the miss rate of the cache model as well as utilization of the shared on-chip communication infrastructure by other CPU components. Moreover, we focus on the correspondence between the instructions of particular subroutines and their trace representation. This allows the designer to profile the traces during high-level design space explorations.

3 System-Level Simulations Using Traces

The workflow for generation of traces and their use in a trace-driven MPSoC simulator is presented in Fig. 2. In the first step, the object code of a cross-compiled application is executed on a cycle accurate simulator of the target CPU. In addition, we deploy an objdump utility to create a symbol table of the code. Information produced by the simulation as well as the symbol table are further processed by a trace generator tool which produces a trace file of the target application.

The resulting trace is used as a part of the workload in our trace-driven multiprocessor SystemC TLM simulator, in which diverse (MP)SoC architectures with a varying number of CPUs, different cache models, hierarchical on-chip interconnect can be modeled and analyzed. The simulator is highly configurable by means of an XML description file in which the parameters of the components as well as their interconnection can be specified.

In the trace simulator, the resulting traces are executed on the abstract CPU models in a new multiprocessor environment. Thus, the designer can evaluate

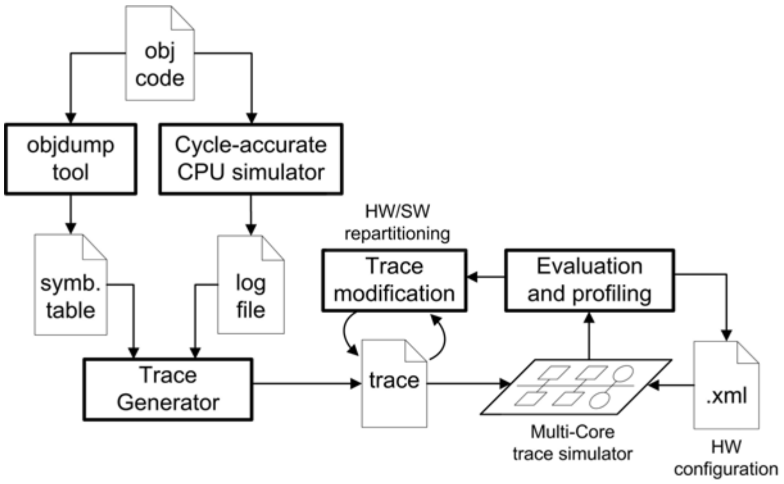


Fig. 2. Workflow for high-level simulations using traces

the impact of shared resources on performance of the application. In addition, the simulator can profile the application traces allowing the designer to identify possible options for HW/SW functional repartitioning. The repartitioning can be performed by a simple trace modification and analyzed in the simulator in an iterative manner. The following sections provide further details on each step of the proposed workflow.

3.1 Generation of Traces

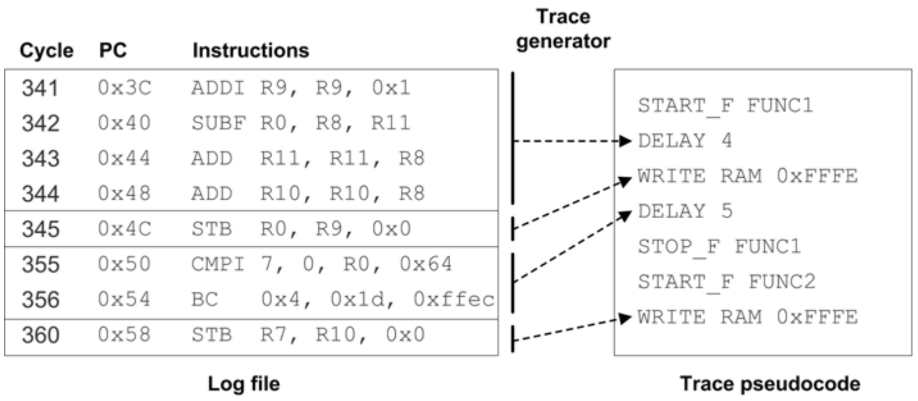
In order to estimate the execution time of the target software, cycle accurate CPU simulators typically contain models of micro-architectural components, e.g., branch predictors or instruction pipelines. Using a CPU simulator, the designer can observe the contents of CPU's internal registers and memory addresses in load and store instructions. For the generation of traces, the order of the instructions as well as their timing information is of the most importance.

In a cycle accurate simulation, the executed instructions can be categorized into two types:

- Processing instructions not resulting in a request on the bus;
- Communication instructions that perform load and store operations.

Since none of the processing instructions initiates memory accesses, a group of subsequent processing instructions are translated to a `DELAY` trace primitive (Fig. 3). The trace generator calculates the overall execution time of this group and stores it as a parameter of the primitive. Correspondingly, the communication instructions are translated to `READ` and `WRITE` trace primitives. Information on the destination target of each transaction and the target memory address is stored as primitive's parameters as well.

In order to enable profiling of the application trace, trace primitives are divided into groups. Each group represents an execution of the instructions within a particular subroutine of the target SW code. For each group, the trace generator identifies the subroutines' names using the symbol table of the target code



```
Cycle  PC    Instructions
341    0x3C  ADDI R9, R9, 0x1
342    0x40  SUBF R0, R8, R11
343    0x44  ADD  R11, R11, R8
344    0x48  ADD  R10, R10, R8
345    0x4C  STB  R0, R9, 0x0
355    0x50  CMPI 7, 0, R0, 0x64
356    0x54  BC   0x4, 0x1d, 0xffec
360    0x58  STB  R7, R10, 0x0
```

```
START_F FUNC1
DELAY 4
WRITE RAM 0xFFFE
DELAY 5
STOP_F FUNC1
START_F FUNC2
WRITE RAM 0xFFFE
```

Fig. 3. Generation of traces using a log file from a CPU cycle accurate simulation

and program counter values obtained during the cycle accurate simulation. The start and end positions of the groups are denoted using `START_F` and `STOP_F` primitives.

In the final step of the trace generation, each primitive type is encoded using a unique identifier. Afterwards, the primitives are stored in a file in the binary form, thus, reducing the size of the generated trace.

3.2 Trace Simulator

During the trace simulation which is similar to [15], the black-box CPU models execute primitives of the trace files. The simulation time is advanced according to the processing latencies given in the delay primitives. The absolute SystemC time interval which the CPU model waits is calculated as the annotated number of cycles multiplied with the value of the clock frequency of that CPU component.

On execution of read or write primitives, the CPU performs a request to the cache component. Given a memory address tagged to the primitive, the cache model indicates either a hit or a miss for the current transaction. In case of a cache miss, the CPU issues a blocking TLM transaction on the shared arbitrated bus. The cache model used in the trace simulator is highly configurable. The user can configure cache associativity as well as various replacement policies. In contrast to computational latencies that are statically defined by the delay primitives, communication latencies are obtained dynamically at simulation runtime, depending on how shared resources, e.g. on-chip interconnect, are utilized by other CPUs. Please note that neither read nor write primitives are annotated with the data that has to be transferred. Since the functionality of the memory module is abstracted to access latencies only, the actual data written or read from the memory is not required.

On execution of `START_F` primitives, the trace simulator starts accumulating the execution time of the annotated subroutine until a `STOP_F` primitive is reached. This information is used later for generation of the profiling results. In addition to the profiling capabilities, the trace simulator is able to measure the utilization of each SoC component. In order to do so, the simulator accumulates the busy time of the respective component. At the end of the simulation, the utilization is calculated as a ratio between the overall busy time and the total simulated time.

3.3 Trace Modification

Abstracted representation of the target application using traces allows rapid changes of the workload generated by a CPU. Using a simple text processing tool, the application trace can be modified in an arbitrary way, thus, enabling faster design exploration cycles. Particularly, this could be useful when the designer wants to repartition the functionality between CPUs and HW accelerators and evaluate the resulting impact on the application performance. Since the trace simulation does not require transfers of functional data, the user can create an

```
WRITE_HW dev_id n_bytes
READ_HW  dev_id n_bytes
```

Fig. 4. Trace primitives for accessing a hardware accelerator

abstracted model of the desired HW accelerator, which on request will simulate certain functionality.

Functional repartitioning can be performed by a substitution of certain parts of the trace by new patterns for accessing the accelerator component. For this purpose, we have introduced new trace primitives shown in Fig. 4. A `WRITE_HW` primitive represents a transfer of input data to the accelerator denoted by `dev_id`. Upon reception of the data, the hardware accelerator simulates internal processing by waiting for a certain amount of time configured by the user. On execution of a `READ_HW` primitive, the CPU constantly polls the accelerator until the processed data becomes available, and continues executing next trace primitives after the successful read operation. The amount of data that should be written to or read from the accelerator is specified by `n_bytes` parameter.

Although the functional data is not transferred to the accelerator, the designer can still investigate the impact of associated communication latencies on the application's performance. In our framework, the abstracted model of the accelerator was designed in a way in which the component gets locked for a certain CPU during data processing. Therefore, in MPSoC architectures other CPUs will be stalled on a simultaneous access to the component. Please note that the designer can specify more complex access patterns, depending on the type of the hardware accelerator.

The processing latency of the HW accelerator can be annotated from the data specifications if the peripheral's implementation is already available. As an alternative, the proposed methodology can be used for setting the requirements for a not yet existing accelerator. In both cases, the designer is assisted in finding the optimal trade-off between the accelerated execution of the function and the additional load on the on-chip communication infrastructure.

4 Experimental Results

In this section, we estimate accuracy of our trace generation method and demonstrate the proposed workflow for evaluation of alternative MPSoC architectures using traces.

In the following experiments, cycle accurate simulations were performed in COMeT tool developed by VaST Systems [14]. The tool provides a library of cycle-accurate CPU models that are widely used in industry, as well as models of on-chip buses, memories and other components. Thus, the designer can create a complete model of a system-on-chip which is capable of executing the target binary code.

4.1 Accuracy of Generated Traces

In order to evaluate accuracy of the proposed trace generation method and to estimate a possible gain in simulation performance compared to the cycle accurate simulator, we took a set of five benchmarks. Each benchmark represented a particular application domain. In *jpeg* benchmark, the application performed encoding of a bitmap image into the JPEG format [3]. Additionally, we selected *bitcount*, *FFT*, *stringsearch*, and *sha* benchmarks from MiBench suite [5] that represent correspondingly Automotive, Telecommunications, Office, and Security application categories. The experiments were conducted on a PC with a 2 GHz Intel Core 2 Duo processor and 1 GB of RAM. During the tests we used the following input data for the applications:

– *Jpeg*: a test bitmap image of size 104×72 pixels;
– *FFT*: polynomial function with one random sinusoid and 512 samples;
– *Bitcount*: 50000 iterations;
– *Sha*: the small ASCII text file provided with the benchmark;
– *Stringsearch*: the large string provided with the benchmark.

The cross-compiled C-code of each application was executed on a CoMET virtual platform. The SoC architecture modeled in the tool consisted of a PowerPC e200z6 CPU with a 32 kB write-back cache, a generic memory model with an access latency of 1 cycle, and a generic model of a bus with a request latency of 1 cycle. The resultant log was further processed by the trace generator which produced a trace file for each benchmark code.

In the next step, we executed the generated traces in the trace simulator (TS) containing the same SoC components as in the CoMET virtual platform. Parameters of the abstracted modules, e.g. the memory and bus latencies as well as the cache size, were adjusted to the parameters of the CoMET modules. Clock frequencies of the components both in CoMET and TS were set to 100 MHz. Results of the estimated execution time for each benchmark are given in Table 1.

Although the trace simulator does not have a notion of instructions, the simulation performance in MIPS was calculated as the number of instructions of the corresponding application divided by the time of the trace simulation. We refer to the simulation time as a real time elapsed from the start to the end of

Table 1. Comparison of VaST CoMET and trace-based simulations

Benchmark	Number of instructions	Estimated cycles			Simulation performance, MIPS			VaST perf. with trace gen., MIPS
		VaST	TS	Error,%	VaST	TS	Speedup	
stringsearch	1.68M	2.86M	2.93M	2.18	7.01	11.97	1.71	0.94
jpeg	3.01M	5.18M	5.16M	-0.37	6.52	25.07	3.84	0.98
sha	10.81M	12.8M	12.45M	-2.77	54.60	72.07	1.32	1.18
FFT	21.68M	27.23M	25.96M	-4.67	28.64	60.23	2.10	1.00
bitcount	34.18M	43.16M	42.71M	-1.04	90.17	126.57	1.40	1.40

the simulation. This time does not include the initialization phase of the simulation, in which the components' models are instantiated. The initialization phase in VaST CoMET was approximately measured to be 5 s. In contrast, due to the higher abstraction level of the components, the initialization in the trace simulator took 0.03 s. Small initialization times may be desirable when several simulations have to be performed iteratively. The performance of VaST simulations given in the table was measured without generation of traces. With the enabled trace generation, the simulator's performance was reduced to 0.94–1.4 MIPS due to the overhead associated with creating the trace files.

The errors in the trace-based simulation originate from the simplified memory access mechanisms. First, during the trace generation the actual timing of the read/write primitives is considered to be 1 cycle in case of cache hits. In fact, the execution of the corresponding load/store instructions can take more than one cycle, e.g., due to possible pipeline stalls. In the trace simulation, these effects are omitted and, thus, the execution time is underestimated. Second, on cache misses the CPU model issues blocking transactions to the memory. However, in reality the CPU would continue executing next instructions until it stalls due to the data dependencies of the upcoming instruction. Therefore, the executed time becomes overestimated. The overall error produced by the trace simulation is a superposition of these two effects. If higher simulation accuracy is required, the trace generation method has to be correspondingly modified with a consideration of possible data dependencies inside delay primitives.

4.2 Design Space Exploration

In order to demonstrate the presented workflow during high-level explorations of MPSoC architectures, we selected the *jpeg* application as a simplified but representative example. As one of many possible architectural solutions, the application can be executed on a symmetric multi-processor platform, in which each CPU performs processing of a particular fragment of the picture.

On an architecture with multiple CPUs that share common resources, the execution time of the application will be altered due to additional contention periods. In order to study this effect in the trace simulator, we modeled a sample architecture consisting of 4 CPUs, a common arbitrated bus and a shared memory. Since each CPU should process an individual block of data, we generated 4 different traces. Each trace represented the processing of an image block of size 128×128 pixels. The traces were executed in parallel without any data dependencies between them. Table 2 presents results of the application's profiling in the trace simulator.

As can be concluded from the experiment, the most performance demanding subroutine was a calculation of the discrete cosine transform (*dct*). As a possible solution during the design space exploration, this function can be offloaded to a hardware accelerator attached to the peripheral bus (Fig. 5). The discrete cosine transform is performed on an image block of size 8×8 pixels resulting in data transfers of 64 bytes to the accelerator. In order to enable the hardware acceleration, we replaced every part of the traces representing the execution of

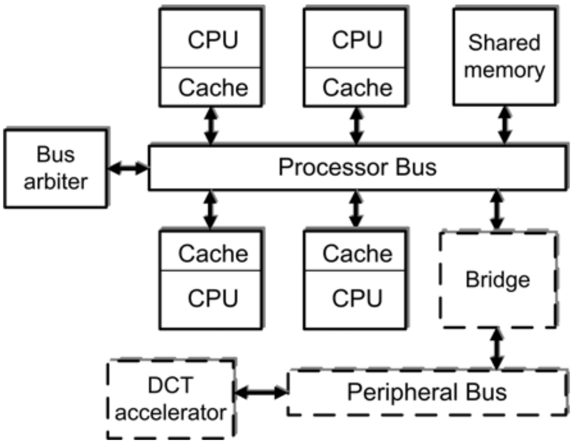


Fig. 5. MPSoC architecture modeled in the trace simulator before and after repartitioning

dct function by `WRITE_HW` and `READ_HW` primitives configured with value `64`. On execution of `WRITE_HW`, the CPU was trying to access the component by polling its status. If the peripheral was idle, the CPU performed 16 write transactions on the 32-bit bus. After the processing latency assigned to the accelerator had expired, the CPU performed 16 read transactions representing the transfer of the processed data, and the accelerator became available for other CPU components.

In [1], L. V. Agostini et al. demonstrated an FPGA implementation of the DCT algorithm and obtained the processing time of an 8×8 pixel block (5.6 μs). We took this value as a reference and annotated the equivalent processing latency to the abstracted accelerator (560 cycles). The latency parameter is configurable and can be changed if other HW implementations have to be considered. The simulation results of the new MPSoC architecture with the DCT accelerator are given in Table 3.

Table 2. Profiling results of *jpeg* application on a 4-CPU architecture in the trace simulator

Subroutine	Executed cycles				Share of total execution time, %			
	CPU0	CPU1	CPU2	CPU3	CPU0	CPU1	CPU2	CPU3
dct	2,278,193	2,277,795	2,277,716	2,277,917	28.1	29.4	28.2	28.1
WriteRawBits16	1,339,100	1,326,305	1,333,059	1,345,854	16.5	17.1	16.5	16.6
EncodeDataUnit	842,872	839,441	841,284	845,318	10.4	10.8	10.4	10.4
zzq_encode	817,885	819,343	816,977	818,143	10.1	10.6	10.1	10.1
get_MB	475,130	475,671	474,828	475,123	5.9	6.1	5.9	5.9
RGB2YCrCb	438,654	438,916	438,539	439,052	5.4	5.7	5.4	5.4
...	...	...	...	...	...	...	...	...

Table 3. Comparison of execution time of *jpeg* application on a 4-CPU architecture with and without the DCT accelerator

CPU instance	Execution time, cycles			
	dct function (average)		*jpeg* application (total)	
	w/o DCT-HW	with DCT-HW	w/o DCT-HW	with DCT-HW
CPU0	5,910	668	8,098,417	6,107,352
CPU1	5,909	653	7,752,352	5,755,998
CPU2	5,909	656	8,091,063	6,091,775
CPU3	5,909	657	8,109,849	6,115,119

Due to the additional data transfers to the hardware accelerator and simultaneous use of the component by multiple CPUs, *dct* function took approximately 1.2 times more than the annotated processing time of the accelerator. Nevertheless, the overall application's performance was increased by a factor of 1.3 compared to the architecture without the DCT accelerator.

5 Conclusions

In this paper, an approach for MPSoC performance estimation was presented in which two domains of cycle accurate and trace-based simulations are combined. For this purpose, we developed a tool that automatically generates a trace file from a CPU cycle accurate simulation at the instruction level. We showed that the trace simulation allows achieving better simulation performance with a marginal loss of accuracy compared to the reference CPU simulator. We further demonstrated how a trace can be applied during high-level design space explorations, particularly for the analysis of functional repartitioning between CPUs and HW accelerators in an MPSoC. Our approach can be also applied for trace generation using RTL processor models. This would allow achieving better accuracy of the traces and even higher simulation speed up comparing to the instruction set simulators.

There are some problems that have to be solved in the presented method. Currently, an instruction cache is not considered during the trace generation. It is also assumed that there are no data dependencies between the traces running on a MPSoC architecture. In our future work, in addition to overcoming these problems we are planning to add inter-trace synchronization to our trace simulator, and extend the framework for modeling complex multi-tasking applications running under control of a real-time operating system.

Acknowledgments. We would like to thank J. Zeppenfeld and H. Rauchfuss for their help on the trace-based SystemC TLM simulator, as well as Z. Wang for his suggestions for the improvement of simulation performance. We also would like to express our gratitude to the VaST Academic Program for providing us with CoMET tool, BMW and Bosch for supporting our work, as well as to many reviewers for their valuable comments.

References

1. Agostini, L.V., Silva, I.S., Bampi, S.: Pipelined Fast 2-D DCT Architecture for JPEG Image Compression. In: SBCCI 2001: Proceedings of the 14th symposium on Integrated circuits and systems design, Washington, DC, USA, p. 226. IEEE Computer Society, Los Alamitos (2001)
2. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. J. VLSI Signal Process. Syst. 41(2), 169–182 (2005)
3. Embedded JPEG Codec Library, `http://sourceforge.net/projects/mbjpeg`
4. Formaggio, L., Fummi, F., Pravadelli, G.: A Timing-Accurate HW/SW Co-Simulation of an ISS with SystemC. In: International Conference on Hardware/Software Codesign and System Synthesis, CODES + ISSS 2004, pp. 152–157 (2004)
5. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Workload Characterization, Annual IEEE International Workshop, pp. 3–14 (2001)
6. Kempf, T., Karuri, K., Wallentowitz, S., Ascheid, G., Leupers, R., Meyr, H.: A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation. In: DATE 2006: Proceedings of the conference on Design, automation and test in Europe, Leuven, Belgium, pp. 468–473. European Design and Automation Association (2006)
7. Lahiri, K., Raghunathan, A., Dey, S.: System-Level Performance Analysis for Designing On-Chip Communication Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20(6), 768–783 (2001)
8. Lieverse, P., Stefanov, T., van der Wolf, P., Deprettere, E.: System Level Design with Spade: An M-JPEG Case Study. In: IEEE/ACM International Conference on Computer Aided Design, ICCAD 2001, pp. 31–38 (2001)
9. Mahadevan, S., Angiolini, F., Sparso, J., Benini, L., Madsen, J.: A Reactive and Cycle-True IP Emulator for MPSoC Exploration. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(1), 109–122 (2008)
10. Pimentel, A.D., Thompson, M., Polstra, S., Erbas, C.: Calibration of Abstract Performance Models for System-Level Design Space Exploration. J. Signal Process. Syst. 50(2), 99–114 (2008)
11. Prete, C.A., Prina, G., Ricciardi, L.: A Trace-Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor Systems. IEEE Trans. Parallel Distrib. Syst. 6(9), 915–929 (1995)
12. Schnerr, J., Bringmann, O., Viehl, A., Rosenstiel, W.: High-Performance Timing Simulation of Embedded Software. In: DAC 2008: Proceedings of the 45th annual Design Automation Conference, pp. 290–295. ACM, New York (2008)
13. Sherman, S.W., Browne, J.C.: Trace Driven Modeling: Review and Overview. In: ANSS 1973: Proceedings of the 1st symposium on Simulation of computer systems, pp. 200–207. IEEE Press, Los Alamitos (1973)
14. VaST Systems Technology, `http://www.vastsystems.com`
15. Wild, T., Herkersdorf, A., Lee, G.-Y.: TAPES – Trace-Based Architecture Performance Evaluation with SystemC. Design Automation for Embedded Systems 10(2-3), 157–179 (2005)

JetBench: An Open Source Real-Time Multiprocessor Benchmark

Muhammad Yasir Qadri[1], Dorian Matichard[2], and Klaus D. McDonald Maier[1]

[1] School of Computer Science and Electronic Engineering
University of Essex, CO4 3SQ, UK
yasirqadri@acm.org, kdm@essex.ac.uk
[2] Ecole Nationale d'Electronique, Informatique et Radiocommunications de Bordeaux, ENSEIRB
matichar@enseirb.fr

Abstract. Performance comparison among various architectures is generally attained by using standard benchmark tools. This paper presents JetBench, an Open Source OpenMP based multicore benchmark application that could be used to analyse real time performance of a specific target platform. The application is designed to be platform independent by avoiding target specific libraries and hardware counters and timers. JetBench uses jet engine parameters and thermodynamic equations presented in the NASA's EngineSim program, and emulates a real-time jet engine performance calculator. The user is allowed to determine a flight profile with timing constraints, and adjust the number of threads. This paper discusses the structure of the application, thread distribution and its scalability on a custom symmetric multicore platform based on a cycle accurate full system simulator.

Keywords: Real-time, Multiprocessor, Application Benchmark.

1 Introduction

Benchmarks are generally classified into two types, i.e. 1) synthetic benchmarks and 2) application benchmarks. Synthetic benchmarks are designed to exploit particular property of a processor such as instruction per second (IPS), cache performance, I/O bandwidth etc, whereas application benchmarks are centred towards one particular application such as automotive, office automation, etc. The concept of using benchmarks for performance characterization of the system is common practice and some processor manufacturers have proposed their own benchmarks [1]. However such benchmarks strive to give better performance on a particular platform, third party benchmarks are a good way to compare the performance amongst various architectures impartially and transparently.

The JetBench benchmark presented in this paper is an application benchmark written in C, for real-time jet engines thermodynamic calculations. It is a multithreaded application for shared memory architectures. The benchmark is based on OpenMP [2], and could be seamlessly ported to any platform supporting it. The benchmark provides user the flexibility to specify custom workload, that could be a real flight profile with deadlines. The benchmark records the time consumed in calculating individual data points, and reports the miss of deadlines. The benchmark is scalable to theoretically any number of cores and could be used as a tool to measure an operating system's scheduling characteristics. This paper is divided into five sections. The following section overviews the related work in the area of embedded benchmarking, section 3 and 4 detail the proposed benchmark characteristics, and results based on a multicore architecture. Finally the last section forms the conclusion.

2 Related Work

With the current drive towards multicore platforms, standard APIs like OpenMP, POSIX [3] and Message Passing Interface (MPI) [4] have facilitated the development of multicore threaded applications. Multicore platforms have been widely applied in the real-time systems to achieve higher throughput and lower power consumption.

The embedded system community has long been using non-embedded benchmarks such as SPEC [5], Whetstone [6], Dhrystone [7] and NAS parallel benchmarks [8], to evaluate the performance of the target systems. A limited number of benchmarks are specifically designed for the embedded system evaluation.

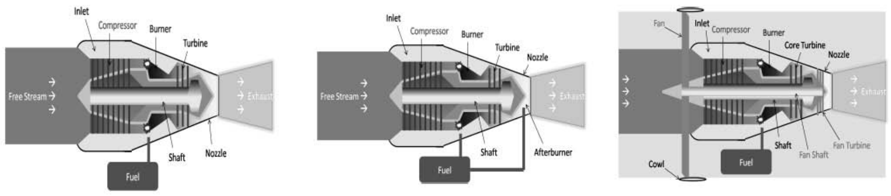
One of the few embedded system specific benchmark suites is the Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark tools suite comprising of algorithms and applications targeting telecommunication, networking, automotive, and industrial products. A recent addition of a so called MultiBench [9] suite has realized the performance evaluation of shared memory symmetric multicore processors. These benchmarks could be targeted to any platform supporting POSIX thread library, and are delivered as customizable set of workloads, each comprising of one or more work items. Although computationally rich and extensive the benchmarks by no means provide real time performance statistics of the system, and for such applications EEMBC has two applications in a separate single core benchmark suite called AutoBench [10]. This benchmark suite comprises of real time applications such as 'Angle to Time Conversion' and 'Tooth to Spark' [11]. The Angle to Time Conversion application simulates an embedded automotive application, where the processor measures the real-time delay between pulses sensed from the gear on the crankshaft. Then it calculates the Top Dead Center (TDC) position on the crankshaft, computes the engine speed, and converts the tooth wheel pulses to crankshaft angle position. The Tooth-to-Spark application simulates an automotive application that processes air/fuel mixture and ignition timing in real-time. Another real-time single core embedded benchmark is PapaBench [12], that is based on a unmanned aerial vehicle (UAV) control software for AVR and ARM microcontroller systems. The benchmark provides the worst case execution time computation which is useful for systems scheduling analysis. Guthaus et al. [13] presented the MiBench embedded benchmark suite. This benchmark suite is a single core, non real-time implementation of 35 applications in the areas such as automotive/industrial, consumer, office, network, security, and telecommunication. As all of the above mentioned benchmarks either are not using threaded implementation or are not real-time applications, a more specific benchmark suite addressing the two issues altogether is developed by Express Logic

Inc., i.e. the so called ‘Thread-Metric’ benchmark suite [14]. The tool is specifically designed to measure a real-time operating system’s (RTOS) capability to handle a threaded application. The benchmark is not a multiprocessor implementation as the thread model executes in a round-robin fashion and is useful to explore real-time context switching and memory management capabilities of an RTOS.

The related research in the embedded benchmarking area is pointing to the need of a more specific multicore real-time benchmark suite, capable to instrument performance characteristics of shared memory architectures. The following section introduces and overviews the JetBench benchmark application, an Open-Source tool for real-time, multiprocessor embedded architectures.

3 Benchmark Characteristics

The JetBench application is composed of thermodynamic calculations based on three types of jet engines, i.e. 1) TurboJet, 2) Turbojet with afterburner, and 3) a Turbofan engine (See Fig. 1). The application contains parameters specific to the said models as described in the NASA’s EngineSim application [15]. The benchmark allows a user defined input flight profile to be simulated containing speed, altitude, throttle, and deadline time, while in response to that, the processing time for various thermodynamic calculations is monitored and reported (See Fig. 2).



(a) Turbojet (b) Jet with Afterburner (c) Turbofan

Fig. 1. Three different Jet Models used in JetBench (Adapted from [15])

An overview of the thermodynamic calculations used in the benchmark application is given in Appendix.

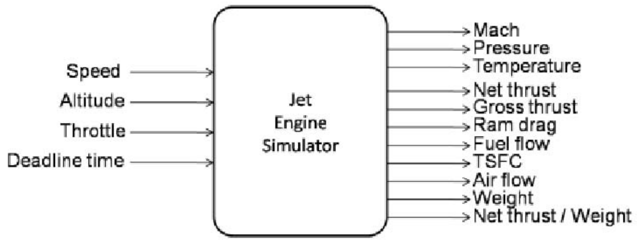


Fig. 2. JetBench Application I/O Parameters

In contrast to a synthetic benchmark, an application benchmark such as JetBench, is a realistic representation of the actual workload; however there are some deviations one has to apply to allow portability of the application on various platforms, which are discussed as follows. Generally, all real applications require a significant amount of I/O operations, which if were implemented in the benchmark would have restricted its portability [16]. Therefore the I/O performance of a platform can not be evaluated through the proposed benchmark. Secondly, as the application has to get executed in a target time period, excessive computations could have caused the benchmark to perform poorly on majority of low end systems. To avoid this problem the JetBench application covers a limited number of typical thermodynamic calculations used in jet engines. As a consequence of the restricted workload of the computations, it may seem small enough to high-end multicore systems that their actual performance may not be reported well, as in contrast to a low end multicore platform. A more detailed analysis of the benchmark on a number of cores is given in the following section.

The JetBench application not only provides the user with an overview of the real-time performance of the system, but could also be used to discover optimum number of threads to achieve desired performance. The JetBench benchmark is mainly comprised of ALU centric operations such as integer/double multiplication, addition, and division for the computation of exponents, square roots, and calculations such as value of pi and degree-to-radian conversion. All these operations are based on real thermodynamic equations and operations required for a jet engine control unit. The benchmark structure is composed of 88.6% of the parallel portion as reported by thread analysis tools, and is described in the pseudo code given in Fig. 3 and the threading diagram in Fig. 4.

```
JetBench Pseudo Code
Inputs:  Engine Type
Data File Defining Speed, Altitude, Throttle, Deadline
Initialization:
            Set Default Parameters
            Select Engine Type
            Open data file
Parallel Section:
            Calculate Pi
            Read an input data point
            Calculate:
                   Environment variables
                   Thermodynamic parameters
                   Engine geometry
                   Engine performance
            Print Results
            If not EOF goto Parallel Section
            Print Results
            End
```

Fig. 3. Pseudo code of the application

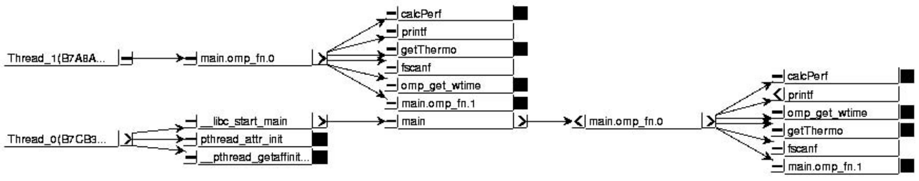


Fig. 4. JetBench Thread Structure

4 Results

To analyze the scalability of the benchmark, un-optimized executions of the application were carried out on shared memory multicore platform based on sixteen x86 CPUs running at 20MHz. The platform was simulated on Simics full system simulator [17], running Linux kernel 2.6.15 including symmetric multiprocessing support. The input dataset comprised of 30 data points and calculation deadlines were uniformly set as 9 sec. As the platform is running at a low clock frequency i.e. 20 MHz, a single thread per core was executed. The benchmark output timing per input data point is shown in Fig. 5. The graph shows normalized timing values against the set deadline time, i.e. 9 sec in this instance, which enables one to compare execution rate instead of execution time. It is worth noting that the execution rate is inconsistent for all the cores, also the rate decreases with the increase in number of cores. The reason behind is that the application is not prioritized statistically by the user but has been prioritized by the kernel itself. Secondly for any application increasing the number of threads beyond a certain level actually decreases performance since thread handling overhead will surpass the per thread execution time. This phenomenon is more observable, when running multiple threads per core where context switches depreciate the performance after a certain level of parallelism.

It can be observed from Fig. 6 and 7 that the overall execution time for the application is around 230 sec for a 4 core machine; however the 8 core machine offers a minimum number of missed deadlines, i.e. 2. This is due to the fact that although for 8 cores platform, threading overhead is higher than for the four core machine, which also effects the computation time per thread. But for 4 cores or less the CPU workload has exceeded the available resource and therefore resulted in missing more deadlines than the later. The output from the benchmark execution thus allows the user to

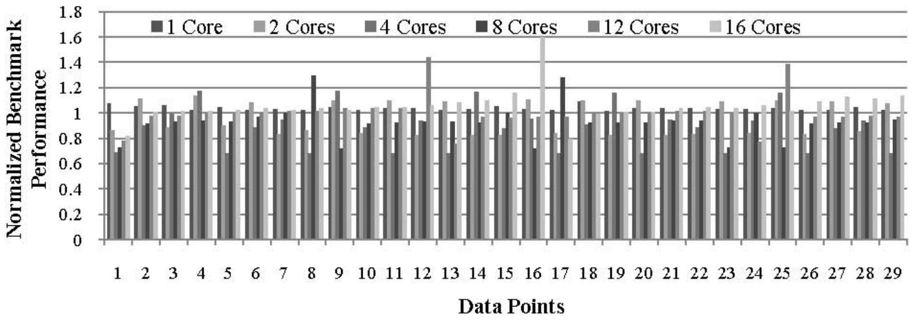


Fig. 5. Application execution rate for different number of cores

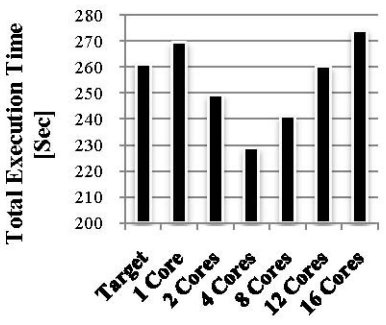


Fig. 6. Application execution time

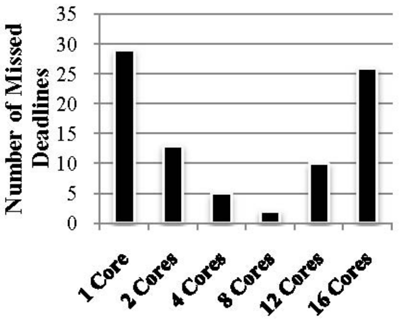


Fig. 7. Missed Deadlines

analyze the impact of threading on a particular platform and could be helpful in the process to decide optimal number of cores as well as OS scheduling characterization.

To validate the phenomenon of performance degradation with an increase in number of threads, a more detailed analysis of the benchmark based on an Intel Dual Core machine [18] was carried out (see Fig. 8). The benchmark is executed for up to 8 threads and processing speedup was calculated using Amdahl's law [19] and Gunther's law (or alternatively termed as Universal Scalability Law (USL))[20-22].

$$Speedup = \frac{1}{(1-p)+\frac{p}{N}}, \quad (1)$$

Amdahl's Law

$$Speedup = \frac{N}{1+s(N-1)+kN(N-1)}, \quad (2)$$

Gunther's Law

where

p = Parallel fraction of the program
s = Serial fraction of the program
k = Delay associated with concurrency
N = Number of processors

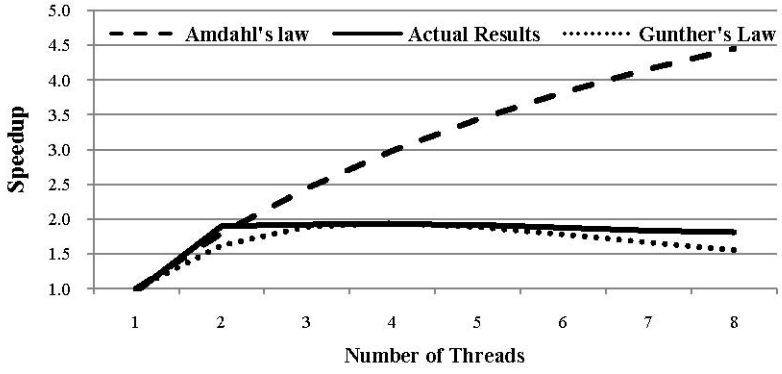


Fig. 8. Comparison of actual speed-up against Amdahl's law and Gunther's law

Amdahl's law is useful in the situations to set an upper limit for the performance gain with increase of parallelization, this however does not take into account the drawbacks of aggressive parallelization such as excessive cache coherency delays, instruction execution, and thread scheduling delays etc. On the other hand Gunther's law provides a more realistic picture in such situations. The results shown in Fig.8 complement the results in Fig. 6, as the throughput tends to decrease with the increase of parallelism beyond a certain limit, which however varies from platform to platform.

5 Conclusion

In this paper, Jetbench an open-source, real-time multicore application benchmark has been presented. The application is designed to be platform independent by avoiding target specific libraries and hardware counters and timers. The application comprises of thermodynamic calculations of a jet engine, and processes user defined input data points with custom deadlines. The benchmark application was tested on a 16 core platform and has demonstrated its usefulness for deciding optimal number of threads, and provided timing information that could be used to deduce an estimate of CPU core utilization and the operating system's real-time behaviour.

Future work will include the testing of the benchmark on various architectures with and without thread prioritization. Also the application's behaviour on an RTOS based platform is to be observed.

JetBench is available from http://jetbench.sourceforge.net/.

References

[1] Morton, G.: MSP430 Competitive Benchmarking. Texas Instruments (2005)
[2] Dagum, L., Menon, R., Inc, S.G.: OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science & Engineering 5, 46–55 (1998)
[3] Drepper, U., Molnar, I.: The native POSIX thread library for Linux. White Paper, Red Hat Inc. (2003)
[4] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22, 789–828 (1996)
[5] Uniejewski, J.: SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter (1989)
[6] Weicker, R.P.: An overview of common benchmarks. Computer 23, 65–75 (1990)
[7] Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Communications of the ACM 27, 1013–1030 (1984)
[8] Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NASA Ames Research Center (1999)
[9] Gal-On, S., Levy, M.: Measuring Multicore Performance. Computer 41, 99–102 (2008)
[10] Leteinturier, P., Levy, M.: The Challenges of Next Generation Automotive Benchmarks. Journal of Passenger Car: Electronic and Electrical Systems 116, 155–160 (2007)
[11] Zadeh, L.A.: Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A. Zadeh, vol. 6. World Scientific, Singapore (1996)

[12] Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., De Michiel, M.: Papabench: a free real-time benchmark. In: 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden, Germany (2006)
[13] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: 4th IEEE International Workshop on Workload Characterization (WWC 2001), Austin, Texas, pp. 184–193 (2001)
[14] Measuring Real-Time Performance of an RTOS: Express Logic Inc.
[15] EngineSim Version 1.7a: NASA Glenn Research Center
[16] Berry, M., Chen, D., Koss, P., Kuck, D., Lo, S., Pang, Y., Pointer, L., Roloff, R., Sameh, A., Clementi, E.: The Perfect Club benchmarks: Effective performance evaluation of supercomputers. International Journal of High Performance Computing Applications 3, 5–40 (1989)
[17] Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. IEEE Computer 35, 50–58 (2002)
[18] Intel, Intel Concurrency Checker v2.1, Intel Corporation (2008)
[19] Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS Joint Computer Conferences, Atlantic City, New Jersey, pp. 483–485 (1967)
[20] Gunther, N.J.: A Simple Capacity Model of Massively Parallel Transaction Systems. In: CMG Conference, San Diego, California, pp. 1035–1044 (1993)
[21] Gunther, N.J.: Guerrilla Capacity Planning: a Tactical Approach to Planning for Highly Scalable Applications and Services. Springer-Verlag New York Inc. (2007)
[22] Gunther, N.J.: Unification of Amdahl's law, LogP and other performance models for message-passing architectures. In: International Conference on Parallel and Distributed Computing Systems, Phoenix, AZ, pp. 569–576 (2005)

Appendix: Thermodynamic Equations

With reference to the Fig. 1, thermodynamic calculations [15] covered in the benchmark are given as follows.

Notations:

- Point 0 is the free stream conditions
- Point 1: the inlet entrance
- Point 2: compressor entrance
- Point 3: compressor exit
- Point 4: turbine entrance
- Point 5: turbine exit
- Point 6: nozzle throat

Inlet performance (0->2)	
$\frac{Tt_2}{Tt_0} = 1$	Inlet Temperature ratio
$\frac{pt_2}{pt_0} = \eta_i$	Inlet Pressure ratio for Mach < 1, where η_i is the inlet efficiency factor
$\frac{pt_2}{pt_0} = \eta_i(1 - 0.075[M - 1]^{1.35})$	Inlet Pressure for Mach > 1, where η_i is the inlet efficiency factor
$D_{spill} = K(\dot{m}_i[V_1 - V_0] + A_1[p_1 - p_0])$	Spillage Drag for inlet, where **K** is the lip suction factor, $\dot{\mathbf{m}}_\mathbf{i}$ is the inlet mass flow rate, **V** is the velocity, **A** is the area, and **p** is denoting the pressure.
Compressor thermodynamics (2->3)	
$CPR = \frac{pt_3}{pt_2} \geq 1.0$	Compressor Pressure ratio
$\frac{Tt_3}{Tt_2} = \left(\frac{pt_3}{pt_2}\right)^{(\gamma-1)/\gamma}$	Compressor Temperature ratio, where γ is the ratio of specific heats
$CW = \frac{c_p Tt_2}{\eta_c}\left[CPR^{(\gamma-1)/\gamma} - 1\right]$	compressor work per mass of airflow, where η_c is the compressor efficiency factor and c_p the specific heat
Burner thermodynamics (3->4)	
$BPR = \frac{pt_4}{pt_3} = 1$	Burner Pressure Ratio
$\frac{Tt_4}{Tt_3} = \frac{1 + \frac{f.\eta_b.Q}{cp.Tt_3}}{1 + f}$	Burner Temperature Ratio, where , f is the fuel to air mass flow ratio, Q is the heat release, η_b is the burner efficiency factor.

Turbine thermodynamics (4->5)	
$TPR = \frac{pt_5}{pt_4} = \frac{Tt_5 5^{(\gamma-1)/\gamma}}{Tt_4}$	Turbine Pressure Ratio
$TW = \eta_t c_p Tt_4 \left(1 - TPR^{(\gamma-1)/\gamma}\right)$	Turbine Work Per Mass Of Airflow, where ηt is the turbine efficiency and c_p is the specific heat.
Nozzle thermodynamics (5->6)	
$NPR = \frac{pt_8}{pt_5} = \left(\frac{Tt_8}{Tt_5}\right)^{\gamma/(\gamma-1)} = 1$	Nozzle Pressure and temperature ratios
$V_e = V_8 = \sqrt{2c_p Tt_8 \eta_n [1 - \{1/NPR\}^{(\gamma-1)/\gamma}]}$	Exit velocity, where η_n is the nozzle efficiency
Output calculations	
$M = \frac{V_0}{a_0} = \frac{V_0}{\sqrt{\gamma R T_0}}$	Mach Number, where V_0 is the aircraft speed, a_0 is the speed of sound and R is the gas constant
$ts_0 = 518.6 - 3.56 \frac{altitude}{1000}$, $ps_0 = 2116. \frac{ts_0}{518.6}^{5.256}$.	Stratospheric Temperature and pressure for altitude < 36152 feet
$ts_0 = 389.98$, $ps_0 = 473.1376 * e^{(36000-altitude)/20805.433}$	Stratospheric Temperature and pressure for 36152< altitude < 82345 feet
$ts_0 = 391.625 . \frac{altitude-82345}{1000}$, $ps0 = 51.96896 * \left(\frac{ts_0}{389.98}\right)^{-11.388}$.	Stratospheric Temperature and pressure for altitude > 82345 feet

$\dot{m}_f = f\dot{m}_a$ where, $f = \frac{\left(\frac{Tt_4}{Tt_3}\right)-1}{(\eta_b Q/c_p Tt_3)-(Tt_4/Tt_3)}$.	Fuel mass flow rate, where $\dot{m}_a$ is the airflow rate, η_b is the burner efficiency, f is the fuel to air ratio and Q is the fuel heating value.
$EPR = \frac{pt_8}{pt_2} = \frac{pt_3}{pt_2}.\frac{pt_4}{pt_3}.\frac{pt_5}{pt_4}.\frac{pt_8}{pt_5}$, $ETR = \frac{Tt_8}{Tt_2} = \frac{Tt_3}{Tt_2}.\frac{Tt_4}{Tt_3}.\frac{Tt_5}{Tt_4}.\frac{Tt_8}{Tt_5}$, $Tt_8 = Tt_2.ETR$, $pt_8 = pt_2.EPR$, $Tt_0 = Tt_2 = T_0(1 + 0.5\,[\gamma - 1]\frac{V_0{}^2}{a_0{}^2})$, $pt_0 = p_0\frac{Tt_0^{\gamma/(\gamma-1)}}{T_0}$.	Thrust Specific Calculations where, ***EPR*** is the engine pressure ratio, and ***ETR*** is the engine temperature ratio.

A Tightly Coupled Accelerator Infrastructure for Exact Arithmetics

Fabian Nowak and Rainer Buchty

Chair for Computer Architecture
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
{nowak,buchty}@kit.edu

Abstract. Processor speed and available computing power constantly increases, enabling computation of more and more complex problems such as numerical simulations of physical processes. In this domain, however, the problem of accuracy arises due to rounding of intermediate results. One solution is to avoid intermediate rounding by using exact arithmetic. The use of FPGAs as application-specific accelerators can speed up such operations compared to their software implementation.

In this paper, we present a system approach employing state-of-the art FPGA and interconnection technology for exact arithmetic with double-precision operands, delivering up to 400M exact MACs/s in total and providing a speedup of up to 88 times over competing software implementations in the case of matrix multiplication.

1 Introduction

With the computation of increasingly complex problems, accuracy issues arose related to rounding effects taking place in current FPU implementations. These may lead to unsatisfying results and even physical damage, if upfront simulations do not indicate certain problems. This problem is typically addressed by certain exact arithmetics built into mathematics and simulation packages. Such arithmetics increase the width of the internal data representation or concatenate several floating-point numbers in order to enlarge the "accuracy window" in which no rounding is necessary. While these software implementations provide a viable workaround, they are magnitudes slower than native hardware support. Custom accelerator hardware can speed up such operations using several design techniques, e.g. pipelined execution and parallelization.

We therefore present an accelerator system for exact arithmetics. Based on a careful examination of the underlying algorithm for implementing exact arithmetics, a hardware solution employing pipelining techniques and exploiting parallelism is proposed delivering up to 400M exact MAC operations per second.

In the remainder, we first outline related work in Section 2. We then present our general architecture design in Section 3 before discussing implementation issues and viable solutions in Section 4. The elaborated design is thoroughly evaluated in Section 5. We conclude this paper by summing up the results and presenting our plans for future work in Section 6.

2 Related Work

Performing floating-point operations in software is quite costly, therefore many FP accelerators exist. Such accelerators were originally the domain of dedicated silicon, but with increasing FPGA capabilities, these gained attention as computation accelerators. In [1], for example, double-precision operations are employed for accelerating physics simulations. Other work focuses on exploiting FPGAs for matrix multiplications with double precision [2], or on the conjugate gradient method [3,4,5].

The problem of exact computation is well-understood, and hence multiple solutions exist. For certain computations, the effect of errors can be evaluated upfront, allowing the use of lower-precision computation [6]. For computations where knowledge of the included error is required, using so-called Staggered Interval Arithmetics [7] (SIA) is an option. The precision of floating-point operations can also be enhanced by increasing mantissa and exponents as e.g. provided by the GNU MP library.

Our work is motivated by the work of Kulisch et al. [8],[9], who developed a concept for exact arithmetics support in regular floating-point units and also implemented a PCI-attached coprocessor for exact arithmetics. Following Kulisch's work, a straight-forward implementation supporting single-precision operands demonstrated the general applicability to FPGA technology, but did not yet offer speedup in comparison to software implementations [10].

In order to further support the use of such accelerators and to ease benchmarking, we developed a runtime system enabling dynamic switching from conventional to exact arithmetics implemented in software or hardware [11]. Dynamically applying exact arithmetics only where required, results in the fastest total runtime while still guaranteeing numerical robustness and therefore delivering precise results.

3 Architecture and Design

The need for exact arithmetics in numerical programs arises in most cases for matrix multiplications. Thus, we give a detailed description of how matrix multiplication can be split and organized in the optimal way for data transfers and parallel pipelined processing in this section.

3.1 Exploiting Matrix Multiplication Properties

Matrix multiplications can be regarded as consisting of matrix-vector multiplications that are simply a series of inner vector products:

$$C = A \cdot B = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot \begin{pmatrix} b_1 \cdots b_o \end{pmatrix} = \begin{pmatrix} A \cdot b_1 \cdots A \cdot b_o \end{pmatrix} = \begin{pmatrix} a_1 \bullet b_1 & \cdots & a_1 \bullet b_o \\ \vdots & \ddots & \vdots \\ a_m \bullet b_1 & \cdots & a_m \bullet b_o \end{pmatrix} \quad (1)$$

where A is an $m \times n$ matrix with rows a_1 to a_m, B is an $n \times o$ matrix with columns b_1 to b_o, C is an $m \times o$ matrix, and $\bullet$ denotes the inner product.

When looking at Equation 1, we easily can see that for data reuse we need to stream in one of the two matrices, while reading the other from local memory, because row a_i is used for computing the complete row i of C. This allows to stream the rows of A and read R vectors of the local matrix B concurrently element by element according to the column of A. Hence, we obtain R elements of C at once after all the n elements have been accumulated. Note that reading more than one element of a vector b_j is not useful as only one product can be computed and accumulated per cycle. Thus, individual EAUs have to be used in parallel in order to gain speedup.

However, not only row a_i can be reused, but the same works for the vectors b_j of B as well: having enough bandwidth available or running the different components at different clock speeds allows to receive more than one value of a_i. This can be exploited to stream in $S > 1$ rows of A concurrently in favor to delivering several elements of a_i because of the same reason as mentioned above. As a result, the vectors b_j are the same for all multipliers, while each "plane" is given different rows a_i of matrix A.

In the obvious approach, the elements of A and B are being sent one by one or packed into tuples $(a_{i,k},\ b_{j,k})$. In that case, no additional overhead due to data organization occurs. When multiplying several vectors at once in the parallelized approach, we read the k^{th} element of R vectors, i.e. of elements $b_{1,1}, \ldots,\ b_{R,1}$, $b_{1,2}, \ldots,\ b_{R,2}, \ldots,\ b_{1,k}, \ldots,\ b_{j,k}, \ldots,\ b_{R,k}, \ldots,\ b_{R,n}, \ldots,\ b_{R+1,1}, \ldots,\ b_{o,n}$, where the second index denotes the row of column vector b_j. Thus, B should be transferred into DDR memory in such a way that subsequent memory accesses happen in exactly this order. In case $R = o$, nothing special has to be done, otherwise the matrix has to be reordered. Finally, for the 2-dimensional approach, the elements of matrix A are accessed rather row by row than column by column. Hence, transposing A is already helpful, but as in the second case, data has additionally to be reordered with respect to block size S that denotes the number of "planes", i.e. the number of elements that are transferred per cycle and number of rows a that are processed in parallel.

3.2 FPGA Integration

Following Sect. 3.1, it is advisable to store one of the two matrices to be multiplied, B, in on-board memory, enabling concurrent computation of different columns of the result matrix C. This requires transferring matrix B from main memory to FPGA memory. Additional control logic applies for controlling initialization and multiplication, resulting in the overall architecture depicted in Fig. 1 where the user logic is clocked with 130 MHz and DDR memory with double the frequency, i.e 260 MHz, as required by the DDR controller.

Our hardware environment consists of the UoH HTX Board [12] containing a Xilinx Virtex-4 FPGA. The board is located in a host PC using the Hyper-Transport (HT) bus. Resulting from this hardware, DDR memory is clockable at a maximum of 266 MHz and HT either at 200 MHz or at 400 MHz.

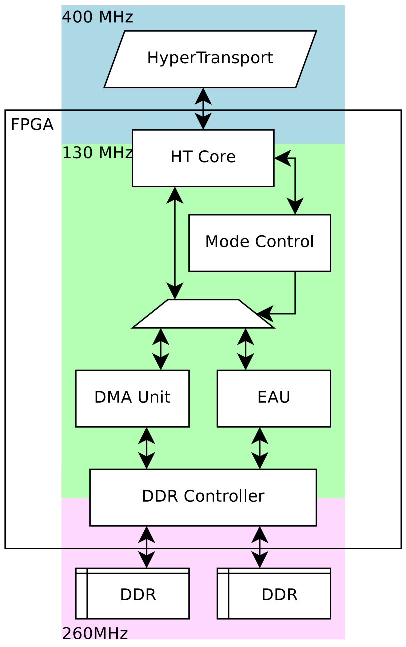


Fig. 1. Integration of the EAU in the UoH HTX Board

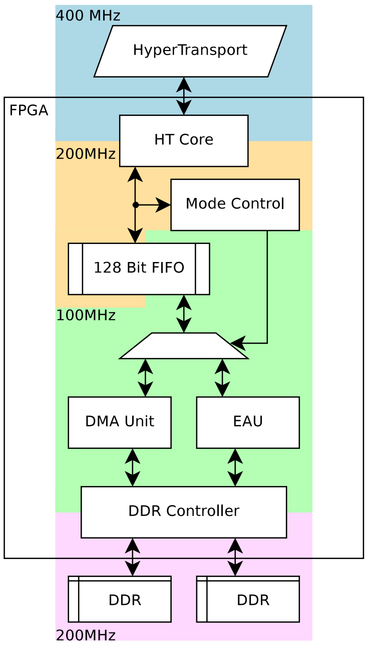


Fig. 2. Enhanced integration of the EAU with intermediate buffering

In order to better exploit the HT-provided bus bandwidth, buffer circuitry can be applied leading to the architecture sketched in Fig. 2, where the buffers decouple the 200 MHz HT backend from the 100 MHz user logic, which in return decreases the DDR memory's frequency to 200 MHz. Such an architecture is expected to perform better due to operating near full bandwidth usage in contrast to the former.

4 Implementation

4.1 Multiplier

The architecture is designed in conformance with IEEE-754, which is addressed as follows: by storing the results of double-precision multiplications in 106-bit registers, overflows and underflows do not occur. As the input data is sent from software applications with their own safety checks included, we assume that no NaN, explicit underflow or infinity is being sent as input data.

When designing and implementing the multiplier, special focus has been put onto high synthesizable frequency. By splitting the multiplication of the 53 mantissa bits into 14 different fully pipelined stages, we achieved a frequency rate of 176 MHz and a low resource foot print when synthesizing. The results for the employed Xilinx Virtex-4 FX100-11 FF1152 are listed in Table 1.

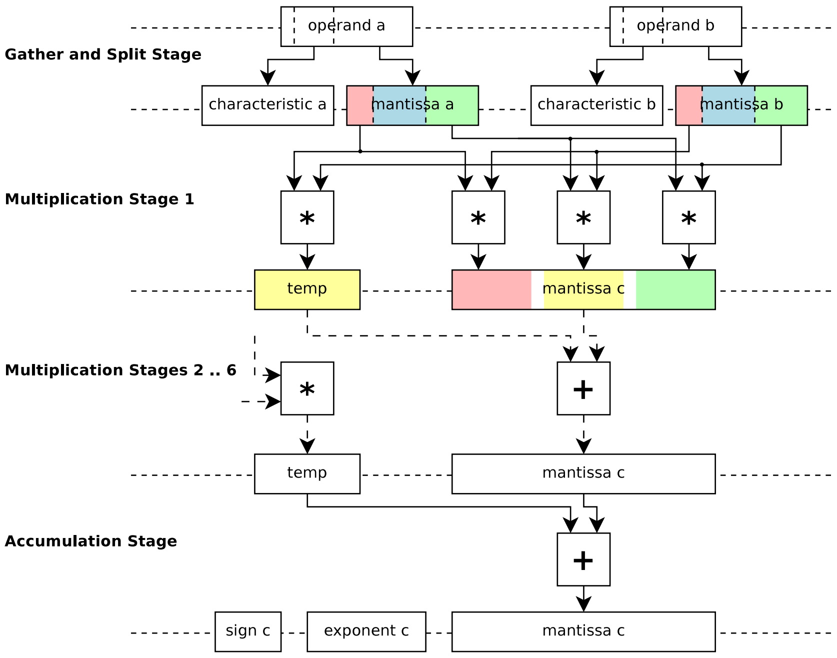


Fig. 3. Simplified sketch of a double-precision multiplier implementation

The implementation follows the scheme depicted in Fig. 3: first, the operands are split into sign, characteristic, and extended 53-bit mantissa in the first gather-and-split stage where the mantissa is split into k blocks requiring k^2 multiplications. We then execute non-overlapping multiplications in parallel, thus initially populating the first pipeline register carrying the result. An additional multiplication can be executed in parallel, but as it overlaps with the other partial results, it must be accumulated onto the mantissa pipeline register in the next stage where the partial product is calculated in parallel. This step repeats for the remaining partial products, resulting in a total of 6 multiplication stages when employing hypothetical 19-bit multipliers. In the final step, the last product has to be accumulated, the sign and new exponent have to be passed.

As the FPGA hardware provides a DSP width of 18 bits with 1 bit used for the sign, we have to divide the mantissa into four blocks of length 13 and 14 bits,

Table 1. Resource usage of the multiplier after synthesis

Resource	Usage	Percentage
Slices	1,946	4%
Flip Flops	2,669	3%
4-Input LUTs	2,874	3%
DSP48s	16	10%

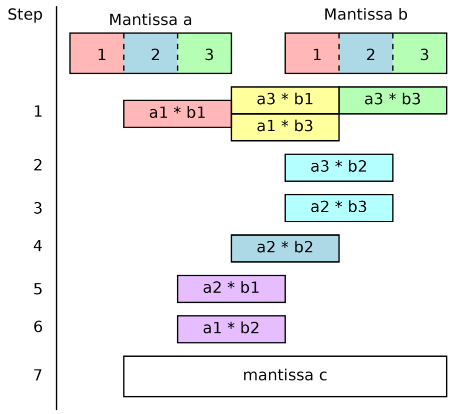


Fig. 4. Simplified sketch of the multiplication pipeline assuming a DSP width of 19

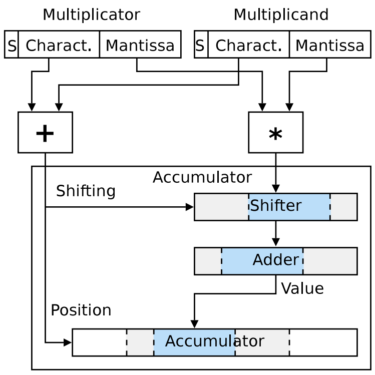


Fig. 5. Product of two operands shifted and added to the correct position of the long accumulator

respectively, resulting in 16 multiplications totally, with the inner steps being 12 multiplication stages, instead of 6 as in the implementation scheme. Note that the exponent requires one more bit, i.e. 12 bits, now, and the mantissa requires twice the amount of bits in contrast to the original operand mantissa, i.e. $2 * 53$ bits $= 106$ bits. In order to report infinity, a minor amount of logic has been added, which sets all exponent bits to '1' to signal infinity. Figure 4 visualizes the pipelining of the multiplication according to Fig. 3 and the former description.

4.2 Accumulation Unit

The unmodified result of the multiplication, i.e. unshorted and with extended exponent, is handed to the accumulation unit, which shifts the product, determines the position where the product is to be added onto the large accumulation register, and finally adds the shifted product to these positions, as is shown in Figure 5. This way, no loss of precisions with regard to the input data happens. The accumulation register is capable of accurately storing the products of both largest and smallest numbers by value with a total width of $64 * 67 = 4288$ bits; additional space is used to avoid overflows (c.f. [13]).

The accumulation unit was designed as a pipelined unit. Data is accumulated in several steps and track of the blocks' values is kept in order to resolve any occurring carries [13]. The flow of operation within this pipeline involves an atomic read-modify-write operation. In hardware, this can either be addressed as one massive pipeline stage limiting the overall pipeline frequency, or a more fine-grained pipeline where after each add/subtract command an atomic read-modify-write scheme is executed, stalling the pipeline for the time of execution. For the sake of reaching a maximum clock rate, we chose the latter approach leading to a latency of 3 cycles for add/subtract operations.

Table 2. Resource usage of the accumulation unit after synthesis

Resource	Usage	Percentage
Slices	17,094	40%
Flip Flops	5,808	6%
4-Input LUTs	32,481	38%
DSP48s	1	1%

Table 3. Resource usage of multiplication and accumulation units after synthesis

Resource	Usage	Percentage
Slices	17,838	42%
Flip Flops	8.449	10%
4-Input LUTs	34,207	40%
DSP48s	16	10%

After accumulation, the pipeline determines the range where valid data is stored in the accumulator, reads its value and converts it to a double-precision floating-point value. Due to the two stall cycles, we can read the accumulator value two cycles after it has been written. The additional approach of assigning each operation an individual ID allows tracking intermediate results, as will be explained in more detail in Sect. 4.3. Table 2 shows the resource usage of the accumulation unit, Table 3 of the combination with a multiplier to one design without additional control or interface instances.

4.3 Communication Principles and Controller

As already indicated above, the accumulator is not only given mantissa and exponent, but also an identification number (ID) and the operation itself. The `CLEAR` operation resets the internal state and accumulator; the accumulator can then be used for the next series of exact MAC operations. For each operation, it is also necessary to indicate whether the command is validin order to separately control the multipliers and accumulator units.

The second part of the communication occurs from the accumulator to its data input site, indicating that it is ready for data (`rfd`). There is a delay of one cycle between the input of an operation and the `rfd=0`, which requires to buffer the subsequent command if indicated as valid operation for automatic processing after the current operation.

4.4 Data Transfers

To begin with, data of one of the matrices has to be brought onto the HTX board's DDR RAM. After successful completion, operation can commence: data needs to be streamed via DMA from host system memory to the HTX board, where it is multiplied and accumulated with the data residing in the board-local memory. Finally, the accumulated result has to be converted to double-precision floating-point format and brought to the host-system, where it can be processed further (Figs. 1, 2).

With the proposed design in mind, the central control instance becomes responsible for controlling DDR RAM operations, especially with regard to initialization and burst transfers, for mode control where mode is one of initialization, in/out transfer phase, regular operation with read accesses for the operands and possibly write accesses for the results, and for future extensions that modify individual blocks of the accumulation units.

From the client side, it is important that after requesting a medium-grained operation (inner product, matrix multiplication) the relevant data is sent. This ideally happens by writing operand data to distinct memory-mapped addresses of the HTX board while respecting the data order according to Sect. 3.

Recalling the capabilities of the hardware platform and the resulting mode of operation, reading the accumulated result can be accomplished in one of two basic ways: explicitly offered `read` operations in the accumulator that are started upon read requests on the HyperTransport posted queue, and streamed, uninterrupted write-out of the accumulator to main memory. Explicit read operations in the accumulator pipeline violate the concept of pipelining, making it even more complicated and potentially decreasing maximum execution speed. The second approach, i.e. to always compute the accumulator's value in double-precision and to uninterruptedly send these values to the system's main memory, using the IDs as least-significant parts of the address, looks fine at first sight. While it does not interfere with the read bandwidth of the EAU on the HTX board, it however does not allow to send the results of more than 3 EAUs, as each EAU produces a new result each 3 cycles.

To circumvent the afore-mentioned limitations, we suggest to combine the two approaches: each EAU converts its accumulation values and returns them together with the IDs that led to these results. A separate controller cares for incoming read requests, buffers the read address that contains the accumulation unit and the ID, and returns the next value of the specified EAU that has the same ID. This task perfectly fits the controller required for initializing the local memory. Still, there is the choice between continuing execution while waiting for the result, or rather stalling the accumulation unit. When choosing the latter, no problems due to subsequent modifications can occur. While the number of bits for the ID has already been set to 4 for a sufficient amount of flexibility, the number of bits for different EAUs could be chosen arbitrarily. According to our experience with a single-precision implementation [10], we can say that a total of 16 EAUs will be enough, so that 4 bits of the address should suffice for choosing a specific EAU. This scheme can be enhanced further by having the controller automatically write back the final results of each completed accumulation.

4.5 Putting It All Together

In Sect. 4.2, we saw that the accumulation unit accepts operands each 3 cycles due to otherwise conflicting accesses. By demultiplexing the products of the multipliers onto different accumulation units as in Figure 6, we can circumvent these disadvantages, allowing to compute uninterruptedly. When we even consider that upon each cycle we can obtain a value of the left input matrix A, we come up with a throughput of 3 exact MAC operations per cycle, which is 300M exact MAC operations per second (eMACs/sec) when running at a frequency of 100 MHz.

Table 4. Resource usage of multiplication and accumulation units after mapping process. Percentages are given for the Xilinx Virtex-4 FX100.

	1 Mult, 1 Accu		2 Mults, 2 Accus	
Resource	**Usage**	**Percentage**	**Usage**	**Percentage**
Occupied Slices:	19,973	47%	36,315	86%
Slices containing only related logic:	19,973	100%	36,315	10%
Slices containing unrelated logic:	0	0%	0	0%
Total Number of 4 input LUTs:	34,336	40%	68,892	81%
Number used as logic:	33,701		61,480	
Number used as a route-thru:	551		7,248	
Number used as Shift registers:	84		184	
Number of BUFG/BUFGCTRLs:	1	3%	2	6%
Number used as BUFGs:	1		2	
DSP48s:	16	10%	32	20%

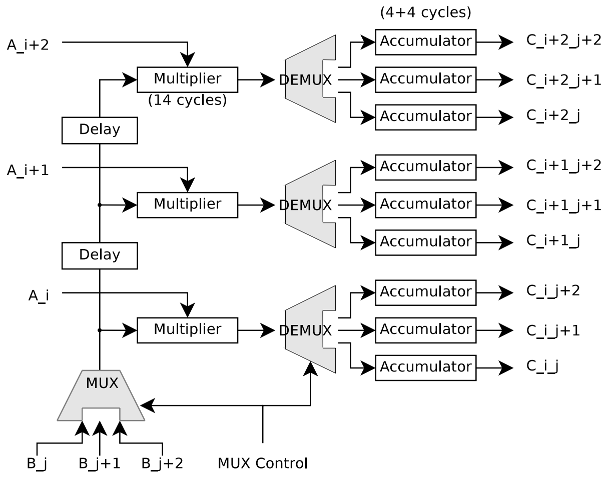


Fig. 6. Demultiplexing multiplier output onto accumulation units when multiplying rows i to $i+2$ of matrix A with columns j to $j+2$ of Matrix B. The iterator index k is not shown.

5 Evaluation

5.1 Performance Estimations

We present the final formula for the overall execution time in Table 5. Here, bw denotes the bandwidth to board memory, f the user frequency. The execution time is composed of transfer to memory, streamed computation, and transfer from memory back to the host system. Note that the transfer to memory is limited by the HTX bandwidth bwh rather than by the memory bandwidth bwr.

Table 5. Overview of the different components involved in maximum bandwidth usage depending on matrix dimensions n, m, o, throughput tp, and transport bandwidth bw

Transfer to RAM	$n * o * 8\text{B}/bw$
Transfer from RAM	$m * o * 8\text{B}/bw$
Computation	$m * n * o/tp$
Overall time	$(m+n) * o * 8\text{B}/bw + m * n * o/tp = (2m + m * n + 2n) * o/tp$
130 MHz Design	$(2m + m * n + 2n) * o/260M/s$
100 MHz Design	$(2m + m * n + 2n) * o/400M/s$

Table 6. Best-case execution times of two different architectures for exact arithmetics on UoH HTX board

Dimensions	130 MHz	100 MHz
1000	3.86 sec	2.51 sec
4000	246 sec	160 sec
8000	3847 sec	2501 sec

Table 7. Comparison between 256-Bits GMP, C-XSC, and double precision (DP), runtime in clock ticks

	GMP	C-XSC	DP
Runtime (ticks)	566,480,613	640,445,526	6,403,510
Runtime (μs)	177,089	200,211	2,001
M eMACs/sec	5.921	5.237	523.810[1]

[1] *see text.*

Table 6 provides numbers for different matrix dimensions $m = n = o$ of 1000, 4000, and 8000, which would result in matrix sizes of 8 MB, 128 MB, and 512 MB, respectively.

Following Tables 5 and 6, we see a preference for the 100 MHz design; in contrast to the 130 MHz design, however, performance tightly depends on the streamed data and puts more pressure on the data transport layer of the hardware design.

Due to resource constraints on the chosen FPGA and due to the chose implementation of the exact accumulation unit, 2 accumulation units and one multiplier are the maximum number of resources to deliver maximum throughput of 2 eMACs each 3 cycles, i.e. 88M eMACs/sec for the 130 MHz design (Fig. 1) and 66M eMACs/sec for the 100 MHz design (Fig. 2), respectively.

5.2 Technology Scaling

As the architecture and current implementation do not look very promising, we conduct a technology scaling onto the Xilinx Virtex-6 LX240T, which is available for example on the Xilinx ML605 evaluation board. This FPGA provides enough space for 6 exact MAC units, so that at least 2 exact MAC operations can be started per cycle. Furthermore, due to the available increased logic speed, some pipeline stages of the time-critical read-modify-write scheme may be mergable, resulting in even 3 exact MAC operations, and hence in a maximum throughput rate of 300M eMACs/sec. Besides, the wider DSP units allow to reduce the multiplier pipeline length to less than 12 multiplication stages; the look-up tables with 6 inputs in contrast to 4 inputs might further reduce area requirements.

5.3 Comparison

In order to evaluate our architecture, we performed 2^{20} multiply-accumulates of the value $v = 2^{-20}$ onto an init value of 2^{20} using our hardware and two software solutions. For the latter, we employed an implementation based on the GNU Multiprecision Library (GMP) using a 256 bits wide internal floating-point representation and one based on C-XSC where a 4288 bits wide fixed-point accumulator is employed. Individual runtimes were measured on an Intel Pentium 4 3,2 GHz over 128 runs. All programs were compiled with `-O2`. Subtracting the initialization value afterwards we obtained the correct result 2^{-20}, which fails when using regular double-precision floating-point operations.

The measurements can be found in Table 7. We additionally provide the execution time with regular double-precision, which is about 100 times less compared to the highly accurate libraries. Our architecture concept is capable of executing exact multiply-accumulate operations in about the same order of time.

Based on the maximum achievable throughput for HT400, we can derive a theoretical maximum speedup of $\frac{400}{5.237} = 76.38$ over state-of-the-art software implementations of exact arithmetics. Following Sect. 5.1, limitations of the target platform still allow a maximum speedup of $\frac{88}{5.237} = 16.80$.

6 Results and Future Work

As of now, our system allows processing two MAC operation each 3 cycles while running at a frequency of 100 MHz, which leads to a theoretical processing speed of 66M MAC operations per second. Note that these operations are carried out in exact arithmetics, avoiding rounding and windowing errors. We showed in Section 3 how a multitude of accumulation units can help to constantly process streamed data and how using the DDR memory can further speed up the overall performance up to 400 MAC operations per second. A technology scaling showed that this architecture is a viable approach to exploit the available bandwidth of the HyperTransport bus. For better results, the implementation of the accumulator unit will be revised, promising a latency of 1 and even less resource usage.

Our future work consists of integrating the DDR memory and DMA controllers in order to carry out detailed measurements with numerical software where computation with exact arithmetics enables solving more complex problems. We also plan to extend our system infrastructure by numerical error measurements so that functions be directed to their exact arithmetics implementation automatically, sacrificing speed over robustness and numerical stability.

Similar to [1], the approach will be extended towards microprogramming capabilities so that (parts of) algorithms can be completely loaded onto the accelerator, thus supporting sparse matrix multiplications. Finally, the architecture is designed to deliver coarse-grained function acceleration of numerical algorithms such as the Lanczos algorithm.

References

1. Danese, G., Leporati, F., Bera, M., Giachero, M., Nazzicari, N., Spelgatti, A.: An Accelerator for Physics Simulations. Computing in Science and Engineering 9(5), 16–25 (2007)
2. Kumar, V.B.Y., Joshi, S., Patkar, S.B., Narayanan, H.: FPGA Based High Performance Double-Precision Matrix Multiplication. In: VLSID 2009: Proceedings of the 2009 22nd International Conference on VLSI Design, Washington, DC, USA, pp. 341–346. IEEE Computer Society, Los Alamitos (2009)
3. DuBois, D., DuBois, A., Boorman, T., Connor, C., Poole, S.: An Implementation of the Conjugate Gradient Algorithm on FPGAs. In: Pocek, K.L., Buell, D.A. (eds.) FCCM, pp. 296–297. IEEE Computer Society, Los Alamitos (2008)
4. Morris, G.: Floating-Point Computations on Reconfigurable Computers. In: HPCMP-UGC '07: Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference, Washington, DC, USA, pp. 339–344. IEEE Computer Society Press, Los Alamitos (2007)
5. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006), April 2006, pp. 259–268 (2006)
6. Herbordt, M., Sukhwani, B., Chiu, M., Khan, M.A.: Production Floating Point Applications on FPGAs. In: Symposium on Application Accelerators in High Performance Computing, SAAHPC 2009 (July 2009)
7. Kulisch, U.W.: Complete Interval Arithmetic and Its Implementation on the Computer. In: Cuyt, A., Krämer, W., Luther, W., Markstein, P. (eds.) Numerical Validation in Current Hardware Architectures. LNCS, vol. 5492, pp. 7–26. Springer, Heidelberg (2009)
8. Bierlox, N.: Ein VHDL Koprozessorkern für das exakte Skalarprodukt. PhD thesis, Universität Karlsruhe (November 2002), `http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1053`
9. Kirchner, R., Kulisch, U.: Accurate arithmetic for vector processors. Journal of Parallel and Distributed Computing 5(3), 250–270 (1988)
10. Nowak, F., Buchty, R., Kramer, D., Karl, W.: Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics. In: Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA 2009), February 2009, pp. 20–29. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg (2009)
11. Buchty, R., Kramer, D., Kicherer, M., Karl, W.: A Light-Weight Approach to Dynamical Runtime Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures. In: Berekovic, M., Müller-Schloer, C., Hochberger, C., Wong, S. (eds.) ARCS 2009. LNCS, vol. 5455, pp. 60–71. Springer, Heidelberg (2009)
12. Fröning, H., Nüssle, M., Slogsnat, D., Litz, H., Brüning, U.: The HTX-Board: A Rapid Prototyping Station. In: Proceedings of the 3rd Annual FPGA World Conference (2006)
13. Kulisch, U.W.: Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units. Springer, Secaucus (2002)

Optimizing Stencil Application on Multi-thread GPU Architecture Using Stream Programming Model

Fang Xudong, Tang Yuhua, Wang Guibin, Tang Tao, and Zhang Ying

National Laboratory for Parallel and Distributed Processing, School of Computer
National University of Defense Technology, Changsha, Hunan, China
fangxudong850403@hotmail.com, yhtang@163.com,
{wgbljl,tt.tang84,minilimn}@gmail.com

Abstract. With fast development of GPU hardware and software, using GPUs to accelerate non-graphics CPU applications is becoming inevitable trend. GPUs are good at performing ALU-intensive computation and feature high peak performance; however, how to harness GPUs' powerful computing capacity to accelerate the applications in the field of scientific computing still remains a big challenge. In this paper, we implement the whole application *Mgrid* taken from Spec2000 benchmarks on an AMD GPU and propose several optimization strategies for stencil computations in the naive GPU code. We first improve thread utilization through using vector types and multiple output streams mechanism provided by the Brook+ programming language. By tuning thread granularity, we try to hit the right balance between locality within each thread and parallelism among threads. Then, we reorganize the stream layout by transforming the 3D data stream into the 2D stream in the block manner. Through stream reorganization, more data locality in the cache is exploited. Further, we propose branch elimination to convert control dependence to data dependence, catering to GPUs' powerful ALU-intensive processing capability. Finally, we redistribute computations between CPU and GPU to make more advisable computing resources usage considering different problem sizes. We demonstrate the effectiveness of our proposed optimization strategies on an AMD Radeon HD4870 GPU using the Brook+ programming language. Using a double-precision floating-point implementation, the experimental results show that the optimized GPU version of *Mgrid* gains 2.38x speedup compared to the naive GPU code and obtains as high as 15.06x speedup versus the CPU implementation run on an Intel Xeon E5405 CPU.

Keywords: Optimization; GPU; Stream; Brook+; Stencil.

1 Introduction

In recent years, GPU has integrated an increasing number of transistors on the chip, which enables a huge leap in its floating computing capacity. New GPU architectures provide easier programmability and increased generality, abstracting away trivial graphics programming details, i.e., Brook+ [1] and CUDA [2]. Therefore, people begin to harness the tremendous processing power of GPUs for non-graphics applications. Now GPU computation has been widely adopted in the field of scientific computing, such as biomedicine, computational fluid dynamics simulation, and molecular dynamics simulation [3].

GPUs were originally developed for graphics processing, i.e., media applications, which have less data reuse and lay more emphasis on real time. However, in scientific applications, there can be rich data reuse while the data access patterns may not be as uniform as media applications. There is much space left for programmers to optimize the GPU codes to exploit more data reuse and hide long memory access latencies. Therefore, besides choosing a good CPU-to-GPU application mapping, we should also try to optimize the GPU codes according to the architecture of the GPU and the mechanisms provided by the programming language.

In this paper, we have implemented and optimized *Mgrid*, a multi-grid application commonly used to solve partial differential equations (PDEs) taken from Spec2000, on an AMD GPU platform. At the heart of *Mgrid* are stencil (nearest-neighbor) computations in each 27-point 3D cube. Stencil computations feature abundant parallelism and low computational intensity which offers great opportunity for optimization in temporal and spatial locality, making them effective architectural evaluation benchmarks [4]. To optimize the naive GPU code, we have proposed four optimization strategies:

(a) Improve thread utilization. Using vector types and multiple output streams provided by the Brook+ programming language, we exploit data reuse within each thread and parallelism among threads to achieve better thread utilization.

(b) Stream reorganization. We reorganize the 3D data stream into the 2D stream in the block manner to catch more data locality in the GPU cache. Stencil computations of *Mgrid* refer data on three consecutive planes when calculating a grid point. Through stream reorganization, we exploit the data reuse within each plane.

(c) Branch elimination. We propose branch elimination to reduce the performance loss caused by branch divergences in the *Interp* kernel. Though changing the control dependence to data dependence, all of the eight branches in the kernel are eliminated, thus improving the kernel performance significantly.

(d) CPU-GPU workload distribution. To make full use of the CPU-GPU heterogeneous system, we should reasonably distribute the workload between the two computing resource according to the work nature, problem size and communication overhead.

Note that though our optimizations are developed for *Mgrid*, they can be applied to any stencil-like computations, making our optimization approaches general for developing GPU applications. In our work, all the experiments are done under double-precision floating-point implementation. The experimental results show that the optimized GPU implementation of *Mgrid* gains 2.38x speedup compared to the naive GPU version and obtains as high as 15.06x speedup versus the CPU implementation run on an Intel Xeon E5405 CPU.

The remainder of this paper is arranged as follows: Section 2 describes the background information of programming with Brook+ on AMD Radeon HD4870. Section 3 illustrates our optimization strategies. Section 4 evaluates the effectiveness of the strategies. Section 5 discusses related work and the final section states our conclusions.

2 Background

Using GPUs and general purpose CPUs to construct heterogeneous parallel systems has attracted much interest in the field of high performance computing (HPC) [5]. GPUs' powerful floating-point operation capacity and high performance-per-watt qualify them as good accelerators to speedup CPU applications for high performance with relatively small system scale and low power consumption. In this section, we introduce some background information concerning programming on an AMD GPU platform using Brook+, including the micro architecture of Radeon HD4870 GPU and the Brook+ stream programming environment.

2.1 Micro Architecture

In this paper, we use an AMD Radeon HD4870 GPU as the accelerator for the CPU. AMD's HD4800 series (codename RV770) is the newest GPU in their stream computing lineup which supports double precision floating point operations. The RV770 core has 10 SIMD engines, each of which contains 16 thread processors. Each thread processor consists of five scalar stream cores. So there are in total 800 cores integrated on a single die. The five cores can execute both single-precision floating point and integer operations, with one of them being able to handle transcendental operations, such as sin, cos, and log. Notably, a thread processor combines four of its stream cores (excluding the transcendental one) to process double-precision operations. In addition, a branch execution unit is contained in each thread processor to handle branch executions. Tab. 1 summarizes the HD4870's specifications.

In the AMD stream computing model, a stream denotes a collection of data elements of the same type that can be operated on in parallel. A kernel is a parallel function that operates on each element of an output stream. An instance of kernel execution on a thread processor is called a thread. Threads are mapped to thread processors for execution and scheduled in *wavefronts*. A

Table 1. AMD's HD4870 Specification

Thread Processors	800	Memory Clock Speed	993 MHz
Texture Units	40	Memory Interface	256 bits
Core Clock Speed	750 MHz	Memory Bandwidth	115 GB/s
Memory Type	GDDR5	Single (Double) Peak	1.2 T (240G) flops

wavefront is a group of threads executed and scheduled together on a SIMD engine. The hardware schedules limited resources, such as the number of general purpose registers (GPRs) used and memory bandwidth, until all threads have been processed. Multiple threads are interleaved to hide latencies due to memory accesses and stream core operations. Since SIMD engines run independently of each other, it is possible for each engine to execute different instructions. Moreover, pipelining is adopted in the GPU hardware to achieve time multiplexing when performing ALU operations [1].

2.2 Brook+ Stream Programming Environment

AMD has provided a complete software programming stack for programmers to easily accommodate stream programming. Programmers can write codes at two levels of abstraction: using Brook+ at a high level and using the compute abstraction layer (CAL) at a low level. Brook+ is an extension of C that supports an explicit model of parallelism based on the BrookGPU [6]. It is a high level programming language that abstracts away architecture details while maintaining features relevant to modern graphics hardware. CAL is a device driver library that provides a forward-compatible interface to and directly interacts with stream processors. Compared with Brook+, CAL reveals a rich opportunity for optimization given that programmers have the knowledge of the underlying hardware. However, our implementation is limited to the high level Brook+ programming. The Brook+ compiler splits a Brook+ program into CPU code and GPU code, and compiles the GPU code (kernels) to intermediate language (IL) code for further GPU-oriented compilation [1].

3 Optimization Strategies

In this section, we describe four optimization strategies and show how they can be applied to optimize stencil computations in *Mgrid*. We use the *Resid* and *Interp* kernels to exemplify our methods since among the four main kernels in *Mgrid*, *Resid* consumes 46% of the whole application executing time and *Interp* is the only kernel with branches in the naive implementation.

3.1 Improving Thread Utilization

A GPU is more suitable for performing computing-intensive rather than memory-intensive applications [7]. Since each thread processor can perform parallel operations, there should be enough threads to occupy the parallel stream cores, thus enabling multiple *wavefronts* to be formed and scheduled to hide memory access latencies in the hope of fully exploiting the tremendous computing power of GPUs.

While thread level parallelism is essential in obtaining high performance, thread locality, i.e., data reuse that can be exploited within each thread, is also very important in light of much data reuse in scientific computing. In the stencil computations of *Mgrid*, many intermediate results can be reused. Reusing these results reduces memory fetches as well as computation. Generally, large thread granularity can improve data locality and computation intensity, though entailing the consumption of more GPRs. We will explain the concept of thread granularity later.

Given limited resources (such as the number of GPRs, memory bandwidth) and the amount of work (e.g., a specific kernel computation), the number of threads that can be created is determined by thread granularity. In other words, the thread granularity is in inverse proportion to the number of threads. This means tuning thread granularity can balance locality within thread and parallelism among threads [8]. So there should be a balance between thread locality and thread parallelism through tuning. To achieve the best performance, programmers should carefully tune the thread granularity to strike the right balance.

In Brook+, the total number of threads is determined by the output stream size (domain size). Note that the thread granularity here means the number of grid points a thread calculates. There are two methods to tuning thread granularity in Brook+:

(a) Using vector types

Brook+ provides built-in short vector types for tuning the code explicitly on the available short-SIMD machines. Short vectors here are built from the name of their base type, with the size appended as a suffix, such as float4 and double2. Using vector types reduces the domain size (the output stream length) by a factor of the vector size, and consequently increases the thread granularity by the same factor. Take double2 for example. Using double2 increases the thread granularity by a factor of two. A thread can now compute two stencil points at a time, so more data reuse can be exploited through using the intermediate results within each thread.

Moreover, using vector types can pack up to four scalar fetches into one vector fetch to form a vector fetch. Vector fetch significantly reduces memory fetches if a kernel is designed to fetch from consecutive data locations, thus making more efficient use of the fetch resources. For example, a kernel can issue a float4 fetch in one cycle versus four separate float fetches in four cycles. In the stencil computations of *Resid* where locations of the data to be fetched are usually consecutive, vector fetches naturally raise arithmetic intensity and improve memory performance.

(b) Multiple output streams

Brook+ supports up to eight simultaneous output streams per kernel using the CAL backend [1]. Using multiple output streams, a thread can complete multi-folds of computations with respect to using a single output stream, which also increases the thread granularity. For example, using two output streams doubles the thread granularity. If we combine vector types and multiple output streams together, even larger thread granularity can be attained. For instance, using double2 and four output streams together, we increase the thread granularity by a factor of eight (24=8).

However, using vector types needs modification of indices in the kernel body, and adopting multiple output streams requires splitting the input streams. Also, both methods increase the requirement for GPRs and consequently reduce the number of active threads that can be created. Whether these kinds of overheads incurred can be offset by the performance gain of tuning thread granularity is determined by kernel size and specific kernel characteristics. As for what thread granularity is best, we should determine this through experiments.

3.2 Stream Reorganization

Although memory access with texture unit (memory) supports 1D, 2D and 3D addressing modes, the texture cache is optimized for 2D locality. In order to exploit more data locality in the cache, the threads in the same *wavefront* should read texture addresses that are close along two dimensions. This process may need transformation on data layout or data structure.

Taking the implementation of the *Resid* kernel on the GPU for example, the runtime library would automatically linearly expand 3D data streams into 2D data streams. This transformation may impact the cache performance, because the computation needs to access adjacent data in three dimensions. To get better performance, the 3D stream should be transformed into a 2D stream in the block manner. The process is illustrated in Fig. 1(a). Compared with the layout in the linearly expanding manner, the data adjacent in the logical space is kept adjacent in the 2D stream. The *Resid* kernel refers data on three consecutive planes when performing stencil computations for a grid point. After stream reorganization, we exploit the cache data locality within each plane.

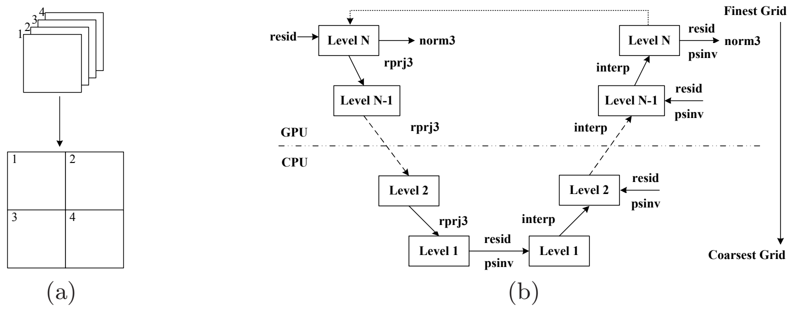


Fig. 1. (a) Transforming the 3D stream into 2D stream in the block manner [9] (b) V-cycle pattern of *Mgrid*

3.3 Branch Elimination

GPUs adopt SIMD execution mode, which incurs large flow control overhead. Take branching for example. AMD GPUs combine all the necessary paths as a *wavefront*. However, even if only one thread within a *wavefront* diverges, the rest of the threads in the *wavefront* have to execute the branch, which means all the paths are executed serially. This situation degrades the kernel performance greatly. Therefore branch divergences in kernels should be eliminated as much as possible.

We convert control dependence to data dependence, which caters to GPUs powerful data processing capability [10]. Our branch elimination is a two-step strategy: a) Branch fusion. Our branch fusion here is only suitable for the situation where the left expressions of *if* and *else* are the same. If not, there is no benefit using branch fusion since the expressions in both branches have to be executed. b) Expression simplification. The second step is used to simplify expressions gotten from branch fusion in the hope of eliminating all the redundant computations. With branch elimination, we can eliminate all the eight branches in the *Interp* kernel.

3.4 CPU-GPU Task Distribution

GPUs are good at performing ALU-intensive tasks, which qualifies them as good accelerators for CPUs. The philosophy of using GPUs to accelerate applications is to manipulate massive threads to exploit parallelism among threads and hide memory access latencies. So when the problem size is very small, there may be not enough threads to occupy stream processing cores to fully exploit parallelism. Take the problem size 16^3 for example. Assuming that there are enough GPRs, only 4K threads are needed to process the computation under this problem size, which is much less than the maximum 10K threads that the RV770 core can provide, not to mention the smaller problem sizes.

When the speedup obtained by the GPU is less than one, we should consider turning the task back to run on the CPU. Nevertheless, porting computing tasks to the CPU entails inevitable overhead such as data communication latency. This indicates the performance gain from distributing the task among the CPU and the GPU must counteract this overhead for the purpose of improving the overall system performance. Distributing tasks between the CPU and the GPU is sure to outperform the CPU- or GPU-single computing device execution.

4 Experimental Evaluation

To examine the benefits of our optimization strategies, we implemented the *Mgrid* application using Brook+ on an AMD Radeon HD4870 GPU. All the results are compared to the single-thread CPU version, which is measured on an Intel Xeon E5405 CPU running at 2GHz with 256KB L1 cache and 12MB L2 cache. We used the Intel *ifort* compiler as the CPU compiler with the optimization option *-O3*.

Mgrid is a 3D multigrid application in the SPECfp/NAS benchmark. Notably, it is the only application found in both the SPEC and NAS benchmark suites, and among those few SPEC 2000 applications surviving through SPEC 95 and SPEC 98. The main process of Mgird is of a V-cycle pattern performed at multi-level grids in multi-pass (multiple iterations), as illustrated in Fig. 1(b). *Mgrid* spends 85% of its execution time performing stencil computations on 3D arrays, which go through all of the four primary kernels, including *Resid*, *Psinv*, *Rprj3*, and *Interp*. The *Resid* kernel computes the residual. The *Psinv* kernel computes the approximate inverse. The *Rprj3* kernel computes the projection from fine grid to coarse grid. The *Interp* kernel computes the interpolation from coarse grid to fine grid.

4.1 Effects of Improving Thread Utilization

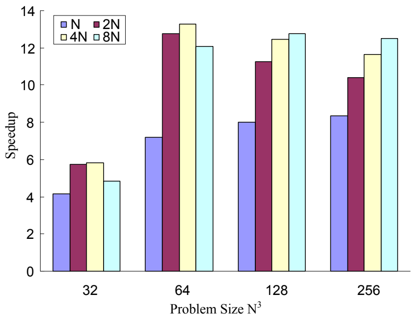
Here we examine the effectiveness of improving thread utilization. Applying the strategies proposed in Subsection 3.1, we used four thread granularities: double, double2, double2 plus two output streams, and double2 plus four output streams. If a thread using double can compute N points, then a thread using double2, double2 plus two output streams and double2 plus four output streams can compute 2N, 4N, and 8N points, respectively. For conveniences sake, we used N, 2N, 4N and 8N to denote different thread granularities.

Fig. 2(a) shows the speedup of the *Resid* kernel over the CPU implementation under different problem sizes using the four thread granularities. Note that since our optimization strategies target stencil computations in *Mgrid*, when evaluating a single kernel, we do not count in the time consumed by loading the kernels to the GPU and by the periodical communication subroutine *Comm3* in each kernel. Nevertheless, in our overall evaluation for *Mgrid*, all the time will be counted.

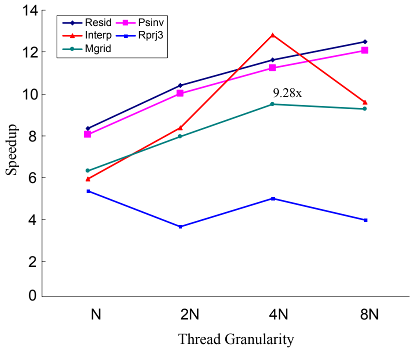
As shown in Fig. 2(a), the thread granularity 8N demonstrates the best performance under problem size 256^3 and 128^3, while 4N yields the best performance under the other two problem sizes. Problem sizes smaller than 32^3 are not shown in the figure since their speedups are less than one. We can see that under large problem sizes 256^3 and 128^3, the speedup for each kernel scales up with the granularity. This is because under large problem size, large thread granularity provides more chances to exploit intermediate result reuse within threads, yet there are enough threads to exploit parallelism among threads. However, under small problem size, large thread granularity requires more GPRs in each thread, thus impacting the number of threads and resulting in limited parallelism among threads. Under problem size 64^3 and 32^3, the speedups first scale with the granularity, reach a maximum in granularity 4N, and then decrease in granularity 8N, see Fig. 2(a). The performance gain through hitting more data reuse within each thread using large granularity (8N) is offset by the performance loss caused by lacking enough threads to fully occupy parallel stream computing cores.

Fig. 2(b) shows the speedup of each kernel and the whole application *Mgrid* under the largest problem size 256^3. We can see that the speedups of the kernel *Resid* and *Psinv* scale up with the thread granularity monotonously. This is because large thread granularity is favorable for intermediate data reuse within each thread, while there are abundant threads for thread level parallelism.

Interp computes the coarse grid through accessing the finer grid, so it tends to be an ALU-intensive kernel. The speedup of the *Interp* kernel increases rapidly with the thread granularity and reaches a maximum in thread granularity 4N, see



(a)



(b)

Fig. 2. (a) Performance of the kernel *Resid* under different problem sizes using different thread granularities (b) Speedup of each kernel and *Mgrid* under the largest problem size 256^3

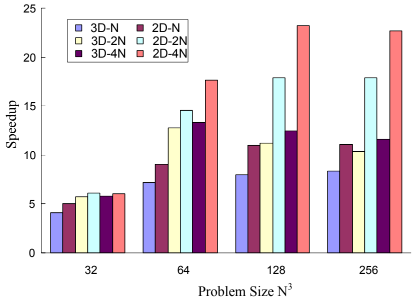
Fig. 2(b). However, the speedup descends when the thread granularity reaches 8N. This is mainly due to large thread granularity impacting the number of threads being created, even though *Interp* is an ALU-intensive kernel.

Rprj3 computes the finer grid through accessing the coarse grid. There is little data reuse between two grid points, so it is more like a memory intensive kernel. For memory intensive kernels, the way to speedup the execution is to schedule enough threads to hide memory fetch latencies. Therefore, the kernel performance degrades with increasing thread granularity, even though larger thread granularity contributes to more data reuse within each thread to some extend, as we can see the speedup trend of the *Rprj3* kernel in Fig. 2(b).

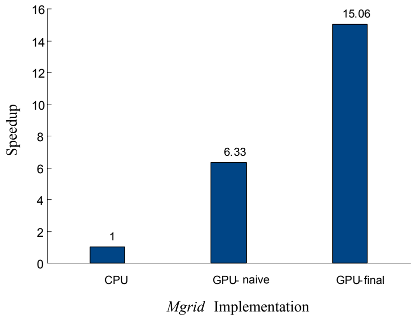
The overall speedup of *Mgrid* peaks in the thread granularity 4N, as shown in Fig. 2(b), which means that when using this thread granularity, we strike the right balance between thread locality and thread parallelism. The largest speedup attained through tuning the thread granularity was 9.28x under the largest problem size.

4.2 Effects of Stream Reorganization

Applying the stream reorganization strategy described in Subsection 3.2, we now examine its effectiveness using the *Resid* kernel. As shown in Fig. 3(a), the speedups increase with the thread granularity under all the problem sizes except the problem size 32^3. Under all the problem sizes, the speedups in the 2D stream organization are higher than that in the 3D stream organization. This demonstrates that the stream reorganization strategy is very effective. Note that in large thread granularity, the speedups differs sharply between the two data layouts. This is because the 2D version has better cache locality and reduced memory fetch ratio. Therefore more active threads can be created while maintaining the performance gain through increasing the thread granularity. Moreover, when



(a)



(b)

Fig. 3. (a) Speedup of the *Resid* kernel in 2D data organization compared to 3D data organization (b) Overall performance improvement

the problem size becomes larger, the cache pressure also gets larger. Then the advantage of 2D data layout to 3D data layout will become more evident since the 2D stream organization enhances cache locality.

4.3 Effects of Branch Elimination

The *Interp* kernel computes the interpolation from coarse grid to fine grid. Eight grid points forming a cube in the coarse grid correspond to one grid point in the fine grid, and every point is calculated according to the parity of its x, y and z coordinates. So there are eight conditional branches in the kernel. Using the branch elimination strategy proposed in Subsection 3.3, we eliminate all its eight branches. Our experiment results show that the improved version obtains a speedup of 5.95x over the CPU version and approximately 2.0x speedup over the naive GPU version. This demonstrates that the branch elimination strategy is very effective.

4.4 Effects of CPU-GPU Work Distribution

When using the work distribution strategy, we should take a holistic view of the entire system, and reasonably distribute work among the CPU and GPU. In our experiment, we found that the problem size 32^3 is the right partition point, and we assigned the work under the problem size 32^3 back to the CPU. As illustrated in Fig. 1(b), the computations of large problem sizes (larger than 32^3) above the dotted line are assigned to the GPU, while the others of small problem size below the dotted line are assigned back to the CPU.

Finally, we applied all the four optimization strategies to the naive GPU implementation. As shown in Fig. 3(b), the final implementation gained a speedup of 15.06x over the CPU version and 2.38x over the naive GPU version.

5 Related Work

With the popular adoption of GPUs in scientific computing, much research has recently been performed in optimizing GPU applications using general programming languages such as Brook+ and CUDA.

Ryoo et al. proposed optimization principles for efficient mapping of computation to graphics hardware [11]. Their main concern was using massive multithreading to exploit the rich stream core resource and hide memory fetch latency. Jang et al. presented an optimization methodology that utilizes architectural information to accelerate programs on AMD GPUs [12]. They exploited optimizations by defining optimization spaces. Their work demonstrated many AMD GPU details. Wang et al. presented GPU implementation of *Mgrid* using CUDA in the single precision floating point version [13]. They exploited data locality in 3-level memory hierarchies and tuned thread granularity thus reducing the pressure on the off-chip memory bandwidth. Due to architecture deviation, their optimization strategies can not directly applied using Brook+ on AMD GPUs.

In our work, we choose *Mgrid* since its stencil computations provide rich opportunity for exploiting on-chip parallelism and hiding memory accessing latencies. Li et al. proposed a compiler framework for automatic tiling of iterative stencil loops to improve the cache reuse [14]. Krishnamoorthy et al. developed an approach for automatic parallelization of stencil codes, explicitly addressing the issue of load-balanced execution of tiles caused by loop skewing in the time dimension [4]. They focused on improving cache locality of the CPU.

6 Conclusions and Future Work

In this paper, we implemented and optimized a real benchmark application *Mgrid* on AMD Radeon HD4870 GPU, and achieved very good experimental results. Though our implementation and optimizations are based on *Mgrid*, the optimization strategies can be use to improve any stencil computations. In the future, we would like to determine thread granularity automatically to simplify the application optimization on the GPU; and we would also consider exploring the GPU application performance at the intermediate language (IL) level. To fully exploit the CPU and GPU heterogeneous platform, we would try to automatically distribute tasks between the two computing resources.

References

1. AMD.: Ati stream computing user guide v1.4beta (2009), `http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf`
2. NVIDIA.: Compute unified device architecture programming guide v2.1beta (2009), `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`

3. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73–82. ACM, New York (2008)
4. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. SIGPLAN Not. 42(6), 235–244 (2007)
5. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: SC 2004: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 47. IEEE Computer Society Press, Los Alamitos (2004)
6. Buck, I.: Brook specification v0.2 (2003), `http://hci.stanford.edu/cstr/reports/2003-04.pdf`
7. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.-Z., Baghsorkhi, S.S., Hwu, W.-m.W.: Program optimization carving for gpu computing. J. Parallel Distrib. Comput. 68(10), 1389–1401 (2008)
8. Mohan, T., de Supinski, B.R., McKee, S.A., Mueller, F., Yoo, A., Schulz, M.: Identifying and exploiting spatial regularity in data memory references. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 49. IEEE Computer Society, Los Alamitos (2003)
9. Harris, M.J., Baxter, W.V., Scheuermann, T., Lastra, A.: Simulation of cloud dynamics on graphics hardware. In: HWWS 2003: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Aire-la-Ville, Switzerland, Switzerland, pp. 92–101. Eurographics Association (2003)
10. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL 1983: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 177–189. ACM, New York (1983)
11. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73–82. ACM, New York (2008)
12. Jang, B., Do, S., Pien, H., Kaeli, D.: Architecture-aware optimization targeting multithreaded stream computing. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 62–70. ACM, New York (2009)
13. Wang, G., Yang, X.J., Zhang, Y., Tang, T., Fang, X.D.: Program optimization of stencil based application on the gpu-accelerated system. In: Intl. Symposium on Parallel and Distributed Processing and Applications, pp. 219–225 (2009)
14. Li, Z., Song, Y.: Automatic tiling of iterative stencil loops. ACM Trans. Program. Lang. Syst. 26(6), 975–1028 (2004)

Author Index