

Integrating Data Caching into the SUDS Runtime System

by

Kevin W. Wilson

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Massachusetts Institute of Technology 2000. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by
Anant Agarwal
Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Integrating Data Caching into the SUDS Runtime System

by

Kevin W. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I describe the design and implementation of a software data cache for speculative loads in the *SUDS* (Software Undo System) runtime system. The basic functionality of this cache exploits short-term temporal locality and some spatial locality in the memory accesses of speculative processing elements. The cache implementation is also extended to exploit temporal locality of accesses on a longer time scale. Using four sample applications, I compare the performance of the caching version of the SUDS runtime system to the performance of the original SUDS runtime system and to the performance of optimized sequential versions of the applications. With software data caching, the sample applications demonstrate speedups of 20% to 300% compared to SUDS without caching. In addition, software caching allows SUDS to demonstrate parallel speedups on three of the four applications.

Thesis Supervisor: Anant Agarwal

Title: Professor

Acknowledgments

I would like to thank my advisor, Anant Agarwal, for giving me the freedom to explore the material presented in this thesis.

I would like to thank Walter Lee and Michael Taylor for “showing me the ropes” of the Raw project.

I would like to thank Matt Frank for creating SUDS, for numerous discussions during which many of the concepts described in this thesis were conceived of and/or refined, and for generally providing guidance in my attempt to extend the SUDS runtime.

I would like to thank my parents and my sister for supporting me during the creation of this thesis as they have in all of my academic and personal endeavors.

Contents

1	Introduction	7
2	Background	9
2.1	SUDS	9
2.2	Raw	12
3	Implementation	14
3.1	Design Decisions	14
3.1.1	Correctness	14
3.1.2	Performance	15
3.2	Implementation Details	18
4	Results	22
5	Discussion	30
6	Conclusion	35

List of Figures

2-1	The Raw processor consists of a grid of tiles. Each tile consists of a simply pipelined MIPS-like processor, small local memories, and a network interface.	12
4-1	Speedups for the parallelized portion of each application program. Results for 4-, 8-, and 16-compute-node configurations are shown; in each case, the number of memory nodes is chosen to be twice the number of computed nodes. “Uncached” denotes the original SUDS runtime, before the implementation of caching. “Intra-chunk caching” denotes the runtime system with the capability to cache speculative data within a chunk. “Intra-chunk and permanent caching” denotes the runtime system with the additional ability to cache data permanently, thus allowing the compute node to retain cached data from one chunk to the next.	24

List of Tables

3.1	Approximate breakdown of operation costs for an uncached load between a compute node and an adjacent memory node.	16
4.1	Cache hit rates for the parallelized loops in each application for a four-compute-node configuration.	26
4.2	Latency measured from the sending of the load request to the receipt of the load reply for a four-compute-node configuration.	27
4.3	Inter-chunk memory node operation costs for Jacobi. Costs are per-line for the 8-word cache line and per-word for the single-word case. . . .	27

Chapter 1

Introduction

The era of billion-transistor chips is approaching, and in order to exploit the computational resources of such chips, scalable methods of exploiting parallelism must be developed [11]. Modern superscalar processors can detect and exploit instruction-level parallelism within a relatively small window of machine instructions, but many of the techniques used by superscalars do not scale well. Compiler analysis can be used to automatically parallelize code that can be proven to be free of data dependences between parts of the code that are executed in parallel, but there are a wide range of applications for which this analysis becomes difficult or impossible.

Another approach to exploiting parallelism in an application is to speculatively execute parts of the application in parallel and to detect if any events occurred during this parallel execution that could cause the parallel execution to be inconsistent with the sequential execution of the code [4]. If such an event occurs, the effects of any incorrect speculation must not be allowed to affect the non-speculative state of the machine. Speculation is performed on a small scale by modern superscalars, but parallelism can also be exposed by speculation at much larger scales.

The need to separate speculative state from non-speculative state introduces additional complexity into systems that support speculation. In the case of speculation across memory accesses that may conflict, if loads and stores to a particular memory location happen in an order that is not consistent with their ordering during sequential execution, the system must detect this and ensure that the effects of the

incorrectly executed code are discarded. The computational complexity and memory requirements of such bookkeeping make it difficult to justify a system implemented completely in hardware. It is possible, though, to speculatively execute at larger scales through the use of appropriate compiler analysis and software runtime support.

The Software Undo System (SUDS) [3] is an example of such a system. SUDS provides mechanisms to support speculative parallel execution of multiple loop iterations. Because extra processing must be done in order to detect incorrect orderings among speculative loads and stores and to undo the effects of incorrect stores, speculative memory accesses are much more expensive than equivalent non-speculative operations, and this high cost of memory operations is a major obstacle to achieving parallel speedups with SUDS. One technique for ameliorating this problem is to amortize the cost of bookkeeping across several memory accesses by caching speculative loads.

This thesis presents and evaluates three techniques for facilitating efficient caching of speculative loads within the SUDS framework. The first, intra-chunk caching, exploits short-term temporal locality. The second, permanent caching, exploits longer-term temporal locality for read-only or read-mostly data. The third, multi-granularity caching, provides a mechanism for exploiting spatial locality while allowing for fine-grained sharing of data. An empirical evaluation of the caching system demonstrates significant performance improvements on four sample applications, and further analysis provides insight into the advantages of individual techniques.

Chapter 2 provides background information on SUDS and the Raw processor. Chapter 3 describes the design and implementation of the software data caching extension. Chapter 4 describes the testing methodology used to evaluate the caching extension and the results of that evaluation. Chapter 5 provides a high-level overview of the results and in light of the constraints placed on the cache by the SUDS runtime environment. Chapter 6 concludes.

Chapter 2

Background

In this chapter, I give an overview of the two systems upon which my software caching system has been built. The software environment within which my work must fit is the SUDS runtime system, and the hardware for which SUDS is currently implemented is the Raw processor.

2.1 SUDS

SUDS is a system for parallelizing loops that may contain true or potential memory dependences across iterations [3]. It consists of two main components, the first of which is a set of compile-time analyses and program transformations that takes a sequential program as input and outputs a speculatively parallel version of the program. The second component is a software runtime system that implements the mechanisms that the parallelized code uses to ensure correctness in the presence of actual and/or potential memory dependences.

In the SUDS model of execution, a set of processing nodes is partitioned into a set of *compute nodes* and a set of *memory nodes* which communicate with each other through message-passing. One of the compute nodes, the *master node*, is responsible for executing all sequential code in the application and for directing the behavior of all other nodes during parallel loops. For such loops, loop iterations are cyclically distributed across the compute nodes and executed in parallel. A set of iterations

running in parallel on the compute nodes is referred to as a *chunk*. To simplify the detection of dependence violations, all nodes are synchronized between chunks.

To allow for efficient parallelization of loops with data dependences between iterations, the compile-time component of SUDS privatizes all memory accesses that it can verify as being local to a single iteration. It also inserts explicit communication between compute nodes for communicating loop-carried dependences that can be identified at compile-time. All memory accesses that cannot be adequately analyzed at compile time are transformed into speculative memory accesses.

Speculative memory accesses require the compute nodes to send request messages to the memory nodes, which are responsible for performing the operations requested by the compute nodes and for maintaining the timestamp information that allows the system to detect dependence violations. In the event of a dependence violation, the memory node detecting the violation informs the master node that a violation has occurred, and the master node directs all memory nodes to roll back their state to the state that existed at the start of the chunk. The master node then sequentially executes all iterations from the failed chunk and returns to parallel execution for subsequent chunks. If no violations were detected by any of the memory nodes, and if all iterations in the chunk completed successfully, the master node will request that the memory nodes commit all writes permanently into the speculative memory and clear their logs and hash tables.

In order to detect and correct dependence violations, the memory nodes maintain three main data structures. The first is a large array that serves as the speculative memory itself. Memory access requests from the compute nodes are satisfied by corresponding accesses to this array. The second data structure is a log of all of the memory locations that have been accessed during the current chunk. This log records the data that was stored in each location at the beginning of each chunk so that, in the event of a dependence violation, the log can be used to restore a consistent state to the memory. The third data structure is a hash table that contains last-reader and last-writer timestamps for each memory location that has been accessed in the current chunk. This timestamp information is updated as appropriate for each speculative

memory request issued by a compute node, and by comparing timestamps in the hash table to the timestamps of incoming requests, dependence violations can be detected.

In the basic SUDS runtime system, addresses in the speculative address space are cyclically distributed across the memory nodes at single-word granularity. When a compute node encounters a speculative memory access, the following sequence of events takes place:

1. The compute node hashes the address to determine which memory node owns it.
2. The compute node sends a request to the appropriate memory node.
3. The memory node receives the request and dispatches to the appropriate message handler.
4. In the case of a load request, the appropriate data is retrieved from the memory array and sent back to the compute node.
5. The address is hashed, and the timestamps in the hash table are compared to the timestamp of the current request.
6. If this is the first access to the location, a log entry is created.
7. If a dependence violation is detected, a flag is set. Otherwise, the hash table time stamps are updated.
8. Memory updates are performed in the case of write requests.

Speculative memory accesses are expensive in SUDS for two reasons. The first reason is that extra bookkeeping must be done in order to detect dependence violations and return the system to a consistent state. During the execution of a chunk, this extra bookkeeping has the effect of reducing the memory bandwidth of the memory nodes, and between chunks, this bookkeeping requires memory nodes to spend time clearing data structures in preparation for the next chunk. The second reason is that, in a simple implementation, all speculative accesses involve remote (inter-node)

Raw μ P

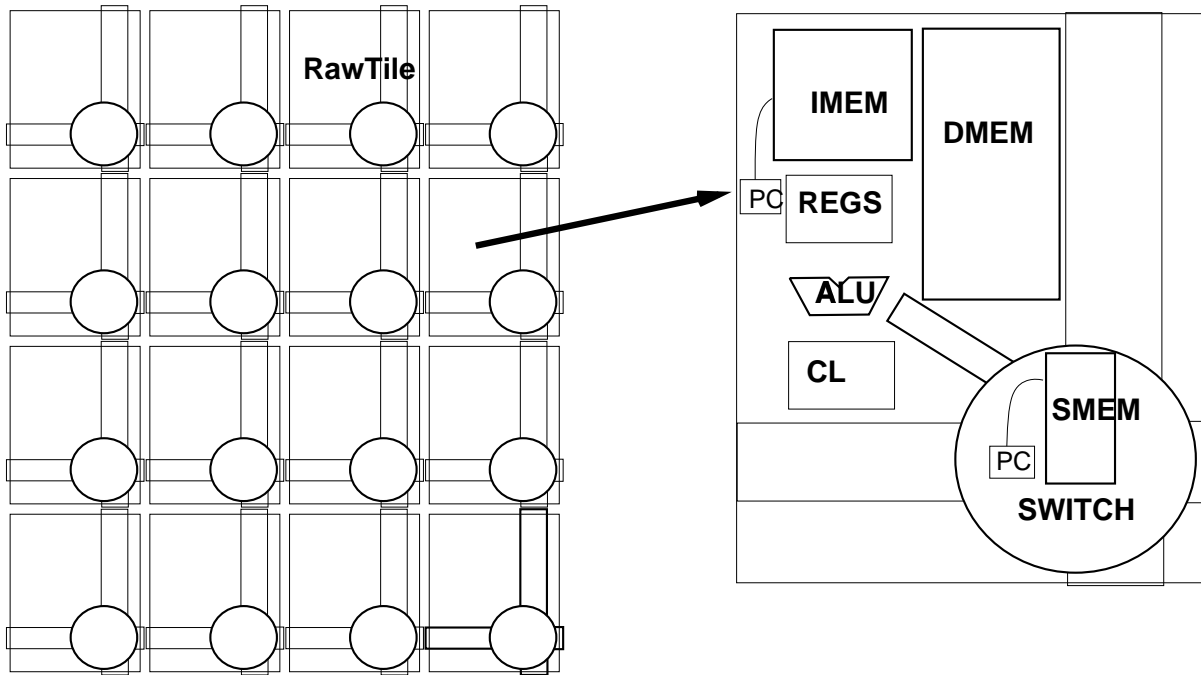


Figure 2-1: The Raw processor consists of a grid of tiles. Each tile consists of a simply pipelined MIPS-like processor, small local memories, and a network interface.

communication. This thesis describes a way to relax the second constraint by adding a small cache to each compute node to exploit memory access locality.

2.2 Raw

The current incarnation of the SUDS runtime system is implemented for the Raw processor [10]. The Raw processor is designed to provide scalable computational power and to provide an effective mechanism for exposing this computational power to software.

The Raw architecture is implemented as a grid of tiles connected by two types of low latency mesh network, referred to as the static networks and the dynamic networks (see Figure 2-1). Each tile contains a simply pipelined MIPS-like processor, a small local memory, and a very simple auxiliary switch processor that is responsible for routing data on Raw's static networks.

Raw's static networks are designed to support messages whose ordering and destinations can be determined at compile time. Because SUDS parallelizes loops with statically unpredictable data dependences, the static network is not well suited to most of the communication that the SUDS runtime system must perform. Instead, SUDS relies on Raw's user-level dynamic network, which is a dimension-ordered, wormhole routed, flow-controlled network suitable for use when message destinations and orderings are not known at compile time.

Because Raw tiles' local memories are small (32kb each for instruction memory and data memory), a mechanism must be provided to allow tile processors to access large external memories. Raw accomplishes this by using each tile's local data memory as a hardware data cache. Misses cause messages to be sent to off-chip memory controllers over the memory dynamic network, which is distinct from the user-level dynamic network but identical in its implementation.

From the viewpoint of SUDS, each Raw tile is a node. The tiles that serve as memory nodes all run identical code which implements a simple dispatch loop for handling memory requests. The tiles that serve as compute nodes execute code generated by the SUDS compile-time component along with code from the runtime system that implements the operations that the compile-time component depends on in order to deal with dependence violations. SUDS uses Raw's user-level dynamic network for all communication, including speculative memory accesses, communication of loop-carried dependences between compute nodes, and synchronization messages between the master node and all other nodes.

Chapter 3

Implementation

This chapter provides an overview of the major influences on the design of the software data cache and a description of the implementation of the system.

3.1 Design Decisions

A successful caching system must satisfy two high-level goals. It must maintain the correctness of the system, and it must improve the performance of the system.

3.1.1 Correctness

In many cases, distributed systems must employ sophisticated cache coherence protocols to ensure correct execution of parallel programs [1]. However, in SUDS, it is possible to use the memory speculation mechanism to detect and correct coherence problems without further complicating the protocol.

A simple discipline that can be used to ensure that data caching does not cause incorrect behavior is to invalidate each compute node's cache after each chunk. Doing this guarantees that at least the first load of each memory location in each chunk must contact the appropriate memory node. This lets the memory node know that that compute node is using the data so that the memory node can update its timestamp information appropriately. Caching will (hopefully) reduce the number of loads of a

given address within each chunk, but this cannot lead to incorrect behavior. This is because each compute node is executing what was originally one iteration of a sequential loop, so if a compute node receives the correct value from a memory node, that value will remain the correct value for that compute node for the remainder of the chunk. If the compute node receives an incorrect value from the memory node as a result of a dependence violation, the memory node will eventually discover this just as it would have in the uncached case.

Because the memory nodes clear their hash tables and logs between chunks, it is not as easy to cache data on the compute nodes from one chunk to the next. Doing this without modifying the rest of the runtime system would make it possible for compute nodes to use stale data without the memory nodes being able to determine this.

A simple extension to the runtime that permits the caching of data from one chunk to the next is to create a new type of request that compute nodes can send to memory nodes that I will refer to as a *permanently cached load*. When a memory node sees a request of this type, it can mark that entry in the hash table as permanently cached, and when the time comes to clean the hash table, it can leave all permanently cached hash table entries intact.

Whenever a permanently cached address is written, a violation must be flagged, and a rollback must occur. Because a dependence violation may have resulted from a write to a permanently cached location, compute nodes must completely flush their caches of all data, including permanently cached data, whenever a rollback occurs.

3.1.2 Performance

The software data cache's purpose is to improve the performance of the system, but before examining the performance implications of a cache, the performance of the uncached runtime system should be examined. Table 3.1 breaks down the cost of an uncached speculative load. In addition to these costs, which lie on the critical path between the time the speculative load is initiated and the time that the data is returned, the memory node must spend roughly 40 cycles updating its hash table and

Action	Approximate cost (cycles)
Function call and request preparation	23
Request time-of-flight	6
Memory node reply preparation	18
Reply time-of-flight	6
Total	53

Table 3.1: Approximate breakdown of operation costs for an uncached load between a compute node and an adjacent memory node.

possibly allocating a log entry for the memory location. This 53-cycle critical-path cost is a best case cost since it is for adjacent nodes in the absence of network or memory node contention.

Adding a local data cache to each compute node’s software runtime system affects the system in several ways, and it is not immediately obvious whether the net effect will be an improvement. Software caching will add tag-check overhead to all cached accesses, which guarantees that a cache miss will be more expensive than an uncached load. Because the cache is implemented in software, even cache hits will be significantly more expensive than memory accesses that have been privatized by the SUDS compile-time subsystem. In addition, the hit rate of a compute node cache will likely be lower than hit rates for similar caches in non-speculative systems because caches must be invalidated at the end of each chunk to ensure program correctness.

In order to achieve reasonable hit rates on a range of applications, it is desirable to exploit spatial locality of accesses by caching multi-word lines. The use of multi-word lines may, however, lead to false sharing, which would increase the number of violations detected by the runtime system. The cost of correcting a dependence violation varies greatly from situation to situation, so it is not possible in general to express quantitatively the trade-off between the increase in hit rate and the increase in misspeculation rate that both come with increasing cache line size. It is clear, however, that the cost of misspeculation is large, so great care must be taken to avoid misspeculation due to false sharing.

The conflicting goals of achieving high cache hit rates and low misspeculation

rates can both be achieved through the use of two separate caches, one with single-word cache lines and one with larger, multi-word, cache lines. The caching system described in this thesis uses such a scheme.

When using two separate caches, there must be some way of determining which cache to use for each memory access. In the system described in this thesis, the speculative address space is partitioned into two segments such that all accesses to one segment are cached at single-word granularity and all accesses to the other segment are cached at multi-word granularity. In the current implementation, for each data object, the segment into which it is placed must be hand-selected at compile time. Hand partitioning avoids the problem of caching data with a “non-optimal” line size and thus demonstrates the potential performance of the system, but in the future, it will be desirable to have a more automated way of using multiple cache line sizes. If an object is placed into the “non-optimal” segment of the address space, the performance of the system may suffer severely because of increased misspeculations due to false sharing, but the execution will still be correct because the memory nodes will detect any true dependence violations in addition to the false violations.

Caching can also affect the ordering of the loads and stores seen by the memory nodes, which may have an effect on the frequency with which dependence violations occur. If a memory location is accessed multiple times within a chunk, the original SUDS runtime would send multiple requests to the memory node, and if any of those load requests reach the memory node after a write to that address from a compute node working on a subsequent iteration, a violation will result. Caching a particular address will tend to reduce the number of requests that occur later in the chunk, which will tend to reduce the number of violations.

The cache’s write policy must also be chosen based on how it interacts with the SUDS speculative execution scheme. The basic choice is between write-through and write-back, and two important facts argue for a write-through cache. The first is the pragmatic observation that the existing runtime code for both the compute nodes and memory nodes can be used nearly unmodified to support a write-through cache, while a write-back cache would require significant modifications to the runtime

system. Because write performance is not as critical as read performance, it does not make sense to spend time modifying a part of the system that will likely have only a minor effect on system performance. The second argument in favor of a write-through cache is that the cache line flushes required at the end of each chunk would lead to network congestion if every compute node must write back several cache lines after finishing a chunk. For these reasons, write-through with update was chosen as the cache write policy.

The permanent caching feature mentioned above will be most useful for programs that frequently access global scalars or small tables of values that are read-only or read-mostly. Read only data that is not accessed frequently is likely to be evicted from the cache before it can be reused, so it is unlikely that permanently caching such data would provide significant benefits.

Because of the SUDS dependence violation detection mechanism, the execution of a program is guaranteed to be correct in the presence of permanent caching even if the permanently cached location is written often. In this scenario, however, the additional rollbacks necessary to maintain memory consistency would severely degrade performance.

3.2 Implementation Details

This section describes the changes to the runtime system that were made in order to implement the caching strategies outlined in the previous section.

In the previous section, it was proposed that the address space be divided into two segments. A fast way to do this in software is to use the high bit of the address to indicate to which segment it belongs. This makes it possible to use a “branch if greater than or equal to zero” machine instruction to differentiate the two segments.

Each address range has its own direct-mapped cache consisting of an array of data structures containing cached data, cache tags, and status bits. Direct-mapped caches were chosen because they are simple to implement and because they allow faster tag checks than would be possible with a cache with a higher level of associativity. The

main drawback of direct-mapped caches is that they suffer from more conflict misses than would other cache implementations.

The segment of memory that is cached with multi-word cache lines can no longer be distributed at word granularity across the memory nodes. Instead, it is distributed at cache-line granularity across the memory nodes, which means that the hash function that is used to map addresses to memory nodes must be changed. For data cached at a coarse granularity, it no longer makes sense to keep coherency information on a per-word basis. Instead, coherency is kept on a per-line basis. This will tend to reduce the per-word overhead of bookkeeping operations for applications that exhibit good spatial locality.

Permanent caching requires only a few additional modifications to the runtime system. The first modification is that each cache line must include a status bit to indicate whether that line is permanently cached. Between chunks, each compute node must invalidate only those lines that are not permanently cached.

On the memory nodes, the changes required to support permanent caching are more complicated. The basic idea is that a hash entry should have its last-reader time stamp set to “infinity,” and instead of being recorded in the normal log, it should be recorded in a log dedicated to permanently cached data. The only minor complication occurs if, during the course of a chunk, one compute node first issues a normal read request and another compute node then issues a permanently cached read request for the same address. In that case, a log entry is entered into both of the logs. If a permanently cached read is issued before a normal read, only a permanent log entry is used. In the event of a rollback, it is necessary to roll back the permanent log first because some locations may be present in both logs. Thus, the only way that an address could be present in both logs is if it is read normally within a chunk before being permanently cached. In this case, the normal log will contain the correct roll back state, so rolling back the normal log after the permanent log ensures that the correct state is restored.

Because a write-through strategy was chosen, few changes need to be made to the speculative write function; the only change made to the write routines is that

whenever a compute node sends a write request, it checks to see if that address is cached, and if so, it updates the value in its cache.

To understand the performance implications of the software data cache, it helps to understand the operations involved. The cache tag check consists of the following steps:

1. Determine the segment in which the address is located.
2. Determine the line number to which the address is mapped.
3. Calculate the address's tag.
4. Compare the address's tag to the tag stored with the cache line.
5. If the tags are identical, return the appropriate data from the cache line.
6. If the tags differ, call the appropriate miss handler.

The cache miss handler performs the following tasks:

1. Determine the memory node responsible for the address being loaded.
2. Send a load request message to that memory node.
3. While waiting for the load reply, update the cache data structure to contain the tag and status bits for the newly cached data.
4. When the load reply arrives, store the data words into the cache line.
5. Return the appropriate data from the cache.

Cache misses will be slower than uncached loads for a few reasons. The first reason is that a tag check must be performed before the miss handler is called. The second reason is that loaded data must be placed in the cache in addition to being returned to the application. The third reason, which applies only to multi-word cache lines, is that the larger cache lines will require additional cycles to be read from the network. The fourth reason is that the addition of support for multi-word cache lines

has slightly increased the amount of time that it takes for memory nodes to respond to load requests. Because misses will be slower than uncached reads, the caching runtime system will perform worse than the original runtime system whenever the cache hit rate drops below a certain level.

In the absence of resource contention, the software data cache described in this thesis has a hit cost of 13 cycles and a miss cost of approximately 87 cycles (for an 8-word cache line). An uncached load takes 53 cycles, so the cached case can be expected to break even with the uncached case for a hit rate of roughly 46%.

Chapter 4

Results

In this section, I use four different application programs to compare the performance of the SUDS runtime with caching modifications to the performance of the original SUDS runtime and to the performance of optimized sequential versions of the applications. All SUDS runtime code was written in C and compiled with “gcc -O3 -funroll-loops.” (Loop unrolling was enabled to optimize the manipulation of multi-word cache lines.) To evaluate the scaling properties of the caching system, each application was run on three different Raw configuration sizes, a 4 compute node configuration, an 8 compute node configuration, and a 16 compute node configuration. For each configuration, the number of memory nodes was set at twice the number of compute nodes. This ratio of memory nodes to compute nodes was chosen to reduce contention for memory nodes to a low level. Configurations with smaller ratios of memory nodes to compute nodes would exhibit increased contention for memory node resources, which would bias the results in favor of the caching version of the runtime system for applications with good cache hit rates.

The applications were run on the Raw simulator, which can simulate arbitrary Raw configurations. The version of the simulator used to obtain the results in this section is faithful to the specification of the Raw chip in almost all respects. There are, however, three small differences which may impact the performance of SUDS. The first difference is that, in the simulator, the load latency is two cycles, while the hardware load latency is expected to be three cycles. The second is that large Raw

configurations were simulated as a single Raw chip containing the appropriate number of tiles with identical bandwidth and latency for all network links. In hardware, large Raw configurations will be built from multiple 16-tile Raw chips, and the network bandwidth from chip to chip will be lower than the bandwidth from tile to tile on a single chip. The third difference is that the simulator currently assumes that each tile has an infinite amount of local memory, while the Raw chip will use a hardware data cache on each tile to cache data from off-chip DRAM. Because cache misses will be communicated over the chip’s memory network, they will be subject to contention for network and memory controller resources. As a result, the exact cache miss time is difficult to predict. I modified the simulator to approximate such a cache by tracking the contents of a simulated cache and by assuming a constant miss time of 30 cycles, which I believe is an optimistic estimate of what the true average cache miss time will be. I do not believe that these minor discrepancies between the simulator environment and the hardware specification have any major impact on the relevance of the results presented in this chapter.

The existence of both Raw’s hardware data cache and the SUDS software data cache may cause some confusion. For the purposes of this thesis, the hardware cache’s job is to cache each tile’s data memory to a separate region of off-chip DRAM. Compute nodes and memory nodes are implemented on Raw tiles, so any data memory access on any node can potentially cause a hardware cache miss. This effectively increases the average cost of all memory operations. The SUDS software data cache is used to optimize the performance of speculative loads, which formerly required remote accesses for every memory operation. For the remainder of this chapter, any references to a “cache” or to “caching” refer to the SUDS runtime system’s software data cache unless otherwise noted.

Figure 4-1 provides a high-level view of the effects of software data caching on SUDS. For each application, speedups are shown for three different configurations of the SUDS runtime: without caching, with intra-chunk caching only, and with intra-chunk and permanent caching.

Jacobi is a dense-matrix implementation of the Jacobian relaxation algorithm. All

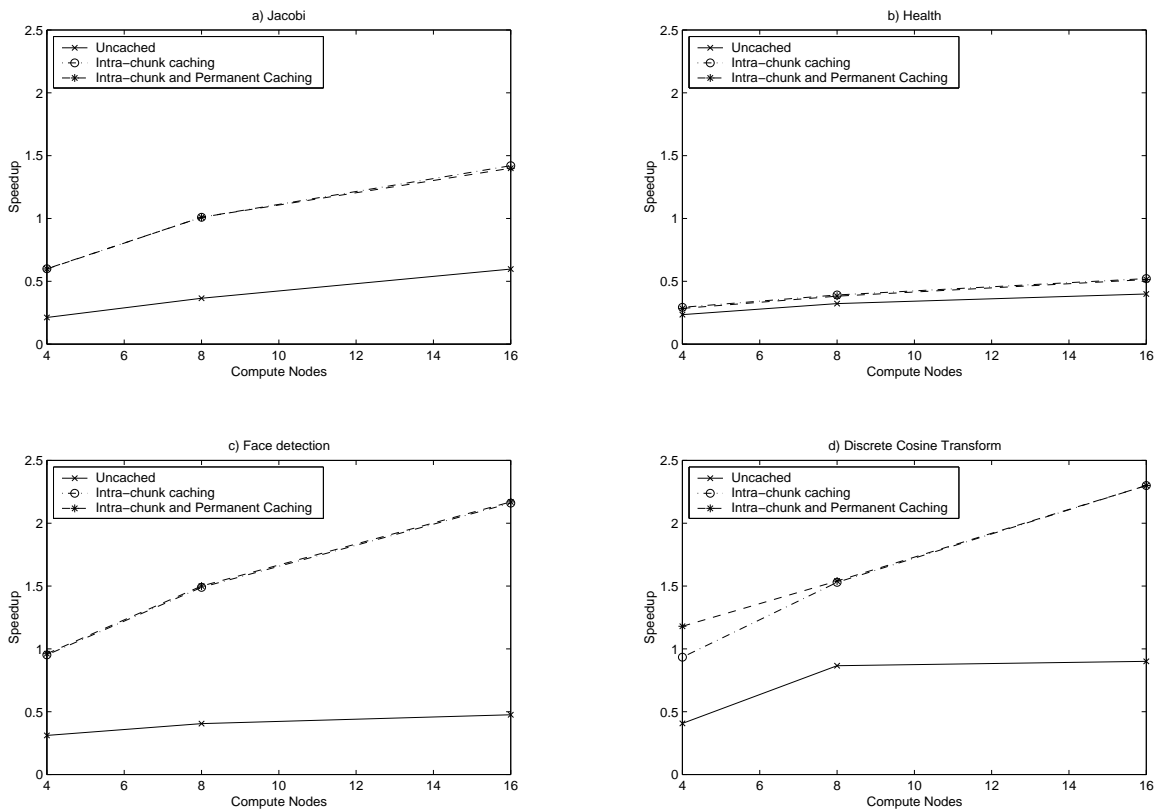


Figure 4-1: Speedups for the parallelized portion of each application program. Results for 4-, 8-, and 16-compute-node configurations are shown; in each case, the number of memory nodes is chosen to be twice the number of compute nodes. “Uncached” denotes the original SUDS runtime, before the implementation of caching. “Intra-chunk caching” denotes the runtime system with the capability to cache speculative data within a chunk. “Intra-chunk and permanent caching” denotes the runtime system with the additional ability to cache data permanently, thus allowing the compute node to retain cached data from one chunk to the next.

of its speculative accesses go to a pair of matrices. Health is an application from the Olden benchmark suite [2]. Most of its speculative accesses go to structs in a tree of linked lists; all structs in health have been padded to end on a cache line boundary to ameliorate problems with false sharing. Face detection is an implementation of a template-based face detection algorithm used by Cog, a humanoid robot [9]. Most of its speculative accesses go to either an input image or a set of data structures that describe the template. Discrete cosine transform (DCT), from the University of Toronto DSP benchmark suite [6], performs a blocked discrete cosine transform on an input image. Most speculative loads go to either the input image or an array of precomputed transform coefficients.

As mentioned in the previous chapter, the SUDS address space has been divided into two address ranges: one address range that will be cached with a line size of one word, and one range that will be cached at a coarser granularity. For all of the results presented in this chapter, each compute node has a 32 line cache for each address range, and the coarse-grained address range is cached using 8 word cache lines. Each cache contains only 32 lines because the cost of invalidating cache entries between chunks is a significant fraction of the total cost of inter-chunk bookkeeping operations. With only 32 lines in each cache, it is possible to overlap cache invalidation almost entirely with the memory node commit operation, but for larger cache sizes, this becomes less feasible. For the four applications studied in this chapter, each data structure is placed by hand into one of the two address ranges. Data that is read-only or written at a coarse granularity is placed in the multi-word-cached address range, and all other data is placed in the single-word-cached address range. Placing data that is shared at a fine granularity into the multi-word-cached address range would introduce false sharing into the system which would have detrimental effects on the performance of the application, but it would not affect correctness.

The results shown for the permanent caching case were obtained by hand-annotating certain loads as permanently-cached loads. For the results discussed in this chapter, loads that always access data from read-only global scalars or small, read-only data tables were marked as permanently cached loads. For Jacobi and health, there were

Application	Cache hit rate
Jacobi	90.5%
Health	54.8%
Face detection - intra-chunk only	94.7%
Face detection w/ permanent caching	95.0%
DCT - intra-chunk only	99.1%
DCT w/ permanent caching	99.2%

Table 4.1: Cache hit rates for the parallelized loops in each application for a four-compute-node configuration.

no suitable data structures, so no loads were permanently cached. For face detection, loads of a few global scalars and loads from a small data structure that encodes the face template were permanently cached. For DCT, loads from the precomputed array of transform coefficients were permanently cached.

Figure 4-1 shows that caching improves the performance of SUDS on all four of the test applications for all hardware configurations. Additionally, it shows that the addition of caching does not generally worsen the scaling properties of the system as the hardware configuration is changed. The remainder of this chapter will examine these result in more detail and introduce a few auxiliary results in an attempt to more precisely characterize the effects of caching on the runtime system.

For three of the four applications, caching improves performance by more than a factor of two. The remaining application, health, only improves by 20 to 30 percent with the addition of the caching mechanism. In addition, it exhibits the poorest parallel speedups of any of the four applications. The latter of these two observations is true because the number of cycles of computation done by each chunk in health is roughly equivalent to the number of cycles spent in communication and bookkeeping operations that SUDS performs between chunks. This is also one reason for the smaller gain obtained from caching, since performance improvements from caching do not apply to inter-chunk communications between nodes. Another reason for health's lackluster performance with caching is that, as shown in Table 4.1, its cache hit rate was far below the hit rate of any of the other applications.

Application	Latency without caching	Latency with caching
Jacobi	69	57
Health	85	75

Table 4.2: Latency measured from the sending of the load request to the receipt of the load reply for a four-compute-node configuration.

Operation	Cost (cycles)
8-word line commit	13
single-word commit	15
8-word line rollback	176
single-word rollback	27

Table 4.3: Inter-chunk memory node operation costs for Jacobi. Costs are per-line for the 8-word cache line and per-word for the single-word case.

Based on the hit rate shown in Table 4.1 and the ideal cache hit and miss times described in the previous chapter, health’s improvement from caching is larger than one might initially expect. This additional performance improvement is due to reduced contention for network and memory node resources. More network traffic means that message travel times will be greater, and more requests to the memory nodes mean an increased likelihood of a request arriving while the memory node is busy processing an earlier request.

I instrumented versions of Jacobi and health to determine the latency between the sending of the load request and the receipt of the reply. As Table 4.2 shows, the average round-trip time in all cases is much longer than the optimistic assumption made in the previous chapter, and it is significantly worse for the noncaching case than for the caching case. Thus we see that, at least for these two applications, the memory bandwidth reduction gained from caching improves the average load latency.

Another benefit of the caching implementation of the SUDS runtime system is that, for applications with good spatial locality, it reduces the amount of time taken by the inter-chunk bookkeeping operations because the logged information is kept per cache line instead of per word. The average commit and rollback costs of the

memory nodes are shown in Table 4.3. Rollback of a cache line takes longer than rollback of a single word, but if more than six of the words in the cache line were actually used during the chunk, the average rollback cost per word will be less for the cache line than for the individual words. Because all that needs to be done is to set a flag for each entry, the commit operation takes approximately the same number of cycles whether it's a commit of a cache line entry or a (non-cached) single word entry. If any spatial locality can be exploited by the cache, the commit operation will be faster for the caching runtime system than the noncaching runtime system. Thus, although rollbacks may be slower when the dependence information is kept at a coarser granularity, the common-case commit operation will be faster.

The results shown for the runtime system with permanent caching are in most cases indistinguishable from the results for the runtime system with only intra-chunk caching. Jacobi and health did not contain any opportunities for permanent caching, so they did not demonstrate any performance improvement when permanent caching was added. Health's performance degraded slightly because of the additional overhead of preserving permanently cached items in each compute node's cache from one chunk to the next. When permanent caching is disabled, inter-chunk cache cleaning can be done by simply invalidating each cache line. When permanent caching is enabled, however, each cache line must be checked to see if it is permanently cached, and only lines that are not permanently cached should be invalidated. This extra check approximately doubles the number of cycles that it takes to maintain the cache between chunks. This effect shows up most obviously in health because it performs such a small amount of computation in each chunk. For the other applications, the number of cycles spent on inter-chunk bookkeeping is much less significant. Face detection and DCT have small read-only data structures that are frequently accessed, but in each application, those data structures are accessed numerous times in each chunk, so a single miss per cache line at the beginning of each chunk is not significant in most cases. Face detection and DCT both demonstrated very small additional speed-ups with permanent caching enabled, and both show small improvements to cache hit rate. The only significant speed-up was demonstrated by DCT with four

compute nodes. DCT benefited from the caching of its transform coefficients from one chunk to the next. DCT would probably have demonstrated additional speed-up with the 8- and 16-tile configurations, but the problem size was so small that these configurations only required 2 or 1 chunks, respectively, to complete the computation. With so few chunks, the benefits of permanent caching cannot manifest themselves.

Chapter 5

Discussion

In this chapter, I present a high-level analysis of the performance of the caching runtime system in light of the results presented in the previous chapter.

The SUDS runtime system’s compute-node cache will, in general, have a lower hit rate than a data cache for a similar sequential program for a number of reasons. One reason is that many of the memory accesses that would be likely to hit in the cache, such as local variable accesses, have already been privatized by the compile-time program transformations. Another reason for the lower hit rate is that caches are kept consistent with the memory nodes by invalidating their contents (except for permanently cached data) after every chunk. The previous chapter demonstrated, however, that a compute-node data cache improves performance even for applications, such as health, that do not have high hit rates.

Privatizing variables at compile-time is usually preferable to using the software cache because privatized variables can be accessed without incurring the tag-check overhead and because privatizing them prevents them from utilizing memory node resources. In the case of small, read-only data structures, it would be possible in some cases, through appropriate use of pointer analysis information [8], to determine that the array can be privatized by each compute node, but this will not be a good solution if the read-only data is only sparsely accessed. This is, in fact, the case with the face detection application examined in the previous chapter. A read-only data structure of a few hundred words encodes the “face template” for the application,

and the program compares the template to each possible face location in an input image. The template comparison will terminate early if the image location differs greatly from the template, and, in practice, only a small fraction of the template data is accessed when testing most image locations. If the compile-time component of SUDS were to send a private copy of the template to each compute node at the beginning of each chunk, most of that data would go unused most of the time, and performance might suffer as a result.

The current implementation uses a direct-mapped cache because the direct-mapped tag check can be efficiently implemented in software. Implementations with higher associativity might improve hit rates in some cases, but they would likely increase the hit time because of the more complicated tag check that would be required. None of the applications examined in the previous chapter have an access pattern that brings out the worst in the direct-mapped cache, so the results from the previous chapter are best-case results in that respect.

The frequent cache invalidations required by SUDS, in addition to lowering the overall hit rate, also make it the case that the number of cycles required to walk through the cache and invalidate appropriate lines can add to the critical path length of the parallelized program. This effectively limits the size of the cache, thus potentially decreasing the hit rate even further. For small caches in the absence of permanent caching, it is possible to overlap the cache invalidation with the commit process of the memory nodes.

The overhead of cache invalidation is, in practice, only on the critical path for the master compute node, so one possible optimization is to use a small cache on the master compute node and to use larger caches on the slave compute nodes.

A higher-level problem with the caching system is that it is not clear how effectively compile-time analysis would be able to partition data objects into one of the two address ranges defined by the caching system. The performance implications of placing a data object into the “non-optimal” address range can be extreme. Introducing false sharing by placing something inappropriate into the multi-word cache line address range can greatly increase the number of rollbacks that occur, thus eliminating

any hope of exploiting application parallelism. Placing something inappropriate into the single-word cache line address range will eliminate the possibility of exploiting spatial locality, which would eliminate almost all of the benefits of caching for programs, such as Jacobi, that benefit mostly from spatial locality of memory accesses. Compile-time pointer analysis should be able to determine many cases where data objects are read-only for the duration of a parallelized loop, and it should be able to determine the stride length for many regular array accesses [12]. Information like this could be used to select an address range for some data objects, but it seems unlikely that such analysis will apply to all data objects in the irregular integer applications that SUDS was designed to handle.

A similar problem applies to the choice of whether to cache data permanently, and it is not obvious that compile-time analysis would be able to determine which loads to permanently cache. Based on the results from the previous section, however, it seems that permanent caching is useful in only a limited set of circumstances. It may be reasonable to assume that loads should not be permanently cached unless it can be proven that they access a data object that is read-only for the duration of the speculative loop.

Permanently caching data that is written frequently will obviously degrade performance, but permanently caching certain types of read-only data can also degrade performance for another, more subtle, reason. Every permanently cached piece of data permanently occupies a hash table entry on one of the memory nodes. If many distinct elements of a large data structure are accessed, this will gradually increase the load factor of the hash table. Eventually, this may lead to a collision in the hash table, which will trigger a rollback. On rollback, the hash table will be completely emptied, including entries corresponding to permanently cached data, and the process will start again. The cost of rollbacks due to these additional collisions may outweigh any benefits from increased hit rates that come from permanently caching the data. It may be possible to avoid this problem by having a separate hash table for permanently cached data; when a collision occurs in the “permanent” hash table, the memory node could record the access in the “normal” hash table, and send a reply to

the compute node indicating that the data is only safe for intra-chunk caching instead of permanent caching.

Due to the perceived inadequacy of current compiler analysis to completely handle the problem of deciding what type of caching is appropriate for a given data object, it seems desirable to add support to the runtime system for dynamically determining the type of caching to use for each static load in the program. Implementing such features in the runtime system is challenging because it requires the efficient gathering of information about the speculative loads performed by the program and the ability to use this information without adversely affecting the speed of the hit case in the cache. Preliminary work is underway to add support for dynamically determining whether it is suitable to permanently cache the data from a given speculative load in the program. It appears that this can be done without in any way affecting the hit case in the cache, since permanently caching data does not affect its location in the cache.

Because the numerical value of the address itself currently determines the granularity at which it is cached, it is not clear how to dynamically change granularity size. It may be necessary to make more profound changes to the mechanism by which caching granularity is determined before the runtime system will be able to dynamically adjust the caching granularity in a useful way. Support for dynamically changing the cache line size would make it somewhat more complicated for the memory nodes to maintain access ordering information on the speculative memory, but because this is not on the critical path of the program, this does not seem to be a major obstacle to the implementation of such a system. Dynamically changing the cache line size would also increase the time required to do a tag check, but in the worst case, it would require two tag checks, which may be an acceptable price to pay for a more robust method of determining cache line size. One way to hide some of the increased costs that a more proactive runtime system would incur is to integrate some of the techniques used by the Hot Pages system [7] to optimize tag checks.

I have spent most of this chapter dwelling on the shortcomings of the caching system, but, as was demonstrated by the previous chapter, the cache provides a

major improvement to the performance of the overall system on the test applications. It does this by avoiding the high cost of issuing speculative load requests to the memory nodes and waiting for the data in the reply. Cache hits are much faster than uncached loads, and cache misses are not much slower than uncached loads under normal operating conditions.

The results from the previous chapter demonstrate that using two different cache line sizes allows SUDS to exploit spatial locality where possible while avoiding false sharing of data that is written at a fine granularity. Another effect of two cache line sizes is that much of the dependence information maintained by the memory nodes is now maintained at a coarser granularity than it was in the original runtime system. For applications that exhibit significant spatial locality, this reduces the cost of maintaining dependence information, and for applications with poor spatial locality, there is very little additional overhead in the common case.

Chapter 6

Conclusion

As computing systems increase in size and complexity, new ways of exploiting application parallelism, such as SUDS, will become increasingly important. However, such systems cannot be expected to perform well if many of their memory accesses incur large remote access overheads and overheads from the operations necessary to maintain speculative state.

This thesis demonstrates that adding software caching to the SUDS runtime can improve performance over a system without caching and can enable the system to achieve parallel speedups on some applications. The speculative, chunk-based nature of the system places somewhat unusual demands on the software cache, but it also provides a few simple mechanisms that can be used to ensure correct program execution without requiring additional network communication to maintain cache coherence.

The system described in this thesis is a first step toward an efficient implementation of the SUDS runtime, but it leaves a few questions unanswered. Specifically, it provides support for coarse-grained caching and for permanent caching of read-only and read-mostly data, but it places the burden of choosing when to use these mechanisms on the programmer, which is not an acceptable long-term solution.

Future work should include a way of effectively choosing when to use the various caching mechanisms without requiring so much help from the programmer, but this thesis has demonstrated that a system that effectively employs such mechanisms can perform much better than a system operating completely without the caching

speculative data.

Bibliography

- [1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, New York, June 1988. IEEE.
- [2] Martin C. Carlisle and Anne Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.
- [3] Matthew Frank, C. Andras Moritz, Benjamin Greenwald, Saman Amarasinghe, and Anant Agarwal. SUDS: Primitive Mechanisms for Memory Dependence Speculation. Technical report, M.I.T., January 6 1999.
- [4] Manoj Franklin and Gurindar Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *1996 IEEE Transactions of Computer*, pages 552–571, May 1996.
- [5] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.
- [6] Corinna Lee and Mark Stoodley. UTDSP BenchMark Suite. <http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html>, 1992.
- [7] Csaba Andras Moritz, Matt Frank, Walter Lee, and Saman Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. (LCS-TM-599), Sept 1999.

- [8] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.
- [9] Brian M. Scassellati. Eye Finding via Face Detection for a Foveated Active Vision System. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, July 1998.
- [10] Michael B. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1999.
- [11] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.
- [12] Robert P. Wilson. Efficient Context-Sensitive Pointer Analysis For C Programs. In *Ph.D Thesis, Stanford University, Computer Systems Laboratory*, December 1997.