Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine

Walter Lee, Rajeev Barua, Devabhaktuni Srikrishna, Jonathan Babb,
Vivek Sarkar, Saman Amarasinghe, *
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.

{walt,barua,chinnama,jbabb}@lcs.mit.edu
{vivek,saman}@lcs.mit.edu
http://cag-www.lcs.mit.edu/raw

Abstract

Advances in VLSI technology will enable chips with over a billion transistors within the next decade. Unfortunately, the centralized-resource architectures of modern microprocessors are ill-suited to exploit such advances. Achieving a high level of parallelism at a reasonable clock speed requires distributing the processor resources – a trend already visible in the dual-register-file architecture of the Alpha 21264. A Raw microprocessor takes an extreme position in this space by distributing all its resources such as instruction streams, register files, memory ports, and ALUs over a pipelined two-dimensional interconnect, and exposing them fully to the compiler. Compilation for instruction-level parallelism (ILP) on such distributed-resource machines requires both spatial instruction scheduling and traditional temporal instruction scheduling. This paper describes the techniques used by the Raw compiler to handle these issues. Preliminary results from a SUIF-based compiler for sequential programs written in C and Fortran indicate that the Raw approach to exploiting ILP can achieve speedups scalable with the number of processors for applications with such parallelism. The Raw architecture attempts to provide performance that is at least comparable to that provided by scaling an existing architecture, but that can achieve orders of magnitude improvement in performance for applications with a large amount of parallelism. This paper offers some positive results in this direction.

1 Introduction

Modern microprocessors have evolved while maintaining the faithful representation of a monolithic uniprocessor. While innovations in the ability to exploit instruction level parallelism have placed greater demands on processor resources, these resources have remained centralized, creating scalability problem at every design point in a machine. As processor designers continue in their pursuit of an architecture

*This research is funded in part by ARPA contract # DABT63-96-C-0036 and in part by an NSF Presidential Young Investigator Award.

MIT-LCS-TM-572, December 1997

that can exploit more parallelism and thus requires even more resources, the cracks in the view of a monolithic underlying processor can no longer be concealed.

An early visible effect of the scalability problem in commercial architectures is apparent in the clustered organization of the Multiflow computer [12]. More recently, the Alpha 21264 [8] duplicates its register file to provide the requisite number of ports at a reasonable clock speed. A cluster is formed by organizing half of the functional units and half of the cache ports around each register file. Cross-cluster communication incurs an extra cycle of latency.

As the amount of on-chip processor resources continues to increase, the pressure toward this type of non-uniform spatial structure will continue to mount. Inevitably, from such hierarchy, resource accesses will have non-uniform latencies. In particular, register or memory access by a functional unit will have a gradation of access time. This fundamental change in processor model will necessitate a corresponding change in compiler technology. *Instruction scheduling becomes a spatial problem as well as a temporal problem.*

The Raw machine [17] is a scalable microprocessor architecture with non-uniform register access latencies (NURA). As such, its compilation problem is similar to that which will be encountered by extrapolations of existing architectures. In this paper, we describe the compilation techniques used to exploit ILP on the Raw machine, a NURA machine composed of fully replicated processing units connected via a mostly static programmable network. The fully exposed hardware allows the Raw compiler to precisely orchestrate computation and communication in order to exploit ILP within basic blocks. The compiler handles the orchestration by performing spatial and temporal instruction scheduling, as well as data partitioning using a distributed on-chip memory model.

This paper makes three contributions. First, it describes the space-time instruction scheduling of ILP on a Raw machine using techniques borrowed from two existing domains: mapping of tasks to MIMD machines and mapping of circuits to FPGAs. Second, it introduces a new control flow model based on asynchronous local branches inside a machine with multiple independent instruction streams. Finally, it shows that independent instruction streams give the Raw machine the ability to tolerate timing variations due to dynamic events, in terms of both correctness and performance.

The rest of the paper is organized as follows. Section 2 motivates the need for NURA machines, and it introduces the Raw machine as one such machine. Section 3 describes RAWCC, a compiler for NURA machines. Section 4 discusses Raw's decentralized approach to control flow. Section 5 shows the performance of RAWCC. Section 6 presents related work, and Section 7 concludes. Appendix A gives a proof of how a mostly static machine can tolerate skews introduced by dynamic events without changing the behavior of the program.

2 Motivation and Background

This section motivates the Raw architecture. We examine the scalability problem of modern processors, trace an architectural evolution that overcomes such problems, and show that the Raw architecture is at an advanced stage of such an evolution. We highlight non-uniform register access as an important feature in scalable machines. We then describe the Raw machine, with emphasis on features which make it an attractive scalable machine. Finally, we describe the relationship between a Raw machine and a VLIW machine.

The Scalability Problem Modern processors are not designed to scale. Because superscalars require

significant hardware resources to support parallel instruction execution, architects for these machines face an uncomfortable dilemma. On the one hand, faster machines require additional hardware resources for both computation and discovery of ILP. On the other hand, these resources often have quadratic area complexity, quadratic connectivity, and global wiring requirements which can be satisfied only at the cost of cycle time degradation. VLIW machines address some of these problems by moving the cycle-time elongating task of discovering ILP from hardware to software, but they still suffer scalability problems due to issue bandwidth, multi-ported register files, caches, and wire delays.

Up to now, commercial microprocessors have faithfully preserved their monolithic images. As pressure from all sources demands computers to be bigger and more powerful, this image will be difficult to maintain. A crack is already visible in the Alpha 21264. In order to satisfy timing specification while providing the register bandwidth needed by its dual-ported cache and four functional units, the Alpha duplicates its register file. Each physical register file provides half the required ports. A cluster is formed by organizing two functional units and a cache port around each register file. Communication within a cluster occurs at normal speed, while communication across clusters takes an additional cycle.

This example suggests an evolutionary path that resolves the scalability problem: impose a hierarchy on the organization of hardware resources [16]. A processor can be composed from replicated processing units whose pipelines are coupled together at the register level so that they can exploit ILP cooperatively. The VLIW Multiflow TRACE machine is a machine which adopts such a solution [12]. On the other hand, its main motivation for this organization is to provide enough register ports. Communication between clusters are performed via global busses, which in modern and future-generation technology would severely degrade the clock speed of the machine. This problem points to the next step in the scalability evolution – providing a scalable interconnect. For machines of modest sizes, a bus or a full crossbar may suffice. But as the number of components increases, a point to point network will be necessary to provide the required latency and bandwidth at the fastest possible clock speed – a progression reminiscent of the multiprocessor evolution.

The result of the evolution toward scalability is a machine with a distributed register file interconnected via a scalable network. In the spirit of NUMA machines (Non-Uniform Memory Access), we call such machines *NURA machines* (Non-Uniform Register Access). Like a NUMA machine, a NURA machine connects its distributed storage via a scalable interconnect. Unlike NUMA, NURA pools the shared storage resources at the register level. Because a NURA machine exploits ILP of a single instruction stream, its interconnect must provide latencies that are much lower than that on a multiprocessor.

As the base element of the storage hierarchy, any change in the register model has profound implications. The distributed nature of the computational and storage elements on a NURA machine means that locality should be considered when assigning instructions to functional units. Instruction scheduling becomes a spatial problem as well as a temporal problem. This extra dimension requires compilation techniques beyond that which are used to exploit ILP on modern machines.

Raw architecture The Raw machine [17] is a NURA architecture motivated by the need to design simple and highly scalable processors. As depicted in Figure 1, a Raw machine comprises a simple, replicated tile, each with its own instruction stream, and a programmable, tightly integrated interconnect between tiles. A Raw machine also supports multi-granular (bit and byte level) operations as well as customizable configurable logic, but this paper does not address these features.

Each Raw tile contains a simple five-stage pipeline, interconnected with other tiles over a pipelined, point-to-point network. The tile is kept simple and devoid of complex hardware resources in order to maximize the clock rate and the number of tiles that can fit on a chip. Raw's network resides between

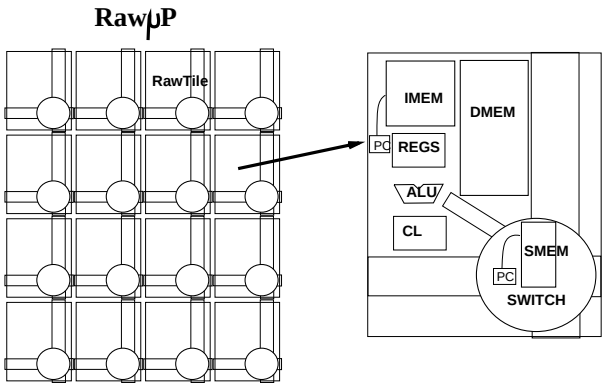

Figure 1: A Raw microprocessor.

the register files and the functional units to provide fast, register-level communication. Unlike modern superscalars, the interface to this interconnect is fully exposed to the software.

A switch on a Raw machine is integrated directly into the processor pipeline to support single-cycle sends and receives of word-sized values. A word of data travels across one tile in one clock cycle. The switch contains two distinct networks, a static and a dynamic one. The static switch is programmable, allowing statically inferable communication patterns to be encoded in the instruction streams of the switches. This approach eliminates the overhead of composing and routing a directional header, which in turn allows a single word of data to be communicated efficiently. Communication instructions (*send*, *receive*, or *route*) have blocking semantics that provide near-neighbor flow control; a processor or switch stalls if it is executing an instruction that attempts to access an empty input port or a full output port. This specification ensures correctness in the presence of timing variations introduced by dynamic events such as cache misses, and it obviates the lock-step synchronization of program counters required by many statically scheduled machines. The dynamic switch is a wormhole router that makes routing decisions based on the header of each message. It includes additional lines for flow control. We focus on communication using the static network in this paper.

The Raw prototype we have developed uses an MIPS R2000 processor on its tile. For the switch, it uses a stripped down R2000 augmented with a *route* instruction. Communication ports are added as extensions to the register set. Figure 2 shows the organization of ports for the static network on a single tile. It takes one cycle to inject a message from the processor to its switch, receive a message from a switch to its processor, or route a message between neighboring tiles. A single-word message between neighboring processors would take four cycles. Note, however, that the ability to access the communication ports as register operands allows useful computation to overlap with the act of performing a send or a receive. Therefore, the *effective* overhead of the communication can be as low as two cycles.

In addition to its scalability and simplicity, the Raw machine is an attractive NURA machine for several reasons:

- *Multisequentiality:* Multisequentiality, the presence of multiple flows of control, is useful for four reasons. First, it significantly enhances the potential amount of parallelism a machine can exploit [11]. Second, it enables *control localization*, a technique we introduce in Section 4.1 to allow ILP to be scheduled across branches. Third, it enables *asynchronous global branching* described in Section 4.2, a means of implementing global branching without global synchronization and

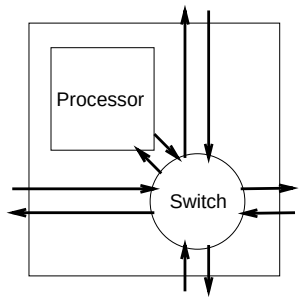

Figure 2: Communication ports for the static network on a prototype tile. Each switch on a tile is connected to its processor and its four neighbors via an input port and an output port. The figure is not drawn to scale.

degradation of clock speed. Finally, it gives a Raw machine better tolerance of dynamic events compared to a VLIW machine, as shown in Section 5.

- *Simple means of expanding the register space:* Traditionally, an ISA prevents a processor from increasing the size of the compiler-visible register set, even if applications can benefit from such increase. Modern architectures circumvent this problem by providing compiler-invisible registers which are utilized by complex dynamic renaming logic. This interface, however, undermines static compiler analysis for discovering ILP, and it requires expensive supporting hardware. Moreover, compiler-invisible registers cannot be used to reduce register spills. A Raw machine, on the other hand, adds registers by increasing the number of tiles, without any need to change the ISA on a tile. The number of register ports increases proportionally as well.

- *A compiler interface for locality management:* The Raw machine fully exposes its hardware to the compiler by exporting a simple cost model for communication and computation. In turn, the compiler is responsible for the assignment of instructions to Raw tiles. We believe instruction partitioning should be accomplished at compile time for several reasons: first, it requires sophisticated analysis of the structure of the program dependence graph, for which the necessary information is readily available at compile time but not at run-time; second, we can ill-afford its complexity at run-time.

- *Mechanism for precise orchestration:* Raw's programmable static switch is an essential feature for exploiting ILP on the Raw machine. First, it allows single-word register-level transfer without the overhead of composing and routing a message header. Second, the Raw compiler can use its full knowledge of the network status to minimize congestion and route data around hot spots. Most importantly, the static network gives the Raw compiler the fine-grain control required to orchestrate computation and communication events to eliminate synchronization overhead.

Relationship between Raw and VLIW The Raw architecture draws much of its inspiration from VLIW machines. They both share the common goal of statically scheduling ILP. From a macroscopic point of view, a Raw machine is the result of a natural evolution from a VLIW, driven by the desire to add more resources.

There are two major distinctions between a VLIW machine and Raw machine. First, they differ in resource organization. VLIW machines of various degrees of scalability have been proposed, ranging from completely centralized machines to machines with distributed functional units, register files, and

memory [12]. The Raw machine, on the other hand, is the first ILP microprocessor that provides a scalable, two-dimensional interconnect between clusters of resources. This feature makes the space-time scheduling problem more general for Raw machines than for VLIWs, and it limits the length of all wires to the distance between neighboring tiles, which in turn enables a higher clock rate.

Second, the two machines differ in their control flow model. A VLIW machine has a single flow of control, while a Raw machine has multiple flows of control. As explained above, this feature increases available exploitable parallelism, enables control localization and asynchronous global branching, and improves tolerance of dynamic events.

3 The Raw compiler

This section describes the Raw compiler which performs the space-time scheduling of ILP on a Raw machine. Section 3.1 explains its memory model. Section 3.2 describes the compiler itself. Section 3.3 describes the basic block orchestrater, the space-time scheduling component of the compiler.

3.1 Memory model

Memory on a Raw machine is distributed across the tiles. The Raw compiler distributes arrays through *low order interleaving*, which interleaves the arrays element-wise across the memory system. The Raw memory model provides two ways of accessing this memory system, one for static reference and one for dynamic reference. A reference is static if every invocation of it can be determined to be referring to memory on one specific tile. We call this property the *static reference property*. A static reference is handled by mapping it to the corresponding tile at compile time. A non-static or dynamic reference is handled by disambiguating the address at run-time either in software or in hardware, using the dynamic network to handle any necessary communication.

The Raw compiler attempts to identify as many static references as possible. Static references are attractive for two reasons. First, they can proceed without any of the overhead due to dynamic disambiguation and synchronization. Second, they can potentially take advantage of the full memory bandwidth. For array references which are affine functions of loop indices, we have developed a technique which uses unrolling to satisfy the static reference property. Our technique is a more fully developed version of the technique used by the Bulldog compiler to perform memory-bank disambiguation [6]. A description of this theory is beyond the scope of this paper.

This paper focuses on results which can be attained when the Raw compiler succeeds in identifying static references. We do not address the issues pertaining to dynamic references in this paper. However, we observe in Section 5 that decoupled instruction streams allow the Raw machine to tolerate timing variations due to events such as dynamic memory accesses.

3.2 RAWCC

RAWCC, the Raw compiler, is implemented using SUIF [18], the Stanford University Intermediate Format. It compiles both C and Fortran programs. The Raw compiler consists of three phases. The first phase performs high level program analysis and transformations, including traditional techniques such as memory disambiguation, loop unrolling, and array reshape, plus a new control localization technique to

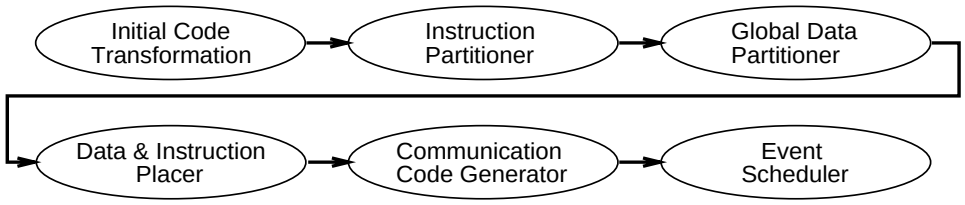

Figure 3: Task composition of the basic block orchestrater.

be discussed in Section 4.1. In the future, this phase can be extended with the advanced ILP-enhancing techniques discussed in Section 6. The second phase, the *basic block orchestrater*, performs the space-time scheduling of ILP on each basic block. It will be discussed in detail below. The final phase generates code for the processors and the switches. It uses the MIPS back-end developed in Machine SUIF [15], with a few modifications to handle the communication instructions and communication registers.

3.3 Basic block orchestrater

The basic block orchestrater exploits the ILP within a basic block by distributing the parallelism within the basic block across the tiles. It transforms a single basic block into an equivalent set of basic blocks that can be run in parallel on Raw. The orchestrater generates intermediate code for both the processor and the programmable switch on each tile. It performs four tasks: mapping instructions to processors, mapping scalar data to processors, generating any necessary communication, and scheduling both computation and communication.

The basic access model for program variables is as follows. Every variable is assigned to a home tile, where persistent values of the variable are stored. At the beginning of a basic block, the value of a variable is transferred from its home to the tiles which use the variable. Within a basic block, references are strictly on local renamed variables, allocated either in registers or on the stack. At the end of the basic block, the updated value of a modified variable is transferred from the computing tile to its home tile.

The opportunity to exploit ILP across statically-scheduled, MIMD processing units is unique to Raw. However, the Raw compiler problem is actually a composition of problems in two previously studied compiler domains: the task partitioning/scheduling problem and the communication scheduling problem. Research on the abstract task partitioning and scheduling problem has been extensive in the context of mapping directed acyclic task graphs onto MIMD multiprocessors (*e.g.,* [14][19]). The Raw task partitioning and scheduling problem is similar, with tasks defined to be individual instructions. The communication scheduling problem has been studied in the context of FPGA routing. For example, a technique called VirtualWires [4] alleviates the pin limitation of FPGAs by multiplexing each pin to communicate more than one values. Communication events are scheduled on pins over time. In Raw, the communication ports on the processors and switches serve the same role as pins in FPGAs; the scheduling problems in the two contexts are analogous.

The following are the implementation details of the basic block orchestrater. Figure 3 shows its task decomposition. Each task is described in turn below. To facilitate the explanation, Figure 4 shows the transformations performed by RAWCC on a sample program.

Initial code transformation Initial code transformation massages a basic block into a form suitable for subsequent analysis phases. Figure 4a shows the transformations performed by this phase. First,

software renaming converts statements of the basic block to static single assignment form. Such conversion removes anti-dependencies and output-dependencies from the basic block, which in turn exposes the available parallelism. It is analogous to hardware register renaming performed by superscalars.

Second, two types of dummy instructions are inserted. *Read* instructions are inserted for variables which are live-on-entry and read in the basic block. *Write* instructions are inserted for variables which are live-on-exit and written within the basic block. These instructions simplify the eventual representation of *stitch code*, the communication needed to transfer values between the basic blocks. This representation in turn allows the event scheduler to overlap the stitch code with other work in the basic block.

Third, expressions in the source program are decomposed into instructions in three-operand form. Three-operand instructions are convenient because they correspond closely to the final machine instructions and because their cost attributes can easily be estimated. Therefore, they are logical candidates to be used as atomic partitioning and scheduling units.

Finally, the dependence graph for the basic block is constructed. A node represents an instruction, and an edge represents a true flow dependence between two instructions. Each node is labeled with the estimated cost of running the instruction. For example, the node for a floating point add in the example is labeled with two cycles. Each edge is labeled with the data size, from which the communication cost can be computed after the instruction mapping is known. A simple instruction always generates a word of data; its outgoing edge label is always one word and is implicit.

Instruction partitioner The instruction partitioner partitions the original instruction stream into multiple instruction streams, one for each tile. It does not bind the resultant instruction streams to specific tiles – that function is performed by the instruction placer. When generating the instruction streams, the partitioner attempts to balance the benefits of parallelism against the overheads of communication. Figure 4b shows a sample output of this phase.

Certain instructions have constraints on where they can be partitioned and placed. Read and write instructions to the same variable have to be mapped to the processor on which the data resides (see *data partitioner and placer* below). Similarly, loads and stores satisfying the static reference property must be mapped to a specific tile. The instruction partitioner performs its duty without such constraints. These constraints are taken into account in the global data partitioner and in the data and instruction placer.

Partitioning is performed in two phases called clustering and merging. We describe each in turn:

Clustering: Clustering attempts to partition instructions to minimize run-time, assuming non-zero communication cost but infinite processing resources. It groups together instructions that either have no parallelism, or whose parallelism is too small to exploit relative to the communication cost. Subsequent phases guarantee that instructions with no mapping constraints in the same cluster will be mapped to the same tile. The clustering technique approximates communication cost by assuming an idealized fully-connected switch with uniform latency.

RAWCC employs a greedy, critical path based technique called Dominant Sequent Clustering [19]. Initially, each instruction node belongs to a unit cluster. Communication between clusters is assigned a uniform cost. The algorithm visits instruction nodes in topological order. At each step, it selects from the list of candidates the instruction on the longest execution path. It then checks whether the selected instruction can merge into the cluster of any of its parent instructions to reduce the estimated completion time of the program. Estimation of the completion time is dynamically updated to take into account the clustering decisions already made, and it reflects the cost of both computation and communication. The algorithm completes when all nodes have been visited exactly once.

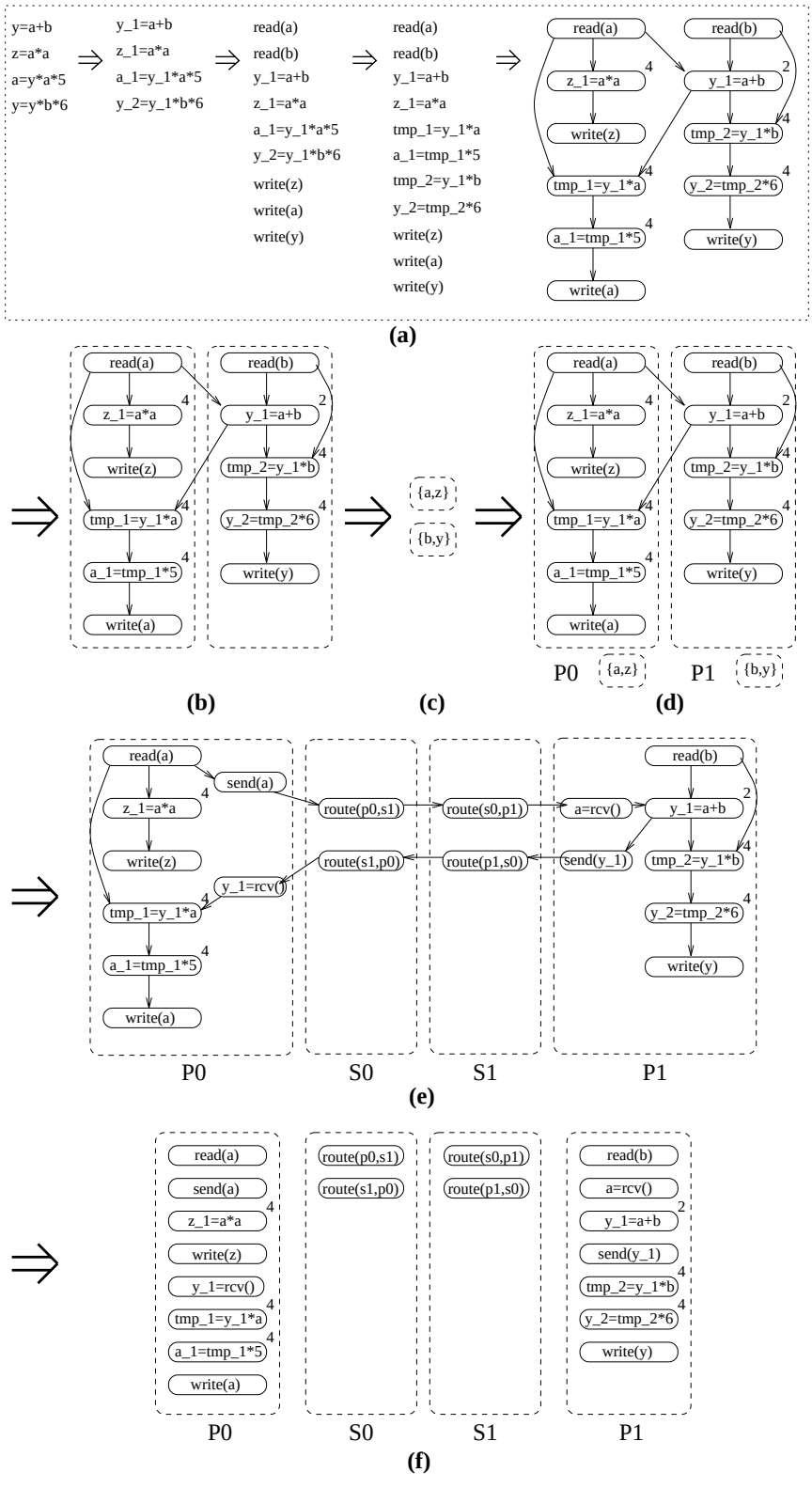

Figure 4: An example of the program transformations performed by RAWCC. (a) shows the initial program undergoing transformations made by *initial code transformation*; (b) shows result of *instruction partitioner*; (c) shows result of *global data partitioner*; (d) shows result of *data and instruction placer*, (e) shows result of *communication code generator*; (f) shows final result after *event scheduler*.

Merging: Merging combines clusters to reduce the number of clusters down to the number of tiles, again assuming an idealized switch interconnect. Two useful heuristics in merging are to maintain load balance and to minimize communication events. The Raw compiler currently uses a locality-sensitive load balancing technique which tries to minimize communication events unless the load imbalance exceeds a certain threshold. It will consider other strategies, including an algorithm based on estimating completion time, in the future.

The current algorithm is as follows. The Raw compiler initializes N empty partitions (where N is the number of tiles), and it visits clusters in decreasing order of size. When it visits a cluster, it merges the cluster into the partition with which it has the highest affinity, unless such merging results in a partition which is 20% larger than the size of an average partition. If the latter condition occurs, the cluster is placed into the smallest partition instead.

Global data partitioner To communicate values of data elements between basic blocks, a scalar data element is assigned a "home" tile location. Within basic blocks, renaming localizes most value references, so that only the initial reads and the final write of a variable need to communicate with its home location. Like instruction mapping, the Raw compiler divides the task of data home assignment into data partitioning and data placement.

The job of the data partitioner is to group data elements into sets, each of which is to be mapped to the same processor. To preserve locality as much as possible, data elements which tend to be accessed by the same thread should be grouped together. To partition data elements into sets which are frequently accessed together, RAWCC performs global analysis. The algorithm is as follows. For initialization, a virtual processor number is arbitrarily assigned to each instruction stream on each basic block, as well as to each scalar data element. In addition, statically analyzable memory references are first assigned dummy data elements, and then those elements are assigned virtual processor numbers corresponding to the physical location of the references. Furthermore, the access pattern of each instruction stream is summarized with its affinity to each data element. An instruction stream is said to have affinity for a data element if it either accesses the element, or it produces the final value for the element in that basic block. After initialization, the algorithm attempts to localize as many references as possible by remapping the instruction streams and data elements. First, it remaps instruction streams to virtualized processors given fixed mapping of data elements. Then, it remaps data elements to virtualized processors given fixed mappings of instruction streams. Only the true data elements, not the dummy data elements corresponding to fixed memory references, are remapped in this phase. This process repeats until no incremental improvement of locality can be found. In the resulting partition, data elements mapped to the same virtual processor are likely related based on the access patterns of the instruction streams.

Figure 4c shows the partitioning of data values into such affinity sets. Note that variables introduced by *initial code transformation* (*e.g.,* y_1 and tmp_1) do not need to be partitioned because their scopes are limited to the basic block.

Data and instruction placer The data and instruction placer maps virtualized data sets and instruction streams to physical processors. Figure 4d shows a sample output of this phase. The placement phase removes the assumption of the idealized interconnect and takes into account the non-uniform network latency. Placement of each data partition is currently driven by the those data elements with processor preferences, *i.e.,* those corresponding to fixed memory references. It is performed before instruction placement to allow cost estimation during instruction placement to account for the location of data. In additional to mapping data sets to processors, the data placement phase also locks the dummy read and

write instructions to the home locations of the corresponding data elements.

For instruction placement, RAWCC uses a swap-based greedy algorithm to minimize the communication bandwidth. It initially assigns clusters to arbitrary tiles, and it looks for pairs of mappings that can be swapped to reduce the total number of communication hops. For large configurations, this greedy algorithm can be replaced by one with simulated annealing for better performance.

Communication code generator The communication code generator translates each non-local edge (an edge whose source and destination nodes are mapped to different tiles) in the instruction task graph into communication instructions which route the necessary data value from the source tile to the destination tile. Figure 4e shows an example of such transformation. To minimize the volume of communication, edges with the same source are serviced jointly by a single multicast operation. Communication instructions include *send* and *receive* instructions on the processors as well as *route* instructions on the switches. New nodes are inserted into the graph to represent the communication instructions, and the edges of the source and destination nodes are updated to reflect the new dependence relations arising from insertion of the communication nodes.

The current compilation strategy assumes that network contention is low, so that the choice of message routes has less impact on the code quality compared to the choice of instruction partitions or event schedules. Therefore, communication code generation in RAWCC uses dimension-ordered routing; this spatial aspect of communication scheduling is completely mechanical. If contention is determined to be a performance bottleneck, a more flexible technique can be employed.

Event scheduler The event scheduler schedules the computation and communication events within a basic block with the goal of producing the minimal estimated run-time. Because routing in Raw is itself specified with explicit switch instructions, all events to be scheduled are instructions. Therefore, the scheduling problem is a generalization of the traditional instruction scheduling problem.

The task of scheduling communication instructions carries with it the responsibility of ensuring the absence of deadlocks in the network. If individual communication instructions are scheduled separately, the Raw compiler would need to explicitly manage the buffering resources on each communication port to ensure the absence of deadlock. Instead, RAWCC avoids the need for such management by treating a single-source, multiple-destination communication path as a single scheduling unit. When a communication path is scheduled, contiguous time slots are reserved for instructions in the path so that the path incurs no delay in the static schedule. By reserving the appropriate time slot at the node of each communication instruction, the compiler automatically reserves the corresponding channel resources needed to ensure that the instruction can eventually make progress.

Though event scheduling is a static problem, the schedule generated should remain deadlock-free and correct even in the presence of dynamic events such as cache misses. The Raw compiler uses the *static ordering property*, implemented through near-neighbor flow control, to ensure this behavior. (See Appendix A.) The static ordering property states that if a schedule does not deadlock, then any schedule with the same order of communication events will not deadlock. Because dynamic events like cache misses only add extra latency but do not change the order of communication events, they do not affect the correctness of the schedule.

The static ordering property also allows the schedule to be stored as compact instruction streams. Timing information needs not be preserved in the instruction stream to ensure correctness, thus obviating the need to insert no-op instructions. Figure 4f shows a sample output of the event scheduler. Note, first, the proper ordering of the route instructions on the switches, and, second, the successful overlap of

computation with communication on $P0$, where the processor computes and writes z while waiting on the value of y_1.

RAWCC uses a greedy list scheduler. The algorithm keeps track of a ready list of tasks. A task is either a computation or a communication path. As long as the list is not empty, it selects and schedules the task on the ready list with the highest priority. The priority scheme is based on the following observation. The priority of a task should be directly proportional to the impact it has on the completion time of the program. This impact, in turn, is lower-bounded by two properties of the task: its *level*, defined to be its critical path length to an exit node; and its *average fertility*, defined to be the number of descendent nodes divided by the number of processors. Therefore, we define the priority of a task to be a weighted sum of these two properties.

Like a traditional uniprocessor compiler, RAWCC faces a phase ordering problem with event scheduling and register allocation. Currently, the event scheduler runs before register allocation; it has no register consumption information and does not consider register pressure when performing the scheduling. The consequence is two-fold. First, instruction costs may be underestimated because they do not include spill costs. Second, the event scheduler may expose *too much* parallelism, which cannot be efficiently utilized but which comes at a cost of increased register pressure. The experimental results for fpppp-kernel in Section 5 illustrate this problem. We are exploring this issue and have examined a promising approach which adjusts the priorities of instructions based on how the instructions effect the register pressure. In addition, we intend to explore the possibility of cooperative inter-tile register allocation.

4 Control flow on Raw

In modern processors, control transition is expensive. Both superscalars and VLIWs employ a centralized branching model, where only one branch can be taken per cycle. Predicated execution alleviates this limitation somewhat, but its utility is limited by its processor utilization inefficiency and its need for hardware support. Furthermore, this centralized model makes the processor non-scalable. It requires global wires to coordinate the machine for synchronous, lock-step transfer of control. On superscalars, branching also incurs expensive overhead in the form of pipeline flushes caused by misprediction.

The Raw architecture proposes a different branching model which addresses these issues. The basic transfer of control is local rather than global. We try to decentralize the control transition when possible through *control localization*. When global control transition is required, we employ *asynchronous global branching*. These techniques are described below.

4.1 Control Localization

Control localization is the technique of turning global branches between basic blocks into local branches within an *extended basic block*. It is implemented by hiding the control flow inside a *macroinstruction*, an abstraction of a sequence of instructions for the purpose of partitioning, placement, and scheduling. Internalizing control flow to within a macroinstruction also allows ILP to be scheduled across it. Once control flow is hidden inside a macroinstruction, it no longer serves as barrier to instruction scheduling.

The construction of macroinstruction consists of three steps, initial formation, promotion of memory operations, and interface construction. Figure 5 shows an example of this process. First, initial formation identifies and groups together sequences of instructions, each to be converted to a macroinstruction. Then, static memory operations inside this sequence of instructions, along with their associated address

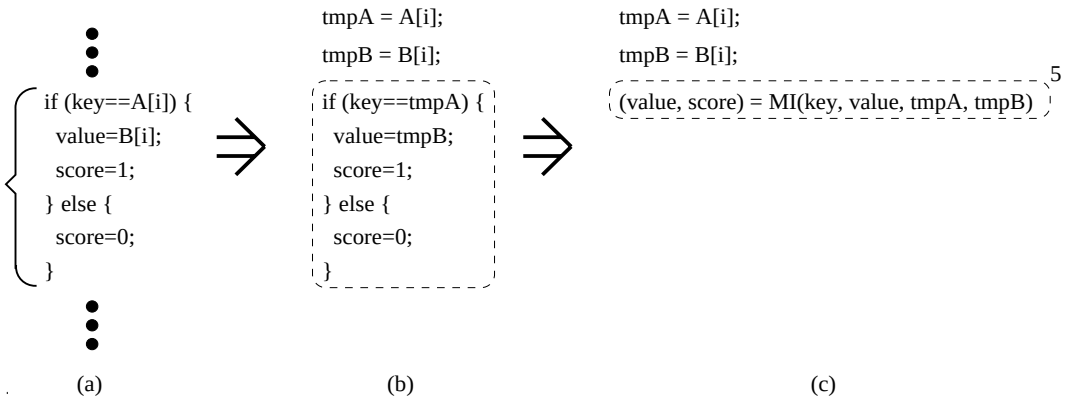

Figure 5: An example of macroinstruction formation. (a) initial formation; (b) promotion of memory operations; (c) interface construction. The superscript next to the macroinstruction indicates the estimated cost of executing the instruction.

computations, are promoted out of the sequence. This transformation is necessary to allow a macroinstruction to be treated as an abstract instruction which can be placed on any processor. Finally, interface construction analyzes the macroinstruction to determine its input variables, output variables, and estimated run-time. Input and output variables for a macroinstruction are computed by taking the union of their corresponding sets over all possible paths within the macroinstruction. If a variable belongs to the output set of at least one but not all paths in the macroinstruction, it needs to be considered as an input variable as well. This case is necessary to enable each path to be able to produce the value of that variable for subsequent dependent instructions. The estimated run-time is a function of the run-times of the individual paths. It can be computed with the help of profiling information. Currently, RAWCC uses the length of the longest path for this purpose.

The Raw compiler currently limits the application of macroinstruction to the control localization of forward control flow. Regions with backward control flow present complications because static memory operations cannot be easily extracted while maintaining the static reference property. Even with this restriction, two important policy decisions remain. First, RAWCC must decide the scope of the application, *i.e.,* to which control flow structure should control localization be applied. This decision shares some similarity with the instruction partitioning problem. By applying control localization to a sequence of instructions, RAWCC is trading off parallelism within that region for the reduction in global branches. Global branches are costly for two reasons. They require global synchronization, and they serve as barriers to the scheduling of ILP. We have found this tradeoff to be worthwhile in at least two situations: in inner loops where global branches prevent unrolling from exposing schedulable parallelism, and for small control flow constructs containing little parallelism.

Second, RAWCC must decide the granularity of control localization application, *i.e.,* whether to convert a selected control flow structure to a single or multiple macroinstructions. This decision requires analyzing the cost of the macroinstruction abstraction, arising from the requirement that communication with other instructions can only occur at the boundaries of the macroinstruction. For example, a large macroinstruction must wait for the communication of all its input values before it can execute, even if one of these values is not needed until very late in the execution. Breaking up a control flow structure into multiple macroinstructions would reduce this overhead, at a cost of executing more local branches.

Currently, RAWCC adapts the simple policy of localizing every forward control flow structure into a

single macroinstruction. This policy is sufficient to attain the speedup reported in Section 5, but a more general policy is necessary to handle other applications. We intend to conduct a comprehensive study of this issue in the future.

It is worthwhile to compare predicated execution to control localization. In terms of flexibility, predicated execution requires fine-grained *if-conversion* [2], while control localization can support coarse-grained if-conversion. In terms of execution cost, predicated execution incurs the cost of executing all paths; an instruction which is nullified still occupies a unit of execution resources. In contrast, control localization consumes the amount of resources needed for the execution of the path that actually occurs. In terms of hardware support, predicated execution has two requirements. It requires support from the ISA, and it requires an extra predicated register file to alleviate the register pressure introduced by predicates. Control localization, on the other hand, only requires the presence of independent instruction streams.

4.2 Asynchronous global branching

The Raw machine implements global branching asynchronously in software by using the static network and local branches. First, the branch value is broadcasted to all the tiles through the static network. This communication is exported and scheduled explicitly by the compiler just like any other communication, so that it can overlap with other computation. Then, each tile and switch individually performs a branch without synchronization at the end of its basic block execution. The asynchrony is permitted because it does not change the static ordering of communication, and thus correct semantics is ensured through the static ordering property. As shown in Section 5, asynchrony gives the Raw machine the ability to tolerate dynamic events. Since the branch asynchrony is known at compile time, the Raw compiler can even generate an instruction schedule which explicitly takes advantage of this information.

Asynchronous global branching does not interfere with application of existing ILP enhancing techniques such as trace scheduling. To the first order, it can be treated abstractly as a global branch; the only difference lies in the cost of this branch relative to other instructions.

The overhead of global branching on the Raw machine is explicit in the broadcast of the branch condition. This contrasts with the implicit overhead of global wiring incurred by global branching in VLIWs and superscalars. Raw's explicit overhead is desirable for two reasons. First, the compiler can hide the overhead by overlapping it with useful work. Second, the approach is consistent with the Raw philosophy of eliminating all global wires, which taken as a whole enables a much faster clock speed.

5 Results

This section presents some performance results of the Raw compiler. We show the performance of the Raw compiler as a whole, and then we measure the portion of the performance due to high level transformations and advanced locality optimizations. In addition, we study how multisequentiality can reduce the sensitivity of performance to dynamic disturbances.

Experiments are performed on the Raw simulator, which simulates the Raw prototype described in Section 2. Latencies of the basic instructions are as follows: 1-cycle integer add or subtract; 12-cycle integer multiply; 35-cycle integer divide; 2-cycle floating add or subtract; 4-cycle floating multiply; and 12-cycle floating divide. Unless otherwise stated, the simulator assumes a perfect memory system with a two-cycle latency for a cache hit.

Benchmark	Source	Lang.	Lines of code	Primary Array size	Seq. RT (cycles)	Description
fpppp-kernel	Spec92	Fortran	735	-	8.98K	Electron Interval Derivatives
btrix	Nasa7:Spec92	Fortran	236	$15\times15\times15\times5$	287M	Vectorized Block Tri-Diagonal Solver
cholesky	Nasa7:Spec92	Fortran	126	$3\times32\times32$	34.3M	Cholesky Decomposition/Substitution
vpenta	Nasa7:Spec92	Fortran	157	32×32	21.0M	Inverts 3 Pentadiagonals Simultaneously
tomcatv	Spec92	Fortran	254	32×32	78.4M	Mesh Generation with Thompson's Solver
mxm	Nasa7:Spec92	Fortran	64	32×64, 64×8	2.01M	Matrix Multiplication
life	Rawbench	C	118	32×32	2.44M	Conway's Game of Life
jacobi	Rawbench	C	59	32×32	2.38M	Jacobi Relaxation

Table 1: Benchmark characteristics. Column *Seq. RT* shows the run-time for the uniprocessor code generated by the Machsuif MIPS compiler.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
fpppp-kernel	0.48	0.68	1.36	3.01	6.02	9.42
btrix	0.83	1.48	2.61	4.40	8.58	9.64
cholsky	0.88	1.68	3.38	5.48	10.30	14.81
vpenta	0.70	1.76	3.31	6.38	10.59	19.20
tomcatv	0.92	1.64	2.76	5.52	9.91	19.31
mxm	0.94	1.97	3.60	6.64	12.20	23.19
life	0.94	1.71	3.00	6.64	12.66	23.86
jacobi	0.89	1.70	3.39	6.89	13.95	38.35

Table 2: Benchmark Speedup. Speedup compares the run-time of the RAWCC-compiled code versus the run-time of the code generated by the Machsuif MIPS compiler.

The benchmarks we select include programs from the Raw benchmark suite [3], program kernels from the nasa7 benchmark of Spec92, tomcatv of Spec92, and the kernel basic block which accounts for 50% of the run-time in fpppp of Spec92. Since the Raw prototype currently does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. Table 1 gives some basic characteristics of the benchmarks.

Speedup We compare results of the Raw compiler with the results of a MIPS compiler provided by Machsuif [15] targeted for an R2000. Table 2 shows the speedups attained by the benchmarks for Raw machines of various sizes. Note that these speedups do not measure the advantage Raw is attaining over modern architectures due to a faster clock. The results show that the Raw compiler is able to exploit ILP profitably across the Raw tiles for all the benchmarks. The average speedup on 32 tiles is 19.7.

All the benchmarks except fpppp-kernel are dense matrix applications. These applications perform particularly well on a Raw machine because arbitrarily large amount of parallelism can be exposed to the Raw compiler by unrolling the loop. Currently, the Raw compiler unrolls loops by the minimum amount required to guarantee the static reference property referred to in Section 3.1, which in most of these cases expose as many copies of the inner loop for scheduling of ILP as there are the number of processors. The only exception is btrix. Its inner loops process array dimensions of either five or fifteen. Therefore, the maximum parallelism exposed to the basic block orchestrater is at most five or fifteen.

Many of these benchmarks have been parallelized on multiprocessors by recognizing do-all paral-

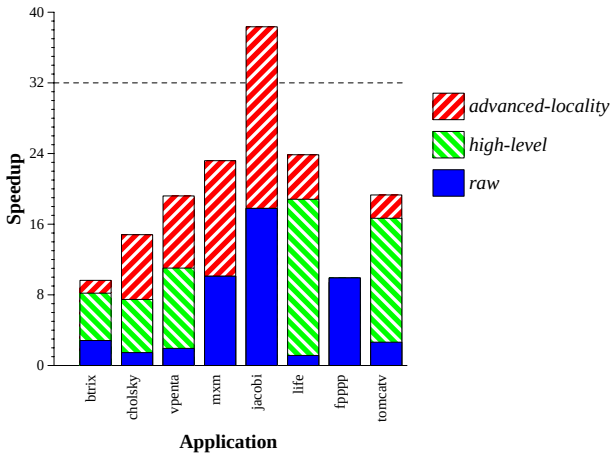

Figure 6: Breakdown of speedup for 32 processors into *raw* component, *high-level* component, and *advanced-locality* component.

lelism and distributing such parallelism across the processors. Raw detects the same parallelism by partially unrolling a loop and distributing individual instructions across tiles. The Raw approach is more flexible, however, because it can schedule do-across parallelism contained in loops with loop carried dependences. For example, several loops in tomcatv contain reduction operations, which are loop carried dependences. In multiprocessors, the compiler needs to recognize a reduction and handle it as a special case. The Raw compiler handles the dependence naturally, the same way it handles other arbitrary loop carried dependences.

The size of the datasets in these benchmarks is intentionally made to be small to feature the low communication overhead of Raw. Traditional multiprocessors, with their high overheads, would be unable to attain speedup for such datasets [1].

Most of the speedup attained can be attributed to the exploitation of ILP, but unrolling plays a beneficiary role as well. Unrolling speeds up a program by reducing its loop overhead and exposing scalar optimizations across loop iterations. This latter effect is most evident in jacobi and life, where consecutive iterations share loads to same array elements which can be optimized through common subexpression elimination.

Fpppp-kernel is different from the rest of the applications in that it contains irregular fine-grained parallelism. This application stresses the locality/parallelism tradeoff capability of the instruction partitioner. For the fpppp-kernel on a single tile, the code generated by the Raw compiler is significantly worse than that generated by the original MIPS compiler. The reason is that the Raw compiler attempts to expose the maximal amount of parallelism without regard to register pressure. As the number of tiles increases, however, the number of available registers increases correspondingly, and the spill penalty of this instruction scheduling policy reduces. The net result is excellent speedup, occasionally attaining more than a factor of two speedup when doubling the number of tiles.

Speedup breakdown Figure 6 divides the speedup for 32 processors for each application into three components: *raw*, *high-level*, and *advanced-locality*. The *raw* speedup is the speedup from a bare compiler which uses simple unrolling and moderate locality optimization.

High-level shows the additional speedup when the bare compiler is augmented with high level trans-

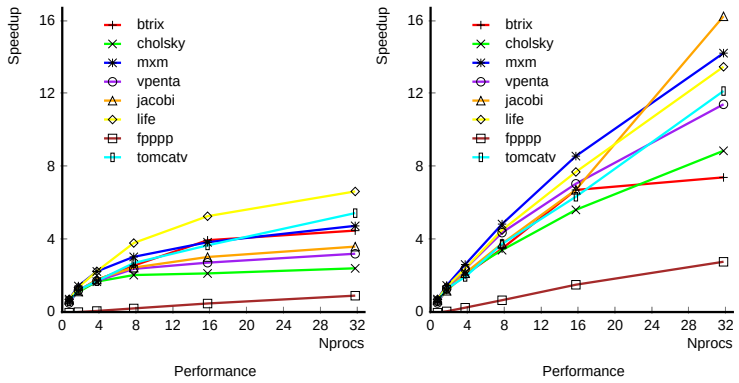

Figure 7: Speedup of applications in the presence of dynamic disturbances for two machine models. The left graph shows results for a machine with a single pc which must stall synchronously; the right graph shows results for a Raw machine with multiple pcs which can stall asynchronously.

formations, which include control localization and array reshape.[1] Array reshape refers to the technique of tailoring the layout of an array to avoid memory bank conflicts between accesses in consecutive iterations of a loop. It is implemented by allocating a tailored copy of an array to a loop when the loop has a layout preference which differs from the actual layout. This technique incurs the overhead of array copying between the global and the tailored array before and after the loop, but most loops do enough computation on the arrays to make this overhead worthwhile. Btrix, cholesky, and vpenta benefit from this transformation, while life and tomcatv get their speedups from control localization.

Advanced-locality shows the performance gain from advanced locality optimizations. The optimizations include the use of locality sensitive algorithms for data partitioning and for the merging phase during instruction partitioning. The figure shows that all applications except btrix and fpppp attain sizable benefits from these optimizations. The average performance gain of all the applications is 60%.

Effects of dynamic events We study the effects of dynamic disturbances such as cache misses on a Raw machine. We model the disturbances in our simulator as random events which happen on loads and stores, with a 5% chance of occurrence and an average stall time of 100 cycles. We examine the effects of such disturbances on two machine models. One is a faithful representation of the Raw machine; the other models a synchronous machine with a single instruction stream. On a Raw machine, a dynamic event only directly effects the processor on which the event occurs. Processors on other tiles can proceed independently until they need to communicate with the blocked processor. On the synchronous machine, however, a dynamic event stalls the entire machine immediately.[2] This behavior is similar to how a VLIW responds to a dynamic event.[3]

Figure 7 shows the performance of each machine model in the presence of dynamic events. Speedup is measured relative to the MIPS-compiled code simulated with dynamic disturbances. The results show that asynchrony on Raw reduces the sensitivity of performance to dynamic disturbances. Speedup for the Raw machine is on average 2.9 times better than that for the synchronous machine. In absolute terms,

[1] Array reshape is currently hand-applied; it is in the process of being automated.

[2] In this experiment, the synchronous architecture is allowed asynchrony arising from communication.

[3] Many VLIWs support non-blocking stores, as well as loads which block on use instead of blocking on miss. These features reduce but do not eliminate the adverse effects of stalling the entire machine, and they come with a potential penalty in clock speed.

the Raw machine still achieves respectable speedups for all applications. On 32 processors, speedup on fpppp is 3.0, while speedups for the rest of the applications are at lest 7.6.

6 Related work

Due to space limitations, we only discuss past work that is closely related to the problem of space-time scheduling of instruction-level parallelism, which is the focus of this paper.

The Bulldog compiler [6] targets a VLIW machine consisting of clusters of functional units, register files, and memory connected via a partial crossbar. Bulldog adopts a two-step approach for space-time scheduling in which instructions are first mapped spatially and then temporally. Spatial mapping is performed by an algorithm called Bottom-Up Greedy (BUG), a critical-path based mapping algorithm that uses fixed memory and data nodes to guide the placement of other nodes. Like the approach adopted by the clustering algorithm in RAWCC, BUG visits the instructions topologically, and greedily attempts to schedule each instruction on the processor that is locally the best choice. Temporal scheduling is then performed by a greedy list scheduling algorithm.

The key differences between the approaches taken in the Bulldog and RAWCC compilers are as follows. First, BUG attempts to find a spatial mapping of instructions that simultaneously addresses critical path, data affinity, and processor preference issues in a single step. RAWCC, on the other hand, performs spatial mapping in two steps; the Instruction Partitioner divides the available parallelism into threads and the Instruction Placer considers where the threads should be placed given that some instructions are locked to specific processors. Second, the spatial mapping performed by BUG is driven by a greedy depth-first traversal that maps all instructions in a connected subgraph with a common root before processing the next subgraph. As observed in [12], such a greedy algorithm is often inappropriate for parallel computations such as those obtained by unrolling parallel loops. In contrast, instruction partitioning and placement in RAWCC uses a global priority function that can intermingle instructions from different connected components of the dag. Third, the temporal scheduling performed by RAWCC is more general than the scheduling in Bulldog because it schedules both computation and communication instructions.

Compilation for some other kinds of clustered VLIW architectures have also been considered in past work. For example, a brief description of compilation for an LC-VLIW (Limited Connectivity VLIW) architecture can be found in [5]. This approach includes partitioning of instructions across clusters as well as insertion of explicit inter-cluster register-to-register data movement instructions. However, there are significant differences between the LC-VLIW machine model and a Raw machine. Though an LC-VLIW machine has partitioned/distributed register files, its machine model assumes uniform access to main memory unlike Raw. Further, an LC-VLIW machine assumes a single VLIW instruction stream, while Raw is multisequential.

Many ILP-enhancing techniques have been developed to increase the amount of parallelism available within a basic block. These techniques include control speculation [9], data speculation [16], trace/superblock scheduling [7] [13], and predicated execution [2]. Several characteristics of Raw affect the application of these techniques. In Raw, a global branch is more costly relative to other instructions because it requires an explicit broadcast followed by local branches. This cost is reflected in the side exits of traces. To apply trace or superblock scheduling to Raw, one needs to reduce these side exits. On the other hand, control localization obviates the need for predicated execution on Raw: it permits local execution of more general control flow constructs, and it does not require support from the ISA.

Control localization is similar to the technique of hierarchical reduction [10]. They both share the

basic idea of collapsing control constructs into a single abstract node. They differ in application and in details. First, hierarchical reduction is used for software pipelining on VLIWs; control localization is used for general instruction scheduling on the Raw machine. Second, control localization requires promotion of memory operations, while hierarchical reduction does not. Third, control localization uses a more flexible approach which allows control constructs to be converted to multiple nodes. Most importantly, control localization takes advantage of the presence of independent instruction streams to schedule the execution of independent control flow constructs in parallel.

7 Conclusion

This paper presents the Raw architecture and compiler system. Together, they exploit instruction-level parallelism in applications to achieve high levels of performance by physically distributing resources and exposing them fully. The resulting non-uniform access times create many challenging compiler problems. This paper focuses on the space-time scheduling of instruction-level parallelism brought about by the physically distributed register files with non-uniform register access times. The Raw compiler detects and orchestrates instruction level parallelism within basic blocks by adapting techniques from MIMD task partitioning and scheduling, as well as from FPGA communication scheduling. It employs a decentralized control flow model, where control is localized when possible and global branching is performed explicitly and asynchronously.

A Raw compiler based on SUIF has been implemented. The compiler accepts sequential C or Fortran programs, discovers instruction-level parallelism, and schedules the parallelism across the Raw substrate. Reflecting our ILP focus, the compiler currently does not exploit coarse-grain parallelism through SUIF's loop-level parallelization passes, although it can be easily extended to do so. This paper presents some promising results on the potential of the Raw compiler. The compiler is capable of realizing 9-way parallelism on 32 tiles for fpppp-kernel, an application with irregular, fine-grained parallelism. We find that Raw machine's decoupled control flow enables it to tolerate dynamic events such as cache misses. Since a Raw microprocessor is already suitable for many other forms of parallelism, including coarse-grain loop level parallelism, stream processing, and static pipelines, its ability to exploit ILP is an important step in demonstrating that the Raw machine is an all-purpose parallel machine.

References

[1] S. Amarasingle, J. Anderson, C. Wilson, S. Liao, B. Murphy, R. French, and M. Lam. Multiprocessors from a Software Perspective. *IEEE Micro*, pages 52–61, June 1996.

[2] D. August, W. mei Hwu, and S. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1997.

[4] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[5] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[6] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. In *Ph.D Thesis, Yale University*, 1985.

[7] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 7(C-30):478–490, July 1981.

[8] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, pages 11–16, Oct. 1996.

[9] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. In *HP Laboratories Technical Report 93-80*, Feb 1994.

[10] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. Int'l Conf. on Programming Language Design and Implementation (PLDI)*, pages 318–328, June 1988.

[11] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[12] P. G. Lowney and et al. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, January 1993.

[13] W. mei Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1), Jan 1993.

[14] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[15] M. D. Smith. Extending suif for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, Jan. 1996.

[16] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[17] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All To Software: Raw Machines. *Computer*, pages 86–93, Sept. 1997.

[18] R. Wilson and et al. SUIF: A Parallelizing and Optimizing Research Compiler. *SIGPLAN Notices*, 29(12):31–37, December 1994.

[19] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

A Static Ordering Property

Dynamic events such as cache misses prevent one from statically analyzing the precise timing of a schedule. The Raw compiler relies on the static ordering property of the Raw architecture to generate correct code in the presence these dynamic events. The static ordering property states that the result produced by a static schedule is independent of the specific timing of the execution. Moreover, it states that whether a schedule deadlocks is a timing independent property as well. Either the schedule always deadlocks, or it never does.

To generate a correct instruction schedule, Raw orders the instructions in a way that obeys the instruction dependencies of the program. In addition, it ensures that the schedule is deadlock free assuming one set of instruction timings. Static ordering property then ensures that the schedule is deadlock free and correct for any execution of the schedule.

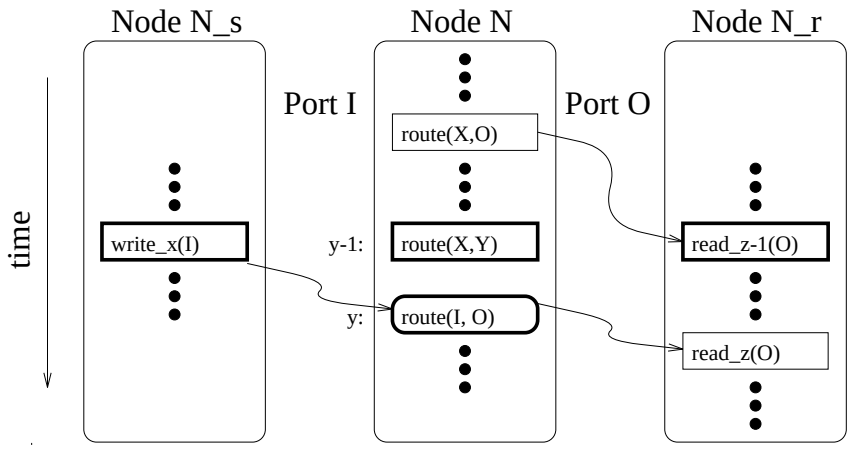

Figure 8: **Dependent instructions of a communication instruction.** Each long rectangle represents an execution node, and each wide rectangle represents an instruction. Spaces between execution nodes are ports. Edges represent flow of data. The focal instruction is the thick rounded rectangle. Its dependent instructions are in thick regular rectangles.

We provide an informal proof of the static ordering property. We restrict the static ordering property to the practical case: given a schedule that is deadlock free for one set of instruction timings, then for any set of instruction timings,

1. it is deadlock free.

2. it generates the same results.

First, we show (1). A deadlock occurs when at least one instruction stream on either the processor or the switch has unexecuted instructions, but no instruction stream can make progress. A non-empty instruction stream, in turn, can fail to make progress if it is attempting to execute a blocked communication instruction. A communication instruction blocks when either its input port is empty, or its output port is full. Computation instructions do not use communication ports; they cannot cause deadlocks and are only relevant in this discussion for the timing information they represent.

Consider a communication instruction c. We derive the conditions under which it can execute. Three resources must be available: its input value, its execution node (processor or switch), and its output ports.[4] The resource requirements can also be represented by execution of a set of instructions. First, note that ports are dedicated connections between two fixed nodes, so that each port has exactly one reader node and one writer node. Let instruction c be the x^{th} instruction that reads its input port I, the y^{th} instruction that executes on its node N, and the z^{th} instruction that writes its output port O. Then the resources for instruction c become available after the following instructions have executed:

1. the x^{th} instruction that writes port I.

2. the $y - 1^{th}$ instruction that executes on node N.

3. the $z - 1^{th}$ instruction that reads (and *flushes*) port O.

See Figure 8.

The key observation is that once a resource becomes available for instruction c, it will *forever* remain

[4]The three resources need not all be applicable. A *send* instruction only requires an output port and a node, while a *receive* instruction only requires the input value and a node.

available until the instruction has executed. The value on the input port cannot disappear; the execution node cannot skip over c to run other instructions; the output port cannot be full after the previous value has been flushed. The reservation of the resources is based on three properties: the single-reader/single-writer property of the ports, the blocking semantics of the communication instructions, and the in-order execution of instructions.

Therefore, a communication instruction can execute whenever its dependent instructions, defined by the enumeration above, have executed.

Now, consider the schedule that is deadlock-free for one known set of timings. Plot the execution trace for this set of timings in a two dimensional grid, with node-id on the x-axis and time on the y-axis. Each slot in the execution trace contains the instruction (if any) that is executed for the specified node at the specified time. The plot is similar to Figure 8, except that real execution times, rather than the static schedule orders, are used to place the instructions temporally.

Finally, consider a different set of timings for the same schedule. Let t_{new} be a point in time for the new timings when the schedule has not been completed, and let $E_{new}(t_{new})$ be the set of instructions that have executed before time t_{new}. We use the above deadlock-free execution trace to find a runnable instruction at time t_{new}. Find the smallest time t in the deadlock-free execution trace that contains an instruction not in $E_{new}(t_{new})$. Call the instruction c. The dependent instructions of c must necessarily be contained in $E_{new}(t_{new})$.[5] Therefore, c must be be runnable at time t_{new} for the new set of timings.

Having found a runnable instruction for any point in time when the schedule is not completed, the schedule must always make progress, and it will not deadlock.

The second correctness condition, that a deadlock-free schedule generates the same results under two different sets of timings, is relatively easy to demonstrate. Changes in timings do not affect the order in which instructions are executed on the same node, nor do they change the order in which values are injected or consumed at individual ports. The blocking semantics of communication instructions ensures that no instruction dependence can be violated due to a timing skew between the sender and the receiver. Therefore, the values produced by two different timings must be the same.

[5] This statement derives from two facts:

1. All dependent instructions of c must execute before c in the deadlock-free execution trace.

2. Since c executes at time t and all instructions executed before time t are in $E_{new}(t_{new})$, all instructions executed before c in the deadlock-free execution trace are in $E_{new}(t_{new})$.