HARDWARE AND SOFTWARE MECHANISMS FOR REDUCING LOAD LATENCY

By

Todd Michael Austin

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

As processor demands quickly outpace memory, the performance of load instructions becomes an increasingly critical component to good system performance. This thesis contributes four novel load latency reduction techniques, each targeting a different component of load latency: address calculation, data cache access, address translation, and data cache misses. The contributed techniques are as follows:

- *Fast Address Calculation* employs a stateless set index predictor to allow address calculation to overlap with data cache access. The design eliminates the latency of address calculation for many loads.
- *Zero-Cycle Loads* combine fast address calculation with an early-issue mechanism to produce pipeline designs capable of hiding the latency of many loads that hit in the data cache.
- *High-Bandwidth Address Translation* develops address translation mechanisms with better latency and area characteristics than a multi-ported TLB. The new designs provide multiple-issue processors with effective alternatives for keeping address translation off the critical path of data cache access.
- *Cache-conscious Data Placement* is a profile-guided data placement optimization for reducing the frequency of data cache misses. The approach employs heuristic algorithms to find variable placement solutions that decrease inter-variable conflict, and increase cache line utilization and block prefetch.

Detailed design descriptions and experimental evaluations are provided for each approach, confirming the designs as cost-effective and practical solutions for reducing load latency.

Acknowledgements

It's hard to believe, but after nearly six years, the time has come to leave Madison. When I think back to all my years here, it is clear there are many to whom I owe my thanks and acknowledgements.

First and foremost, I must thank my family for their endless support in my mostly self-indulgent endeavor to pursue a Ph.D. My wife Emily, son Nicholas, and daughter Katharine have spent far too many years living from check to check in too small of a house with too small of a car, always giving me the love, stability, and diversions I needed to finish my Ph.D. with my sanity intact.

I have made many close friends in the Computer Sciences Department, more than I can include here. Their influences and friendships will be with me all my life.

My advisor, Guri Sohi, has provided me with the guidance and encouragement needed to navigate from CS 552 to this thesis. When I compare myself to the person I was six years ago, it becomes truly evident how much Guri has influenced my attitudes and abilities. I hope he will feel free to share in any successes that I may have after I leave Madison.

The members of my preliminary and final defense committees, Suresh Chalasani, Charles Fischer, Jim Goodman, Jim Larus, and David Wood, provided invaluable assistance by critiquing this work at all stages of its development. Their efforts have made this thesis more accurate, more complete, and easier to read.

The members of the Wisconsin Multiscalar group have been my close confidants since the day I arrived. Scott Breach has been my most trusted sounding board for ideas and a close collaborator on the Safe-C work. T.N. Vijaykumar was a collaborator on the Knapsack work, the primordial seed of this thesis. Dionisios Pnevmatikatos shared his talents with me on the Fast Address Calculation work and served as a willing guinea pig for my SimpleScalar experiments.

Other people that I want to acknowledge for their friendship and company are Steve Bennett, Doug Burger, Babak Falsafi, Alain Kägi, Alvy Lebeck, Andreas Moshovos, Subbarao Palacharla, and Steve Reinhardt.

Finally, I would like to express my thanks to the agencies that helped to fund my endeavors at Madison. My work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and a donation from Intel Corporation.

Todd M. Austin
University of Wisconsin – Madison, April 1996

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory system design is one of the most challenging aspects of computer architecture. The dynamic nature of the computer industry limits the lifetime of any cost-effective design solution, forcing designers to continually re-evaluate the cost-effectiveness of any design in light of changes in implementation technology, workload, and processor architecture. To further challenge designers, the most cost-effective solutions will often span the traditional boundary placed between hardware and software designs, possessing vital components in both domains.

One of the paramount challenges in memory system design is the problem of continually lengthening load latency. For a large class of important programs, growing load latency dominates all other latencies during execution, making the performance of load instructions the most important barrier to good system performance.

The problem of lengthening load latency can be traced to continuing technology, workload, and architecture trends. Each year, advances in implementation technologies reward processors with faster clocks, while memory speeds, particularly for DRAM, remain relatively unchanged. The fallout is a widening gap between the speed of processors and memory. Cache memory has for a long time been an effective mitigator, providing reduced memory access time through the use of the memory hierarchy. However, many newer workloads lack the locality necessary to perform well on traditional memory hierarchies. Familiar examples of this trend can be seen in environments supporting multitasking, multimedia, compression, and encryption. Finally, architectural changes place increased demands on the memory system which complicates its design and further aggravates load latency. A timely example of this trend can be seen in the shift towards instruction-level parallel processors that issue multiple loads and stores per cycle, resulting in increased bandwidth demands on the memory system.

This thesis contributes four novel techniques for reducing load latency. The work presented takes a balanced approach to addressing the problem. A full chapter of material is devoted to each of the four major components of load latency: *address calculation*, *data cache access*, *address translation*, and *data cache misses*. Detailed design descriptions are given for each approach. The solutions presented include a hardware-based and a software-based design as well as two hardware-software codesigns. The effectiveness of each approach is evaluated using qualitative analyses to assess impacts on pipeline complexity and cycle-time as well as cycle-timing simulations to gauge impacts on pipeline throughput.

The remainder of this chapter presents the background and motivation necessary to prepare the reader for the remaining chapters. Section 1.1 introduces load instructions and describes the components of load latency. Section 1.2 explores the impact of load latency on both the performance of a single load instruction as well as the performance of the entire system. Section 1.3 examines existing approaches for reducing the impact of load latency. Section 1.4 presents an overview of the contributions of this thesis, and finally, Section 1.5 details the organization of the remaining chapters.

1.1 Anatomy of a Load

Load instructions move data between the processor and memory. The semantics of a typical load are as follows: two inputs, a base address and offset, are added together to form an effective address which is used to access data memory. In most modern architectures, the base is supplied from a processor register and the offset is supplied by either an immediate constant, *i.e.*, `register+constant` addressing, or via a register, *i.e.*, `register+register` addressing. Architectures often employ variations of this simple scheme; for instance, the IBM System 370 combines the two addressing forms, offering an addressing mode that supports `register+register+constant` addressing. The VAX and Intel (x86) architectures eliminate explicit load instructions altogether by permitting any instruction to access operands from either registers or memory. In this thesis, the discussion is restricted to architectures with explicit load instructions and `register+constant` and `register+register` mode addressing. However, the approaches presented within should easily extend to other machines as well.

A load, while being a single instruction, is decomposed into several component operations when mapped onto a processor pipeline. Figure 1.1 illustrates the major component operations of a load and their order of execution.[1] A load is first fetched by the processor from instruction memory, possibly within a group of instructions. Next, the processor must identify the load instruction. After a load has been identified, it is aligned to the pipe or functional units supporting its execution, base and index register values are read from the register file, and functional unit resources are secured. Finally, the effective address is computed and used to access data memory. As shown in the figure, address translation proceeds in parallel with data memory access.

In a typical pipeline implementation, loads are fetched in the fetch (IF) stage of the pipeline. Identifying, aligning, and reading the register file occur in the decode (ID) stage of the pipeline. In designs with very fast clocks and wide issue, these operations are often split across multiple decode stages. Effective address computation occurs in the execute (EX) stage of the pipeline, and data memory access and address translation in the memory access (MEM) stage of the pipeline.

Load latency is defined as the time it takes to compute the effective address of the access, access data memory, and return a result. In the example in Figure 1.1, effective address calculation takes

[1]Of course, many variations exist upon this basic template; for example, some pipelines require address translation to complete before accessing the data cache. However, the basic template shown is representative of many modern pipelines.

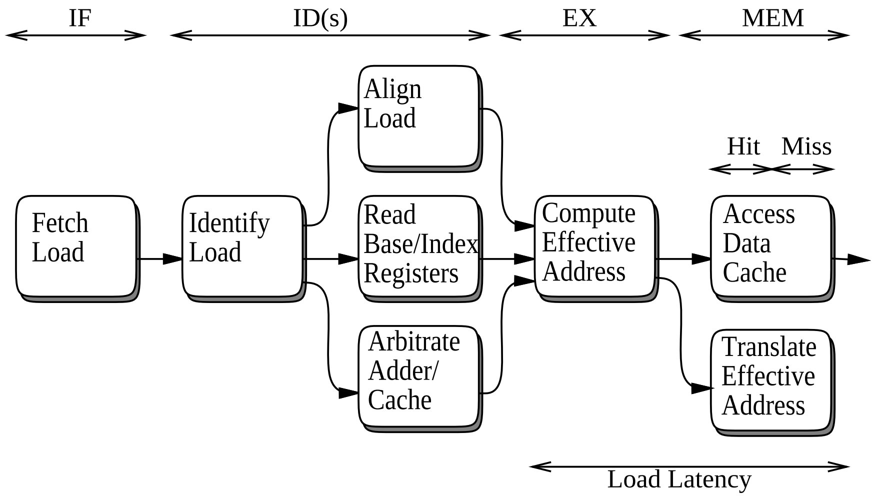


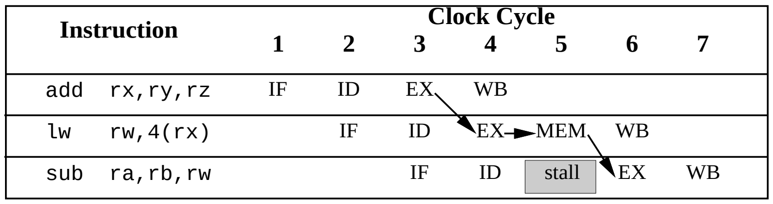
Figure 1.1: Anatomy of a Load.

a single cycle, and data memory access takes a single cycle if the access hits in the data cache. If a load misses in the data cache, its latency is further increased by delays incurred with accessing lower levels of the data memory hierarchy, *e.g.*, cache misses or page faults. The remaining components, *e.g.*, fetch, align, identify, and arbitrate, are not normally considered an integral part of load latency, since they share few dependencies with earlier instructions and thus can be effectively hidden by overlapping their execution with earlier instructions.

1.2 The Impact of Load Latency

Figure 1.2 illustrates how load latency affects program execution. The figure shows a traditional five stage pipeline executing three dependent instructions. Pipelined execution continues without interruption until the `sub` instruction attempts to use the result of the previous `lw` instruction. In a traditional five stage pipeline, a load instruction requires the EX stage for effective address calculation and the MEM stage for data cache access. The result of the load operation is not available until the end of cycle 5 (assuming a single cycle cache access and the access hits in the data cache), forcing the pipeline to stall issue one cycle waiting for register `rw`. As the pipeline stalls, valuable functional unit resources sit idle, wasting resources that could otherwise be used to improve program performance.

Fortunately, the effect of load latency on program performance is tempered by two factors: 1) the processor's ability to tolerate latency, and 2) the relative impact of load latency compared to other latencies. The workload and execution model both affect the degree to which the processor can tolerate latency. If the workload contains sufficient parallelism and the execution model provides the capability to exploit the parallelism, the impact of load latency on overall performance can be reduced by executing independent instructions on idle processor resources. Execution models with high levels



Instruction	1	2	3	4	5	6	7
`add rx,ry,rz`	IF	ID	EX	WB			
`lw rw,4(rx)`		IF	ID	EX	MEM	WB	
`sub ra,rb,rw`			IF	ID	stall	EX	WB

Figure 1.2: Impact of Load Latency.

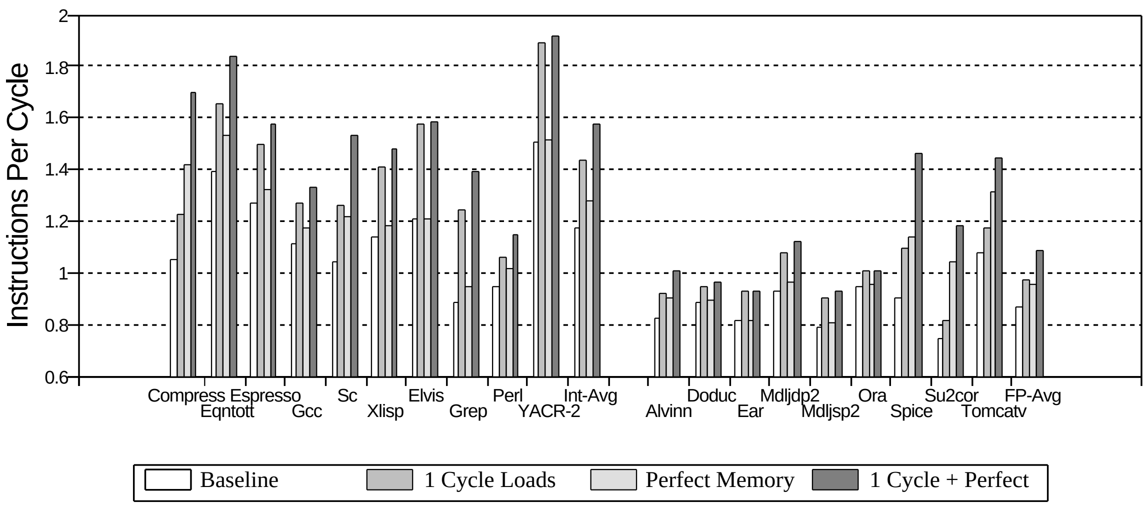


Figure 1.3: Impact of Load Latency on IPC.

of latency tolerating capability include those that support such techniques as out-of-order issue [HP90], non-blocking loads [FJ94], and speculative execution [HP90]. The extent to which a program accesses memory (*i.e.*, the dynamic frequency of loads) also affects the degree to which load latency impacts program performance. Programs that access memory often will need better load performance for good overall performance.

To gauge the extent to which load latency affects system performance, a simulation case study was performed on the SPEC '92 benchmarks. The IPCs of benchmarks compiled with GNU GCC were examined while executing with varied load latency. Experiments were run on an aggressive 4-way in-order issue processor timing simulator (described in Section 3.6). The simulator was configured with a 16k byte dual-ported direct-mapped non-blocking data cache, with 32 byte blocks, and a fixed 6 cycle miss latency. The results of the experiments are shown in Figure 1.3. *Baseline* shows program performance with 2 cycle loads and a 6 cycle cache miss latency, *1-Cycle Loads* reduces the cache hit latency to 1 cycle but retains the 6 cycle cache miss penalty, *Perfect Cache* represents a 2 cycle load latency and a 0 cycle cache miss penalty, and *1 Cycle+Perfect* represents the case where all load instructions complete in 1 cycle. In addition, the graph shows the average IPC, weighted by program run-time (in cycles), for the integer codes (the left group) and the floating point codes.

A number of interesting observations can be made from Figure 1.3. In general, the integer codes saw more improvement in performance when load latency was reduced. The floating point codes did not fair as well because their executions are dominated by other long latency operations such as floating point computations. When comparing the relative impact of the analyzed component latencies, *i.e.*, address calculation, data cache access, and cache misses, no single component stands out as the dominating component across all the codes. The integer code performance was affected more by eliminating the latency of address calculation or data cache access (*i.e.*, 1-Cycle Loads), while the floating point codes benefited more by eliminating the latency of data cache misses (*i.e.*, Perfect Cache). Clearly, load latency can have a significant impact on system performance. Optimizations designed to reduce the impact of load latency, independent of which component of load latency is targeted, stand to gain much.

1.3 Reducing the Impact of Load Latency

A significant body of work proposes techniques for reducing the impact of load latency. The techniques can be broadly bisected into two camps: latency tolerating techniques and latency reducing techniques.

Tolerating techniques work by inserting independent instructions into the pipeline delay slots created by load latencies. Local instruction scheduling [KE93, PS90, GM86] is a commonly used compile-time technique to tolerate load latency (it is employed in this work as well). The scheduler attempts to place independent instructions between loads and their first use, keeping pipeline resources utilized until loads complete. To make good schedules, the scheduler needs independent work, which is finite and usually quite small in the basic blocks of control intensive codes, *e.g.*, many integer codes [AS92]. Global scheduling techniques [Fis81, MLC$^+$92, ME92] have been developed as a way to mitigate this effect; however, these techniques often suffer from ambiguous dependencies, unpredictable latencies, and safety issues that limit the extent of their effectiveness.

Many of the limitations of compile-time scheduling can be overcome by using a dynamic scheduling approach. When processor progress is stalled due to a load delay (or other instruction delays), the dynamic scheduler selects another instruction from a window of available instructions or another independent thread of control. Examples of execution models that perform dynamic scheduling include Multiscalar [Fra93], decoupled [Smi82b], dataflow [Vee86], and multi-threaded [LGN92, Smi81].

The best way by far to reduce load latency is to make memory access time zero by moving the accessed storage into a register. Register allocation is a well developed area and continues to progress. Current research centers on increasing the number of candidates for register allocation, *e.g.*, register allocation for subscripted variables [CCK90], and increasing the utilization of a finite collection of registers through techniques such as live-range splitting [BCT92] and load/store range analysis [KH92b]. Unfortunately, many program variables are still forced into memory, due to the limited size and addressability of register files.

Cache misses are often a significant component of load latency, especially in numeric codes where data locality is low. Much work has been done to reduce both the latency and frequency of data cache misses. Approaches that work to reduce miss latencies include multi-level caches [JW94, BKW90, WBL89], victim caches [Jou90], and cache line prefetching [RL92, CBM+92, MLG92]. Non-blocking caches [FJ94, CB92, Con92, SF91] also help to reduce the impact of cache misses by letting other cache accesses complete while misses are serviced.

Techniques that work to reduce the frequency of cache misses usually attack the problem of reducing conflict misses. Approaches along these lines include set-associative caches [KJLH89, Hea86, Smi82a], column-associative caches [AP93, AHH88], stride tolerant address mappings [Sez93, IL89, CL89], static [Kes91, DS91] or dynamic [BLRC94, LBF92, Kes91] page coloring, program restructuring [LRW91, Wu92, PH90, Fer76], and reference exclusion [McF92, CD89, ASW+93, Hsu94, Con92].

Even with the bevy of work already available to reduce the impacts of load latency, the problem continues to persist, partly because load latency continues to grow and partly because existing approaches have limited applicability or effectiveness.

1.4 Contributions of This Thesis

This thesis contributes four novel load latency reduction techniques, each targeting a different component of load latency: *address calculation*, *data cache access*, *address translation*, and *data cache misses*. Figure 1.4 shows the four contributions of this thesis and the load latency components that each addresses. The following subsections give a brief overview of each approach and key results of their evaluation.

1.4.1 Fast Address Calculation

For many programs, especially integer codes, exposed load instruction latencies that hit in the data cache account for a significant portion of total execution time. For these codes, *fast address calculation* is an effective method to reduce load latency.

The approach works to reduce load latency by allowing effective address calculation to proceed in parallel with data cache access, thereby eliminating the extra cycle required for address calculation. The technique employs a simple circuit to predict the portion of the effective address needed to read the data cache. If the address is predicted correctly, the cache access completes without an extra cycle for address calculation. If the address is mispredicted, the cache is accessed again using the correct

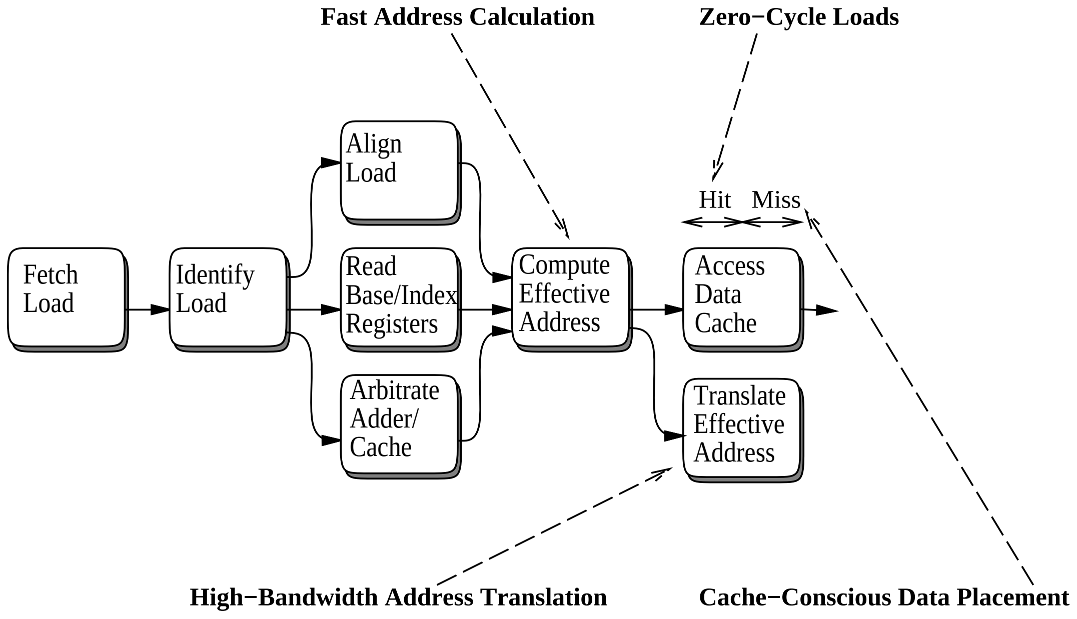

Fast Address Calculation
Zero−Cycle Loads
Align Load
Hit
Miss
Fetch Load
Identify Load
Read Base/Index Registers
Compute Effective Address
Access Data Cache
Arbitrate Adder/ Cache
Translate Effective Address
High−Bandwidth Address Translation
Cache−Conscious Data Placement

Figure 1.4: Thesis Road Map.

effective address. The predictor is designed to minimize its impact on cache access time, adding only a single OR operation before data cache access can commence. Verification of the predicted effective address is also very fast and decoupled from the cache access critical path, ensuring that pipeline control logic impacts are minimal.

Detailed timing simulations of a 4-way in-order issue superscalar processor extended to support fast address calculation show this design is a good one, servicing enough accesses early enough to result in speedups for all the programs tested. Simulations found an average speedup of 14% for the integer codes and 6% for the floating point codes. The approach also responds well to software support. Compiler and linker support for fast address calculation was shown to significantly reduce the number of mispredictions, in many cases provide better program speedups and reduced cache bandwidth demand. Simulated performance with software support improved the average speedup to 19% for the integer codes and 7.5% for the floating point codes.

1.4.2 Zero-Cycle Loads

Address calculation is typically one half of the latency for loads that hit in the cache. *Zero-cycle loads* extend the latency reduction afforded by fast address calculation by combining it with an early-issue mechanism. The resulting pipeline designs are capable of completely hiding the latency of many loads that hit in the cache.

Through the judicious application of instruction predecode, base register caching, and fast address calculation, it becomes possible to complete load instructions up to two cycles earlier than traditional pipeline designs. For a pipeline with one cycle data cache access, loads can complete before reaching the execute stage of the pipeline, creating what is termed a *zero-cycle load.* A zero-cycle load allows subsequent dependent instructions to issue unencumbered by load instruction hazards, resulting in fewer pipeline stalls and increased overall performance.

Two pipeline designs supporting zero-cycle loads are presented: an aggressive design for pipelines with a single stage of instruction decode, and a less aggressive design for pipelines with multiple decode stages. The designs are evaluated in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on architectures with many and few registers.

Programs running on a 4-way in-order issue processor simulator extended to support zero-cycle loads found an average speedup of 45% for the integer codes and 26% for the floating point codes. For the integer codes, program performance was on par with the speedups afforded by an out-of-order issue processor model. Speedups on an out-of-order issue processor simulator extended to support zero-cycle loads were less due to the latency tolerating capability of the execution model. On architectures with few registers, the frequency of loads and their impact of program performance increases significantly. Providing an 8 register architecture with limited zero-cycle load support resulted in performance comparable to a 32 register architecture, suggesting that the approach may be able to negate the impacts of too few architected registers.

1.4.3 High-Bandwidth Address Translation

Address translation is not generally thought of as a latency component of loads. The usual goal of address translation design is not to directly minimize load latency, but rather to keep address translation off the critical path of data cache access. Typically, this task is accomplished by allowing data cache access and address translation to proceed in parallel using, for example, a virtually-indexed cache. The address translation mechanism is then constructed to be at least as fast as data cache access.

However, this design strategy is becoming increasingly more difficult. In an effort to push the envelope of system performance, microprocessor designs are exploiting continually higher levels of instruction-level parallelism, resulting in increasing bandwidth demands on the address translation mechanism. Most current microprocessor designs meet this demand with a multi-ported TLB. While this design provides an excellent hit rate at each port, its access latency and area grow very quickly as the number of ports is increased. As bandwidth demands continue to increase, multi-ported designs may soon impact memory access latency.

To help meet these new demands, four new high-bandwidth address translation mechanisms are presented that feature latency and area characteristics that scale better than a multi-ported TLB design. Traditional high-bandwidth memory design techniques are extended to address translation, developing *interleaved* and *multi-level* TLB designs. In addition, two new designs crafted specifically for high-bandwidth address translation are introduced. *Piggyback ports* are proposed as a technique to exploit spatial locality in simultaneous translation requests, allowing accesses to the same virtual memory page to combine their requests at the TLB access port. *Pretranslation* is proposed as a technique for attaching translations to base register values, making it possible to reuse a single translation many times.

Extensive simulation-based studies were performed to evaluate the proposed designs. Key system parameters, such as execution model, page size, and number of architected registers were varied to see what effects they had on the relative merits of each approach. A number of designs show particular promise. Multi-level TLBs with as few as eight entries in the upper-level TLB nearly achieve the performance of a TLB with unlimited bandwidth. Piggyback ports combined with a lesser-ported TLB structure, *e.g.*, an interleaved or multi-ported TLB, also perform well. Pretranslation over a single-ported TLB performs almost as well as a same-sized multi-level TLB with the added benefit of decreased access latency for physically indexed caches.

1.4.4 Cache-Conscious Data Placement

For many codes, the data cache miss component of load latency dominates all other memory access latencies. These codes often have working sets that are too large or lack the locality necessary for good data cache performance. In other cases, their reference streams perform poorly on commonly used cache geometries. Whatever the case may be, these programs spend much of their execution time waiting for cache misses to be serviced and can benefit greatly from optimizations designed to reduce the impact of data cache miss latencies.

A software-based variable placement optimization, called *cache-conscious data placement*, is introduced as a technique for reducing the frequency of data cache misses. To apply the approach, a program is first profiled to characterize how its variables are used. The profile information then guides heuristic data placement algorithms in finding a variable placement solution that decreases predicted inter-variable conflict, and increases predicted cache line utilization and block prefetch. The generated placement solution is implemented partly at compile-time using a modified linker and partly at run-time with modified system libraries.

Various placement strategies are developed and compared to the performance of natural placement (*i.e.*, the layout of variables using the unmodified linker and system libraries). Random placement performed consistently worse than natural placement, revealing natural placement as an effective placement strategy that sets the bar for artificial placement measures. Cache-conscious data placement improved cache performance for most of the programs tested, with many seeing more than a 10% reduction in data cache misses. A simplified and less expensive version of the placement algorithm (*i.e.*, with reduced computation and storage requirements) lost the stability of the more complex algorithm, suggesting that for consistent performance improvements the more capable and expensive algorithm is required. Run-time performance impacts were also examined for in-order and out-of-order issue processor models. Run-time performance improvements were small, especially for the out-of-order issue processor, because the processors were able to tolerate much of the data cache miss latency eliminated. However, the placement optimizations were able to eliminate a large fraction of the exposed data cache miss latency in many of the experiments.

1.5 Organization of This Thesis

The remainder of thesis is organized as follows. Chapter 2 describes the experimental framework used throughout this thesis. Chapter 3 develops and analyzes fast address calculation. Chapter 4 examines and evaluates zero-cycle loads. Chapter 5 introduces new mechanisms for high-bandwidth and low-latency address translation. Chapter 6 explores the use of cache-conscious data placement as a means to improve data cache performance. Finally, Chapter 7 gives conclusions and suggests future directions to explore. The appendices include a detailed description of the SimpleScalar architecture (used by the experimental framework) and detailed experimental results for Chapter 5.

Chapter 2

Experimental Framework

This section details the experimental framework used for all the experiments in this thesis. Figure 2.1 illustrates the structure of the experimental framework.

2.1 Compiler Tools

All experiments were performed with programs compiled for the SimpleScalar architecture. The SimpleScalar architecture is a superset of the MIPS-I instruction set [KH92a] with the following notable differences:

- There are no architected delay slots for loads, stores, or control transfers.
- Loads and stores support additional addressing modes: indexed, auto-increment, and auto-decrement, for all data types.
- **SQRT** implements single- and double-precision floating point square roots.
- The architecture employs a 64-bit instruction encoding.

The entire instruction set is detailed in Appendix A.

As shown in Figure 2.1, C programs are compiled with a version of GNU GCC targeted to the SimpleScalar architecture. All programs are compiled with maximum optimization (-O3) and loop unrolling enabled (-funroll-loops). FORTRAN sources are compiled by first converting them to C with AT&T's F2C compiler. GNU GCC produces SimpleScalar assembly files which are assembled with a version of GNU GAS assembler ported to support SimpleScalar assembly. The produced object files are compatible with the MIPS ECOFF object format. The programs are linked with GNU GLD. Standard library calls are implemented with a version of GNU GLIBC ported to support the SimpleScalar instruction set and POSIX Unix system calls.

2.2 Simulation Methodology

The baseline simulator is detailed in Table 2.1. The simulator executes only user-level instructions, performing a detailed timing simulation of 4-way superscalar processor and the first level of instruction

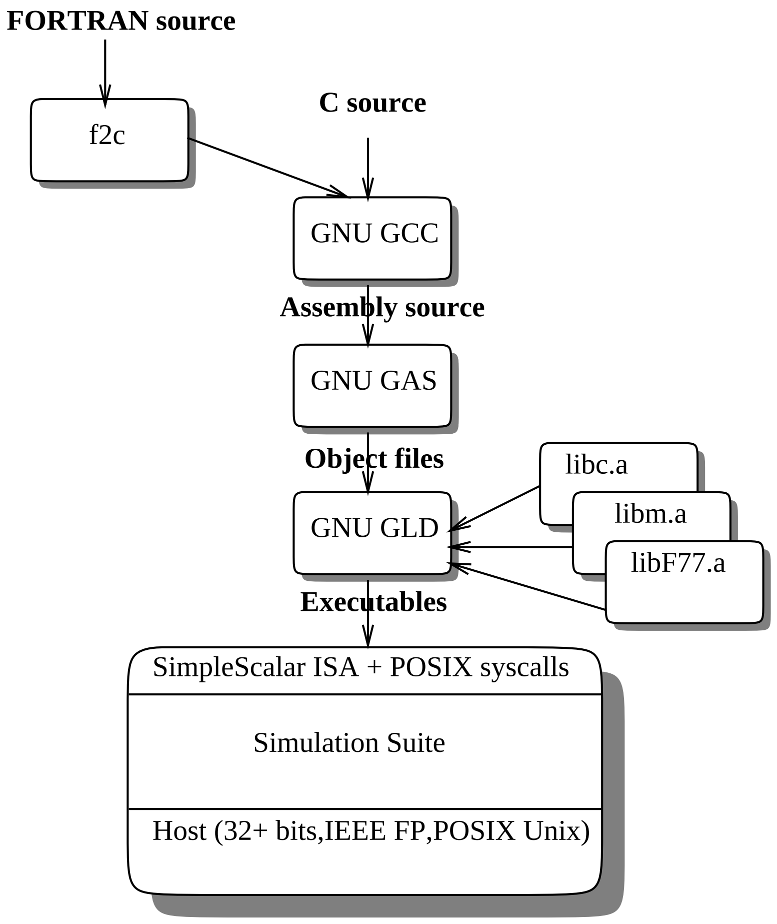


Figure 2.1: Experimental Framework.

and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or misprediction.

The simulator supports both in-order and out-of-order issue execution models. The in-order issue model provides no renaming and stalls whenever any data hazard occurs on registers. The out-of-order issue model employs a reorder buffer to rename registers and hold results of pending instructions. Loads and stores are placed into a load/store queue. Stores execute when all operands are ready; their store values, if speculative, are placed into the load/store queue. Loads may execute when all prior store addresses have been computed; their values come from a matching earlier store in the store queue or from the data cache. Speculative loads may initiate cache misses if the address hits in the TLB. If the load is subsequently squashed, the cache miss will still complete. However, speculative TLB misses are not permitted. That is, if a speculative cache access misses in the TLB, instruction dispatch is stalled until the instruction that detected the TLB miss is squashed or committed. Each cycle the reorder buffer commits completed instruction results in-order to the architected register file. When

Fetch Width	4 instructions
Fetch Interface	able to fetch any 4 contiguous instructions per cycle
I-cache	16k direct-mapped, 32 byte blocks, 6 cycle miss latency
Branch Predictor	1024 entry direct-mapped BTB with 2-bit saturating counters, 2 cycle misprediction penalty
In-Order Issue Mechanism	in-order issue of up to 4 operations per cycle, out-of-order completion, stalls on first data hazard
Out-of-Order Issue Mechanism	out-of-order issue of up to 4 operations per cycle, 16 entry reorder buffer, 8 entry load/store queue, loads execute when all prior store addresses are known
Architected Registers	32 integer, 32 floating point
Functional Units	4-integer ALU, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Functional Unit Latency (total/issue)	integer ALU-1/1, load/store-2/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1, FP MULT-4/1, FP DIV-12/12
D-cache	16k direct-mapped, write-back, write-allocate, 32 byte blocks, 6 cycle miss latency, two read ports, one write port (*e.g.*, dual-ported via replication), non-blocking interface, one outstanding miss per register, 6 cycle miss latency via a single-ported fully pipelined bus to main memory
Store Buffer	16 elements, non-merging

Table 2.1: Baseline Simulation Model.

stores are committed, the store value is written into the data cache.

The data cache modeled is dual-ported, direct-mapped, and non-blocking. Data cache bandwidth is limited, it can only service two loads or stores each cycle, either speculative or otherwise. For the in-order issue pipeline, cache writes are serviced in two cycles using a non-merging store buffer. The store buffer retires data written to the data cache during cycles it is unused. If a store executes and the store buffer is full, the entire pipeline is stalled and oldest entry in the store buffer is retired to the data cache.

2.3 Analyzed Programs

Table 2.2 details the programs analyzed, the language they are written in, and their inputs and options. The top group are integer codes, the bottom group are floating point codes. *Compress*, *Go*, *Perl*, and *Vortex* are from the SPEC '95 benchmark suite [SPE95]. *Elvis* is a VI-compatible text editor performing textual replacements in batch mode. *Ghostscript* is a postscript viewer rendering a page with text and graphics to a PPM-format graphics file. *Grep* performs regular expression matches in a large text file. *MPEG_play* is an MPEG compressed video decoder displaying a 79 frame compressed video file. *TFFT* performs real and complex FFTs on a randomly generated data set. *YACR-2* is a VLSI channel router routing a channel with 230 terminals. The remaining benchmarks are from the SPEC92 benchmark suite [SPE91].

Benchmark	Language	Input	Options/Modifications
Compress	C	in	
Elvis	C	unix.c	%s/for/forever/g, %s/./& /g
Eqntott	C	int_pri_3.eqn	
Espresso	C	cps.in	
GCC	C	1stmt.i	
Ghostscript	C	fast-addr.ps	-dNOPAUSE -sDEVICE=ppm -c quit
Grep	C	3x inputs.txt	-E -f regex.in
Perl	C	tests.pl	
MPEG_play	C	coil.mpg	
Sc	C	loada1	
Xlisp	C	li-input.lsp	Short input (queens 8)
YACR-2	C	input2	
Alvinn	C		NUM_EPOCHS=50
Doduc	Fortran	doducin	
Ear	C	short.m22	args.short
Mdljdp2	Fortran	mdlj2.dat	MAX_STEPS=150
Mdljsp2	Fortran	mdlj2.dat	MAX_STEPS=250
Ora	Fortran		ITER=60800
Spice2g6	Fortran	greycode.in	.tran .7n 8n
Su2cor	Fortran	su2cor.in	Short input
TFFT	Fortran		MEXPONENT=20, ITER=1
Tomcatv	Fortran		N=129

Table 2.2: Benchmark Programs.

Chapter 3

Fast Address Calculation

3.1 Introduction

All load instructions commence execution by computing the effective address of their access, *i.e.*, the base register value plus the offset or index register value. The latency of this operation, as shown in Figure 3.1a, is typically a single cycle in modern pipeline designs. For codes with an abundance of exposed load latency and good cache performance, *e.g.*, many integer codes, effective address calculation latency can account for a significant portion of total execution time.

This chapter focuses on a pipeline optimization, called *fast address calculation*, that reduces the latency of effective address calculation. The basic approach is illustrated in Figure 3.1b. The technique works by predicting early in the pipeline the effective address of a memory access and using the predicted address to speculatively access the data cache. A verification circuit determines if the address prediction is correct. If so, cache access is overlapped with non-speculative effective address calculation. Otherwise, the cache is accessed again in a later cycle using the correct effective address.

For fast address calculation to work well, the predictor must be both fast and accurate. A fast predictor minimizes impact on data cache access latency and processor cycle time. An accurate predictor eliminates most address calculation latency. In Section 3.2, analyses of programs reveal reference characteristics that when combined with properties of on-chip data cache can be exploited to produce a predictor that is both fast and accurate. Section 3.3 details the predictor design. Section 3.3 also describes a prediction verification circuit that is both fast and decoupled from the cache access critical path, ensuring minimal impacts on pipeline control circuitry and processor cycle time. When fast address calculation does fail, its failure modes are few and easy to identify. Section 3.5 describes how simple software support can significantly reduce the number of address mispredictions. In Section Section 3.6 analysis of prediction failure rates and program speedups are presented. Finally, Section 3.7 lists related work, and Section 3.8 summarizes the chapter.

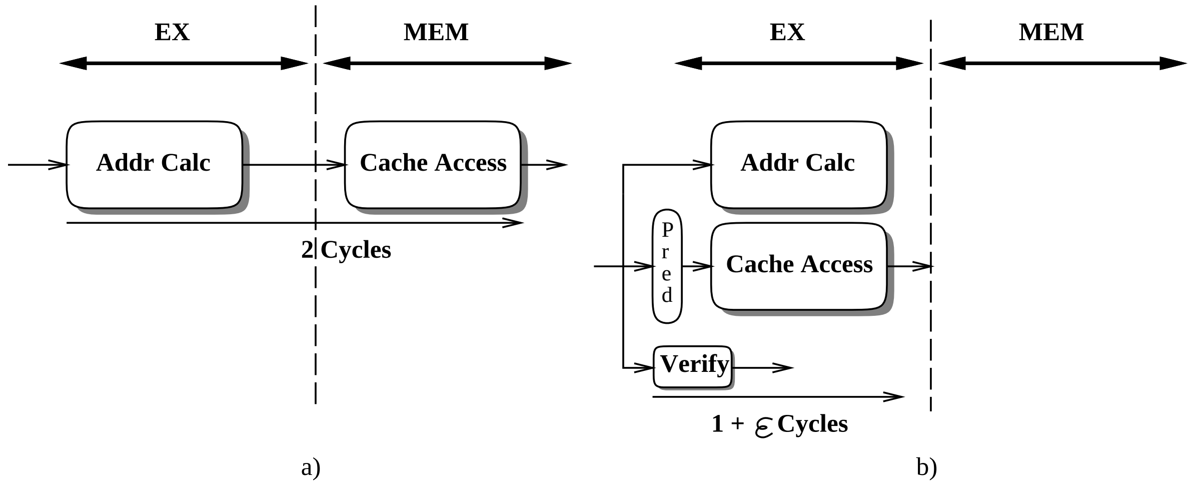


Figure 3.1: Fast Address Calculation.

3.2 Program Reference Behavior

Fast address calculation is best motivated by examining by examining the reference behavior of programs. The load instructions of several benchmarks (detailed in Chapter 2) compiled for the SimpleScalar architecture (detailed in Appendix A) were profiled. A number of key observations are made below.

3.2.1 Reference Type

Three prevalent modes of addressing occur during execution: *global*, *stack*, and *general pointer* addressing. Table 3.1 details the dynamic number of loads and stores executed by each program and the dynamic breakdown by reference type for loads.

Global pointer addressing is used to access small global (static) variables. The SimpleScalar (and MIPS) approach to global pointer addressing uses a reserved immutable register, called the *global pointer*, plus a constant offset to access variables in the *global region* of the program's data segment [CCH$^+$87]. The linker constructs the global region such that all variables referenced by name are grouped together near the target address of the global pointer. As shown in Table 3.1, global pointer addressing is prevalent in some programs, but not all. The frequency of this mode is highly dependent on the program structure and style.

Stack pointer addressing is used to access elements of a function's stack frame. The stack pointer register holds an address to the base of the stack frame of the currently executing function. Accesses to frame elements are made using the stack pointer register plus a constant offset. As is the case with global pointer addressing, stack pointer addressing is also a prevalent, but not an entirely dominating form of addressing.

Benchmark	Insts (Mil.)	Total Refs (Millions)		Loads		
		Loads	Stores	% Global	% Stack	% General
Compress	61.5	14.3	7.5	29.23	9.21	61.56
Eqntott	875.7	205.2	12.6	5.08	7.09	87.83
Espresso	474.4	109.1	25.9	3.91	5.26	90.83
Gcc	121.7	25.8	19.7	7.35	36.02	56.63
Sc	840.1	217.3	91.8	12.68	33.97	53.36
Xlisp	965.2	290.0	172.2	16.78	42.33	40.90
Elvis	249.3	67.7	28.6	1.63	6.33	92.04
Grep	122.2	42.1	1.5	1.13	3.64	95.23
Perl	203.6	50.0	34.2	10.69	43.15	46.16
YACR-2	386.8	59.0	7.1	7.61	32.72	59.68
Alvinn	1015.4	362.5	125.1	0.73	1.51	97.77
Doduc	1597.2	536.3	195.8	29.33	38.44	32.23
Ear	338.4	75.6	43.0	1.04	1.19	97.76
Mdljdp2	729.1	276.9	84.9	2.30	0.23	97.47
Mdljsp2	874.4	219.8	75.6	5.01	1.14	93.86
Ora	1057.1	231.2	98.2	33.19	33.14	33.67
Spice	1250.6	443.9	76.5	27.42	21.03	51.55
Su2cor	796.1	333.8	88.8	2.91	3.76	93.32
Tomcatv	464.2	172.8	35.9	4.68	4.07	91.25

Table 3.1: Program Reference Behavior.

The third mode of addressing, general pointer addressing, encompasses all other accesses. These accesses are the result of pointer and array dereferencing occurring during program execution. Quantitatively, all the benchmarks make heavy use of general pointer addressing with more than half of them using it for more than 80% of loads.

3.2.2 Offset Distribution

A typical load instruction has two inputs: base and offset. The base is added to the offset during effective address computation. In the SimpleScalar architecture, the base is supplied by a register and the offset may be supplied by either a signed 16-bit immediate constant, *i.e.*, `register+constant` addressing, or via a register, *i.e.*, `register+register` addressing.

Figure 3.2 shows the cumulative size distribution of offsets for the global, stack, and general pointer accesses of four benchmarks. (These curves are representative of the other benchmarks.) The graphs include loads using `register+register` addressing, in which case the base and offset of the load are determined by compiler convention.

The offsets applied to the global pointer are typically quite large, being that they are partial addresses. As one would expect, there is a strong correlation between the size of the offsets required and the aggregate size of the global data addressed by the program.

Stack pointer offsets tend to be large as well due to the large size of stack frames. While a stack frame may have only a few local variables, there are overheads not apparent to high-level language programmers which can greatly increase its size. These overheads include register save areas, dynamic

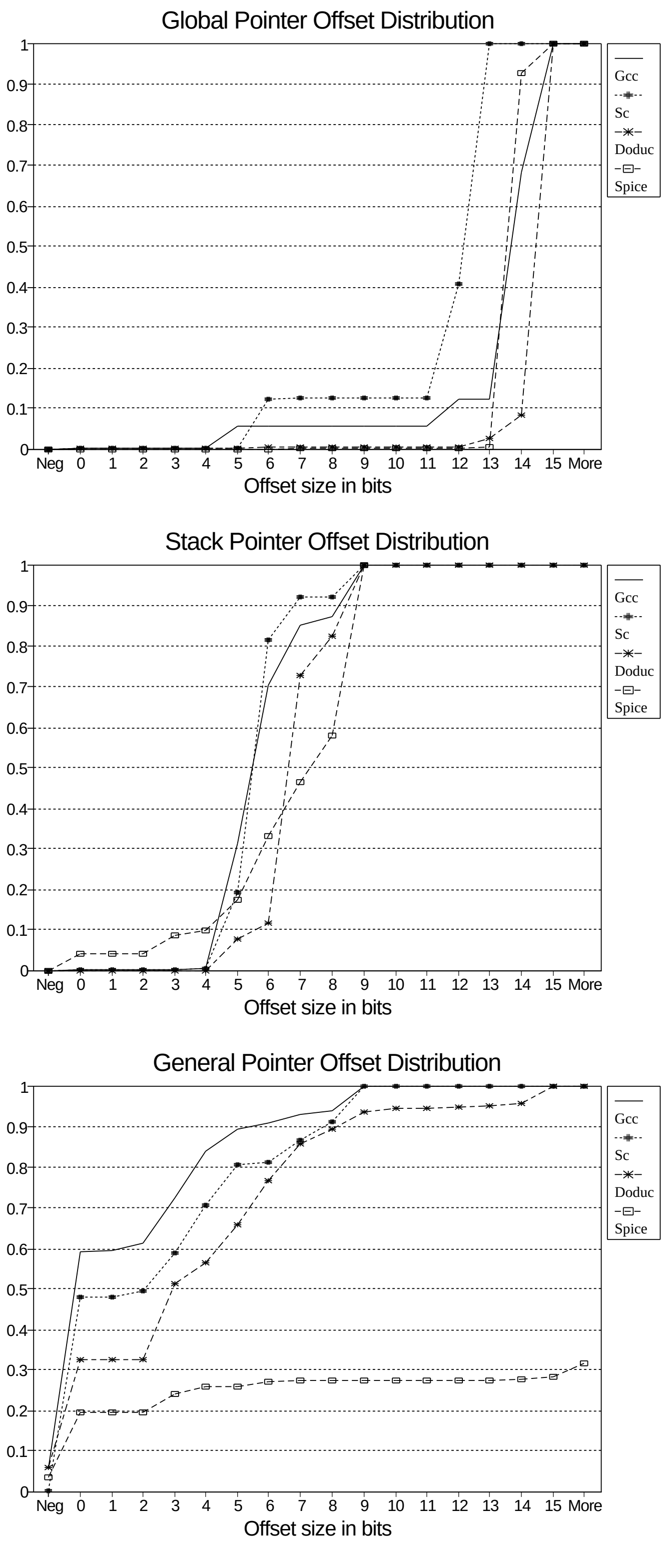


Figure 3.2: Load Offset Cumulative Distributions.

stack allocations, return address storage, among others.

For general pointer accesses, most load offsets are small. In fact, for a number of programs analyzed, *e.g.*, *GCC*, zero was the most common offset used. Zero offsets are primarily the product of array subscript operations where strength reduction [ASU86] of the subscript expression succeeded, pointer dereferences to basic types (*e.g.*, integers), and pointer dereferences to the first element of a structured (record) variable.

Non-zero offsets arise from primarily three sources: structure offsets, some array accesses, and array index constants. Structure offsets are small constants applied to pointers when accessing fields of a structured variable. Array base addresses are combined with index values to implement array accesses, *e.g.*, `a[i]`. The compiler used in this work, a port of GNU GCC to the SimpleScalar architecture, only generates this form of addressing when strength reduction of the subscript expression is not possible or fails. (When strength reduction is successful, a zero offset suffices.) Index constants are generated when part of an array subscript expression is constant, *e.g.*, `array[i+10]`. In addition, the compiler creates many index constants when unrolling loops. Index constants are usually small, although when in the higher dimension of a multi-dimensional array, they can become large.

For a few of the floating point programs, most notable *Spice*, there were a significant number of large offsets. This result indicates strength reduction of array accesses was generally ineffective. Consequently, the compiler had to rely on the brute force approach of adding the index variable to the base address of the array for every array access made, creating many large (index register) offsets.

Negative offsets are usually small immediate constants, generated by negative array subscript constants. They occur infrequently for both the integer and floating point intensive programs, *e.g.* for GCC they account for 5.7% of the general pointer loads and about 3.2% of all loads.

To summarize how these observations affect the design of an address predictor, it is clear that any prediction mechanism must: 1) perform well on all reference types, 2) perform well on small offsets, and 3) perform well on large offsets applied to the stack and global pointers. Secondary goals to good performance include support for predicting large index register offsets and support for small negative offsets. The following section presents a fast address calculation mechanism designed to satisfy these criteria while minimizing cost and pipeline impacts.

3.3 Fast Address Calculation

The fast address calculation mechanism predicts effective addresses early in the pipeline, thereby allowing loads to commence execution and complete earlier. To accomplish this task, an organizational property of on-chip data caches is exploited.

To minimize access time, on-chip caches are organized as wide two-dimensional arrays of memory cells (as shown in Figure 3.3). Each row of the cache array typically contains one or more data blocks

[WRP92, WJ94]. To access a word in the cache, the set index portion of the effective address is used to read an entire cache row from the data array and a tag value from the tag array. Late in the cycle, a multiplexor circuit uses the block offset part of the effective address to select the referenced word from the cache row. At approximately the same time, the tag portion of the effective address is compared to the tag value from the tag array to determine if the access hit in the cache. Hence, on-chip cache organizations require the set index portion of the effective address early in the access cycle and the block offset and tag portion late – after the cache row and tag have been read. The address prediction mechanism leverages this property of on-chip caches, allowing part of the address calculation to proceed in parallel with cache access.

Figure 3.3 shows a straightforward implementation of the effective address prediction mechanism for an on-chip direct-mapped cache, targeting ease of understanding rather than optimal speed or integration. The set index portion of the effective address is supplied very early in the cache access cycle by OR'ing together the set index portion of the base and offset. This operation implements a faster, *carry-free* form of addition, since it ignores any carries generated in or propagated into the set index portion of the address computation.[1] Because many offsets are small, the set index portion of the offset will often be zero, allowing this computation to succeed. For larger offsets, like those applied to the global or stack pointer, it is possible to employ software support to align pointer values, thereby increasing the likelihood that the set index portion of the base register value is zero.

In parallel with access of the cache data and tag arrays, full adders are used to compute the block offset and tag portion of the effective address. Later in the cache access cycle, the block offset is used by the multiplexor to select the correct word from the cache row, and the tag portion of the effective address is compared to the tag value read from the tag array.

Special care is taken to accommodate small negative offsets. The set index portion of negative offsets must be inverted, otherwise address prediction will fail. In addition, the prediction will fail if a borrow is generated into the set index portion of the effective address computation. The design assumes that offsets from the register file arrive too late for set index inversion, thus address predictions for these loads and stores fail if the offset is negative. This conservative design decision has little impact on the designs performance since negative index register offsets are extremely infrequent.

To complete the hardware design, the cache hit/miss detection logic is augmented with a circuit that verifies the predicted address. Using the result of this circuit, the cache controller and the instruction dispatch mechanism can determine if the access needs to be re-executed in the following cycle using the non-speculative effective address (computed in parallel with the speculative cache access). A misprediction is detected by looking for carries, either propagated into or generated in the set index part of the effective address computation. Four failure condition exist: 1) a carry (or borrow) is

[1]Technically, a carry-free addition requires an XOR function, but use of a simpler inclusive OR suffices here because the functions only differ when address prediction fails.

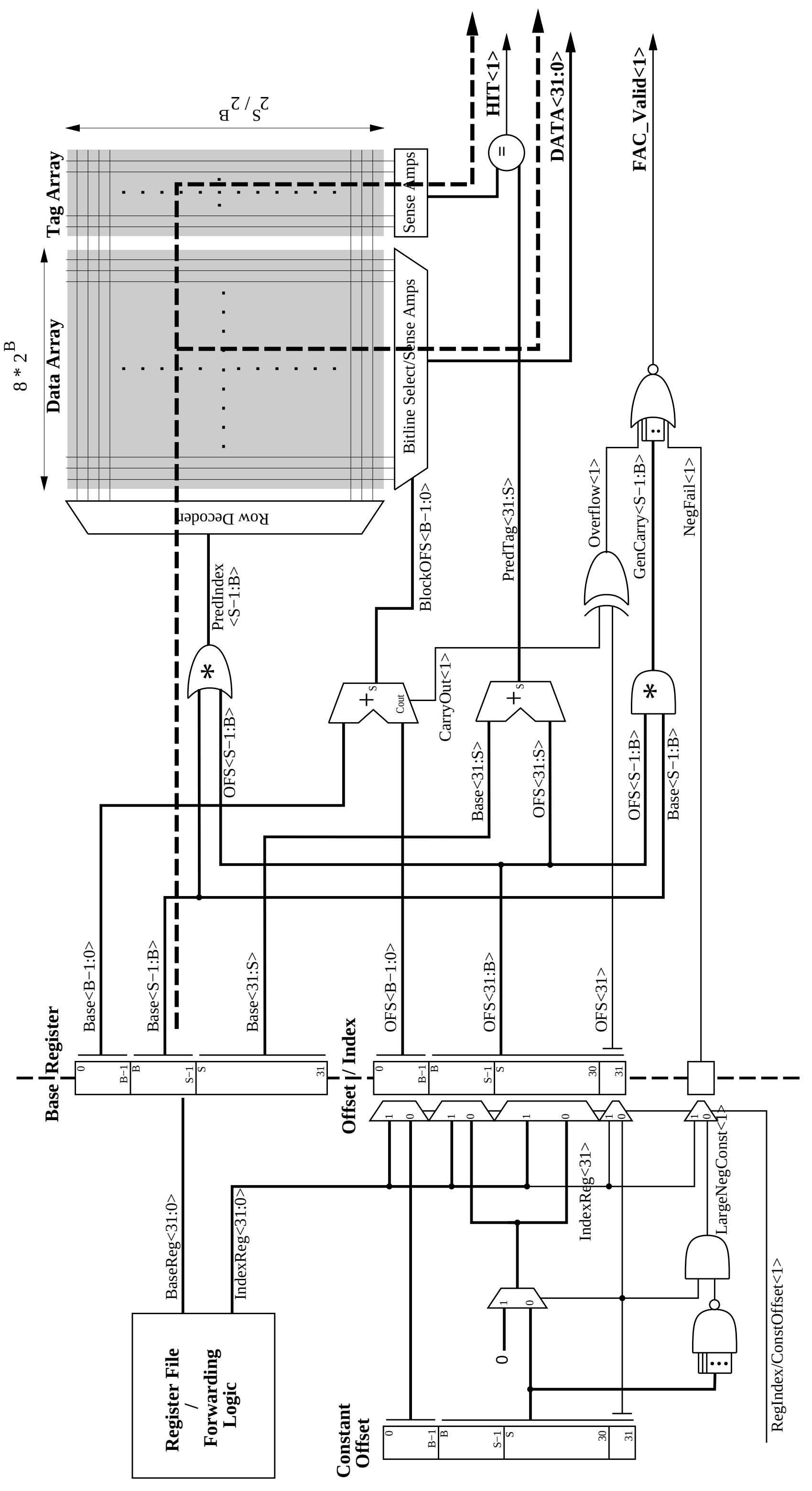


Figure 3.3: Pipeline Support for Fast Address Calculation. Bold lines indicate a bus, gates with an '*' signify a replicated gate for every line of the connected bus. 2^S is the size of a cache set in bytes, 2^B is the block size. The architecture shown has 32-bit addresses.

propagated out of the block offset portion of the effective address (signal *Overflow* in Figure 3.3), 2) a carry is generated in the set index portion of the effective address (signal *GenCarry*), 3) a constant offset is negative and too large (in absolute value) to result in an effective address within the same cache block as the base register address (signal *LargeNegConst*), or 4) an offset from the register file is negative (signal *IndexReg<31>*).

3.3.1 Further Design Considerations

The address prediction mechanism is designed to have minimal impact on the cache access critical path. The two typical cache access critical paths are denoted with bold dashed lines in Figure 3.3. While a much more detailed design would be required to demonstrate the exact impact the prediction mechanism has on the cache access critical path, it is possible point out a few design features of this circuit that indicate it should have minimal impact on cycle time. Three paths through the prediction circuit could affect the cache access cycle time. The first is through the tag adder. The tag portion of the effective address computation must arrive at the tag comparator before the tag array output becomes stable. For small addresses and large cache set sizes, this computation will likely complete before tag comparison. For large addresses and small cache set sizes, this computation may not complete in time. For these designs, the logical OR operation used to compute the set index could also be used to compute the tag portion of the address. All the experiments in Section 3.6 were run with and without full addition capability in the tag portion of the effective address computation and this capability was found to be of limited value. This result is to be expected considering the relatively small size of load offsets compared to cache set sizes, and the large alignments required on either the base or offset for carry-free addition to succeed on the set index portion of the address computation but fail on the tag portion.

The second path that could affect cycle time runs through the block offset adder. This result must arrive at the data array multiplexor before the data array produces stable output. For most cache designs, a 4- or 5-bit adder should suffice for this computation. The complexity of the block offset adder is small, on the order of the cache row decoders, hence, this part of the design will likely not impact the cache access critical path.

The third path that could affect cache access cycle time is through the prediction verification circuit. Since this circuit is completely decoupled from the normal cache access, it cannot affect the cache access cycle time as long as validation of the predicted effective address completes before the end of the clock cycle. This prediction verification signal, however, could affect processor cycle time if it is not available early enough to allow the processor control logic to schedule operations for the following cycle (a function of the success and failure of memory accesses in the current cycle). Since the verification circuit is very simple, it should have minimal impact on processor cycle time.

The OR operation used to generate the upper portion of the effective address is, as shown in Figure

3.3, directly on the critical path of data cache access. The impact of this operation may or may not impact processor cycle time, depending on the specifics of the pipeline design. In some designs, it may be possible to integrate the OR operation into the address decoder logic or the execute stage input latches, possibly reducing cycle time impact to zero. In any event, the predictor is only a single level of logic and should have minimal impact of cache access latency.

The fast address generation mechanism assumes that data cache access can start as soon as the set index part of the effective address is available. If this is not the case, *e.g.*, the cache is indexed with a translated physical address, fast address calculation will not work.

Another important consideration is the handling of stores. If stores execute later in the pipeline than speculative loads, it becomes possible for memory operations to execute out of order, introducing the possibility of data dependencies through memory being violated. Many options exist to eliminate this problem, each with varying degrees of cost, complexity, and performance.

The simplest approach for dealing with stores is to prevent loads from speculatively executing before than an earlier-issued, unfinished store. This approach, while inexpensive in that it need only track the existence of unfinished stores in the pipeline, limits performance due to the conservative assumption that all speculative loads and stores conflict. Many loads will be forced to wait for stores that do not conflict.

A slightly more aggressive approach is to force loads and stores to execute in order, but permit stores to also use fast address calculation. Using this strategy, stores will complete earlier in the pipeline as well, reducing the frequency of stalled loads. Of course, this design must ensure that misspeculated stores can be undone. For designs employing a store buffer [Jou93] and a two-cycle store sequence, this may not pose a problem. In the first cycle, cache tags are probed to see if the access hits in the cache, in the second (possibly much later) cycle, the store is made to the cache. Since the fast address calculation mechanism determines the correctness of the address after one cycle, the store buffer entry can simply be reclaimed or invalidated if the effective address is incorrect. A design similar to this one is adopted in the detailed timing simulator used in Section 3.6.

The most ambitious approach is to check for possible conflicts between loads and stores, and only stall loads that reference data produced by earlier-issued, unfinished stores. This design will have the best performance, but also the highest cost and complexity since speculative load addresses must be compared against all earlier-issued, unfinished stores. When store addresses are not known, loads must stall or execute with possibility of later recovery if a conflict does arise.

3.4 Working Examples

Figure 3.4 shows a number of example address computations using fast address calculation. Example (a) shows a pointer dereference. Since the offset is zero, no carry is generated during address calculation

```
                                              <-- Tag -->  <-- Index-->  <BO>
          load r3, 0(r8)
          r8            = 0x0001ac00   000...01 10   10 1100 0000   0000
    (a)   offset        = 0x00000000   000...00 00   00 0000 0000   0000
          prediction    = 0x0001ac00   000...01 10   10 1100 0000   0000
          actual result = 0x0001ac00   000...01 10   10 1100 0000   0000

          load r3, 2436(gp)
          gp            = 0x00010000   000...01 00   00 0000 0000   0000
    (b)   offset        = 0x00000984   000...00 00   00 1001 1000   0100
          prediction    = 0x00010984   000...01 00   00 1001 1000   0100
          actual result = 0x00010984   000...01 00   00 1001 1000   0100

          load r3, 102(sp)
          sp            = 0x7fff5b84   011...11 01   01 1011 1000   0100
    (c)   offset        = 0x00000066   000...00 00   00 0000 0110   0110
          prediction    = 0x7fff5bea   011...11 01   01 1011 1110   1010
          actual result = 0x7fff5bea   011...11 01   01 1011 1110   1010
                                                         Generated
          load r3, 364(sp)                               Carry
          sp            = 0x7fff5b84   011...11 01   01 1011 1000   0100
    (d)   offset        = 0x0000016c   000...00 00   00 0001 0110   1100
          prediction    = 0x7fff5be0   011...11 01   01 1011 1110   0000
          actual result = 0x7fff5cf0   011...11 01   01 1100 1111   0000

                                                 Generated   Propagated
                                                 Carry       Carry
```

Figure 3.4: Examples of Fast Address Calculation. The address bits are split into the tag, index and block offset fields corresponding to a 16K byte direct-mapped data cache with 16 byte blocks.

and the predicted address is correct. Example (b) shows an access to a global variable though the global pointer. In this example, the global pointer is aligned to a large power of two, so carry-free addition is sufficient to generate the correct address. In example (c), carry-free addition is sufficient to predict the portion of the address above the block offset, but full addition is required to compute the block offset. Since a carry in not generated out of the block offset portion of the effective address computation, the prediction succeeds. Finally, example (d) shows a stack frame access with a larger offset. In this case, the predicted address is incorrect because a carry is propagated out of the block offset and generated in the set index portion of the effective address computation.

3.5 Increasing Prediction Performance with Software Support

Software support can increase the prediction accuracy of fast address calculation by reducing the need for full-strength addition in the set index portion of the effective address calculation. This task is accomplished by decreasing the size of offset constants and index register values, and by increasing the alignment of base register pointers. It is important to note, however, that software support is only used as a mechanism to improve performance, it is not required. As shown in Section 3.6, fast address calculation is a remarkably resilient mechanism, providing good speedups even without software support. Software support targets each reference type (*i.e.*, global, stack, and general pointer), ensuring a high prediction accuracy for each.

3.5.1 Software Support for Global Pointer Accesses

Since the linker controls the value of the global pointer and offsets applied to it, it is trivial to ensure all global pointer accesses are correctly predicted. The linker can limit all offsets off the global pointer to be positive and relocate the global region to an address starting at a power-of-two boundary larger than the largest offset applied. Using this approach, carry-free addition will suffice for any global pointer access.

3.5.2 Software Support for Stack Pointer Accesses

As is the case with global pointer addressing, the compiler completely controls the value of the stack pointer and the organization of stack frames. By aligning the stack pointer and organizing the stack frame so as to minimize the size of offset constants, it is possible to ensure that all stack pointer accesses are correctly predicted.

The compiler can enforce a program-wide stack pointer alignment by initially aligning the stack pointer and then forcing all frame sizes to be a multiple of the alignment. Using this approach, carry-free addition will suffice for address computations in which the offset is smaller than the alignment of the stack pointer.

Nearly all stack pointer addressing is performed on scalar variables. By sorting the elements of the stack frame such that the scalars are located closest to the stack pointer, the compiler can minimize the size of offsets constants applied to the stack pointer.

A few programs, most notably numeric codes, have functions with very large large stack frames that benefit little from a fairly small program-wide stack pointer alignment. For stack frames larger than the program-wide stack pointer alignment, the compiler can employ an alternative approach: the stack pointer can be explicitly aligned to a larger alignment by AND'ing the stack pointer with the adjusted power-of-two frame size times a negative one. Since this approach creates variable size stack frames, a frame pointer will be required for access to incoming arguments not in registers. In addition, the previous stack pointer value must be saved at function invocation and restored when the function returns.

The impact of this approach is increased stack memory usage – frame size overheads can grow as much as 50%. If a program uses more memory, cache and virtual memory performance could suffer. The compiler should provide programmers with an option to limit stack pointer alignments, thereby providing a mechanism to control memory overhead.

3.5.3 Software Support for General Pointer Accesses

For general pointer accesses, offsets are typically small and positive, the result of index constants and structure offsets. The compiler can increase the likelihood of a carry not being generated out of the block offset portion of the effective address by aligning variable allocations to a multiple of the target cache block size.

For global and local variables, alignments can be increased to the next power-of-two larger than the size of the variable, bounded by the block size of the target cache. For dynamic storage allocations, alignments can be increased in the same manner by the dynamic storage allocator, *i.e.*, `malloc()`. Since many languages, *e.g.*, C, employ type-less dynamic storage allocation, the allocator lacks the type information required to minimize alignment overheads. Aligning any dynamic allocation to the maximum alignment required, *i.e.*, the block size of the target cache, will ensure any alignments within the allocation are not perturbed. `Alloca()` allocations (used heavily by the benchmarks *GCC* and *Grep*) can employ a similar approach for dynamic storage allocation within stack frames.

To encourage proper alignments of interior objects, *e.g.*, array elements, the compiler can increase the size of array elements to the next larger power of two, bounded by the block size of the target cache. Since basic types, *e.g.*, integers and floats, are already a power of two in size, overheads will only be incurred for arrays of structured variables. The compiler need not, however, enforce stricter alignments on structure fields, as this would serve to spread out elements of a structure. Experimental results to date have indicated that dense structures is consistently a bigger win than stricter alignments within structured variables.

As is the case with larger stack frame alignments, these techniques can increase memory usage by as much as 50%. Hence, a compiler option should be available to limit the alignments placed on variable addresses and sizes.

In addition to the changes described above, modifications made to existing optimization routines can also improve the performance of optimized code. Specifically, common subexpression elimination (CSE) should give preference to aligned pointer subexpressions. In addition, the address cost function used by the strength-reducer should make `register+register` addressing seem very expensive. This change will make the compiler work harder to strength-reduce loop induction variables, resulting in more zero-offset loads and stores within loops.

3.6 Experimental Evaluation

This section evaluates the effectiveness of fast address calculation by examining the performance of programs running on a detailed timing simulator of a 4-way in-order issue processor extended to support fast address calculation. The performance of programs is examined in a number of contexts: with and without software support, with and without `register+register` addressing mode speculation, and with varied cache block sizes.

3.6.1 Methodology

All experiments were performed on the baseline in-order issue timing simulator detailed in Chapter 2. The simulator implements a detailed timing model of a 4-way in-order issue superscalar processor and the first level of instruction and data cache memory. The simulator implements a traditional five stage pipeline, *i.e.*, all ALU and memory operations begin execution in the third stage of the pipeline (EX), and non-speculative loads and stores execute in the fourth stage (MEM), resulting in a non-speculative load latency of 2 cycles. The data cache modeled is a dual-ported 16k direct-mapped non-blocking cache. Stores are serviced in two cycles using a 16-entry non-merging store buffer.

A number of modifications were made to the simulator to support fast address calculation. Loads may speculatively access the data cache using fast address calculation given that there are no unfinished stores in the pipeline. To reduce the number of stalled loads, stores are also allowed to utilize fast address calculation to speed their execution. When executed speculatively, a store is entered into a store buffer. If the store address is mispredicted, the store is re-executed and its address in the store buffer is updated. When fast address calculation fails (in the EX stage of the modified pipeline), the access can re-execute in the following cycle (in the MEM stage).

All programs were compiled for the SimpleScalar architecture (detailed in Appendix A). Compiler support for fast address calculation, as described in Section 3.5, was added to GNU GCC, linker support was added to GNU GLD. The following fast address calculation specific optimization were applied to both the programs and the system libraries (*e.g.* `libc.a`):

Global pointer alignment: The linker, GNU GLD, aligned the global pointer to a power-of-two value (unbounded) larger than the largest offset applied to it. All global pointer offsets were restricted to be positive. (Normally, the initial value of the global pointer is dependent on the size of the data segment and is not aligned.)

Stack pointer alignment: The compiler, GNU GCC, rounded all stack frame sizes up to the next multiple of 64 bytes, resulting in a program-wide stack pointer alignment of 64 bytes. (Normally, GCC maintains an 8 byte alignment on the stack pointer.) Frames larger than 64 bytes enforce larger stack pointer alignments of up to 256 bytes by explicitly aligning the stack pointer on function invocation and restoring the original value on function return.

Static variable alignments: Static allocations were placed with an alignment equal to the next power-of-two larger or equal to the size of the variable, not exceeding 32 bytes.

Dynamic variable alignments: `malloc()` and `alloca()` allocation alignments were increased from the default of 8 to 32 bytes.

Structured variable alignments: Internal structure offsets were not changed, however, structure sizes were increased to the next power-of-two larger than or equal to the normal structure size, with

Program	Insts (Mil.)	Cycles (Mil.)	Loads (Mil.)	Stores (Mil.)	Miss Ratio I-cache	Miss Ratio D-cache	Mem Usage (K bytes)
Compress	61.5	58.4	14.3	7.5	0.00	15.65	438
Eqntott	875.7	627.4	205.2	12.6	0.00	4.70	2704
Espresso	474.4	374.0	109.1	25.9	0.16	2.54	400
Gcc	121.7	109.5	25.8	19.7	1.63	3.09	1416
Sc	840.1	811.3	217.3	91.8	0.17	7.11	493
Xlisp	965.2	850.6	290.0	172.2	0.71	1.72	115
Elvis	249.3	207.1	67.7	28.6	0.50	0.44	90
Grep	122.2	139.0	42.1	1.5	0.03	3.88	377
Perl	203.5	214.4	50.0	34.2	3.63	4.63	3625
YACR-2	386.9	261.0	59.0	7.1	0.01	0.67	195
Alvinn	1015.4	1236.2	362.5	125.1	0.02	4.21	507
Doduc	1597.2	1820.5	536.3	195.8	1.55	2.26	144
Ear	338.4	416.5	75.6	43.0	0.00	0.02	208
Mdljdp2	729.1	787.3	276.9	84.9	0.00	1.52	267
Mdljsp2	874.4	1110.7	219.8	75.6	0.00	1.52	227
Ora	1057.1	1112.9	231.2	98.2	0.00	0.33	50
Spice	1250.6	1388.9	443.9	76.5	0.36	10.16	3227
Su2cor	796.1	1073.3	333.8	88.8	0.08	23.55	4131
Tomcatv	464.2	431.6	172.8	35.9	0.01	8.63	945

Table 3.2: Program Statistics without Software Support.

the overhead not exceeding 16 bytes.

The analyzed programs include both integer and floating point codes. Table 3.2 lists the programs analyzed and their baseline execution statistics without software support. The integer codes are in the top group, the floating point codes in the bottom group. The benchmarks are detailed in Chapter 2. Shown are the number of instructions, execution time in cycles on the baseline simulator (*i.e.*, a 4-way in-order issue superscalar processor without fast address calculation support), total loads and stores executed, instruction and data cache miss ratios for 16k byte direct-mapped caches with 32 byte blocks, and total memory size.

3.6.2 Prediction Performance

Figure 3.5 shows the prediction failure rates for loads with a 32 byte cache block size, *i.e.*, the case where the prediction circuitry is able to perform 5 bits of full addition in the block offset portion of the effective address computation. The figure shows the failure rates as a percentage of total speculated accesses when running with only fast address calculation hardware support (*i.e.*, H/W), with hardware and software support (*i.e.*, H/W+S/W), and with hardware and software support but no speculation of `register+register` mode accesses (*i.e.*, H/W+S/W - R+R).

Without any software support, the percentage of incorrect predictions is quite high, suggesting that many pointers are insufficiently aligned to allow for carry-free addition in the set index part of the effective address calculation. Some of the programs, however, have very low prediction failure rates, *e.g.*, *Elvis* and *Alvinn*. For these programs, the frequency of zero-offset loads is very high, indicating

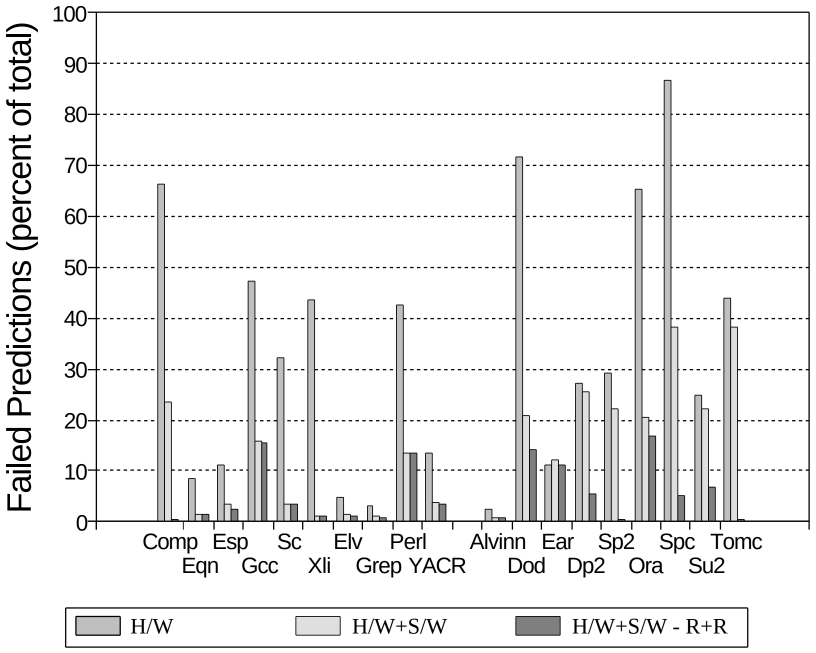


Figure 3.5: Prediction Performance. The cache block size is 32 bytes, *i.e.*, the predictor supports 5 bits of full addition in the block offset portion of the address computation.

that prediction is working fairly well because effective address computation is not required. Prediction performance with 16 byte blocks was also examined (detailed results can be found in an earlier paper on fast address calculation [APS95]). Overall the prediction failure rate decreased slightly when the block size increased, since misaligned pointers benefited from more full addition capability in the fast address calculation mechanism.

As Figure 3.5 shows, software support was extremely successful at decreasing the failure rate of effective address predictions. Compared to the prediction failure rates without software support, the percentage of loads and stores mispredicted is consistently lower, the prediction failure rate decreasing by more than 50% in many cases.

Even with software support, a number of the programs, *e.g.* *Spice* and *Tomcatv*, still possessed notably high address misprediction rates. To better understand their cause, loads and stores were profiled to determine which were failing most. The two dominating factors leading to address prediction failures were:

Array index failures: Many loads and stores using `register+register` addressing resulted in failed predictions. The compiler only uses this addressing mode for array accesses, and then only when strength-reduction fails or is not possible, *e.g.*, an array access not in a loop. (If strength-reduction is successful, a zero-offset load or store suffices.) As one would expect, array index values are

Benchmark	Insts % Change	Cycles % Change	Loads % Change	Stores % Change	Miss Ratio Change I-cache	Miss Ratio Change D-cache	Mem Usage %Change
Compress	-0.38	+0.50	+0.00	-0.01	-0.00	+0.00	+1.14
Eqntott	+0.40	+0.15	+1.39	-0.04	-0.00	+0.00	+0.11
Espresso	-0.33	-0.59	-0.01	-0.03	-0.00	-0.00	+13.25
Gcc	+0.86	+1.76	+1.01	+0.90	-0.02	+0.00	+1.20
Sc	-0.06	+0.12	-0.01	-0.02	-0.00	+0.00	+1.01
Xlisp	-0.09	+0.80	-0.12	-0.08	-0.01	+0.00	+15.65
Elvis	-0.17	+0.27	-0.30	-0.36	-0.01	-0.00	+1.11
Grep	-0.72	-1.14	-0.95	-24.82	-0.00	-0.00	+3.18
Perl	-0.95	+0.22	-0.53	+0.71	-0.04	+0.01	+20.03
YACR-2	+0.42	-0.17	+2.33	-0.01	-0.00	-0.00	+0.00
Alvinn	+0.12	-0.03	-0.01	-0.00	-0.00	-0.00	+0.20
Doduc	+0.13	+0.21	+0.06	-0.25	-0.02	-0.00	+2.78
Ear	+0.04	+0.09	+0.18	-0.02	-0.00	+0.00	+2.40
Mdljdp2	+0.02	-0.34	+0.05	-0.00	-0.00	-0.00	+1.87
Mdljsp2	-0.07	+0.03	+0.10	+0.01	-0.00	+0.00	+0.44
Ora	+0.24	+1.51	+0.00	+0.00	-0.00	+0.00	+10.00
Spice	-0.13	+0.46	+0.12	+0.01	-0.00	+0.00	+0.28
Su2cor	+0.59	+0.54	+0.24	+0.51	-0.00	-0.00	+0.12
Tomcatv	+0.00	+0.03	-0.00	+0.00	-0.00	+0.00	+0.32

Table 3.3: Program Statistics with Software Support.

typically larger than the 32 byte alignment placed on arrays, resulting in high prediction failure rates. The bars labeled "H/W+S/W - R+R" in Figure 3.5 show the prediction failure rate for all loads and stores except those using `register+register` mode addressing. For many programs, array index operations are clearly a major source of mispredicted addresses.

Domain-specific storage allocators: A number of programs, most notably *GCC*, used their own storage allocation mechanisms, this led to many pointers with poor alignment and increased prediction failure rates.

These factors, however, are not without recourse. A strategy for placement of large alignments could eliminate many array index failures; for example, in the case of *Spice* aligning a single large array to its size would eliminate nearly all mispredictions. In addition, program tuning could rectify many mispredictions due to domain-specific allocators.

Table 3.3 shows the program statistics for the benchmarks compiled with software support. The table lists the percent change in instruction count, cycle count (on the baseline simulator without fast address calculation), number of loads and stores, and total memory size with respect to the program without fast address calculation optimizations (the results in Table 3.2). For the instruction and data cache (16k byte direct-mapped), the table lists the absolute change in the miss ratio.

Generally, fast address calculation specific optimizations did not adversely affect program performance on the baseline simulator. The total instruction count as well as the number of loads and stores executed are roughly comparable. The cycle count differences (without fast address calculation support) were small; the largest difference was 1.76% more cycles for *GCC*. Cache miss ratios saw little impact for both the instruction and data caches. Total memory usage was also examined, as it is an indirect metric of virtual memory performance. The largest increases experienced were for *Perl*, *Espresso*, and *Xlisp* where memory demand increased by as much as 20%. However, the absolute change in memory consumption for these programs was reasonably small, much less than a megabyte for each. In addition, we examined TLB performance running with a 64 entry fully associative randomly replaced data TLB with 4k pages and found the largest absolute difference in the miss ratio to be less than 0.1% (for *Perl*). Given these metrics, fast address calculation specific optimizations are not expected to adversely impact program performance when executing on a machine without fast address calculation support.

3.6.3 Program Performance

Prediction performance does not translate directly into program run-time improvements. A successful effective address prediction may or may not improve program performance, depending on whether or not the access is on the program's critical path. To gauge the performance of fast address calculation in the context of a realistic processor model, baseline program performance was compared to the performance of programs running on the baseline timing simulator extended to support fast address calculation (detailed in Section 3.6.1).

Figure 3.6 shows execution speedups as a function of three design parameters: with and without software support, with 16 and 32 byte blocks, and with and without `register+register` mode speculation. Also shown are the average speedups for the integer and floating point codes, weighted by the run-time (in cycles) of the program. All speedups are computed with respect to the execution time (in cycles) of the baseline program (no fast address calculation specific optimizations) running on the baseline simulator.

On the average, fast address calculation without software support improves the performance of integer programs by 14%, largely independent of block size or speculation of `register+register` mode accesses. The floating point programs show a smaller speedup of 6%. This is a very positive result – even without software support, one could expect program performance to consistently improve, and by a sizable margin for integer codes.

The combination of software and hardware manages to give somewhat better performance improvements. An average speedup of 19% was found for the integer codes, with no individual program speedup less than 6%. For the floating point programs, speedups were smaller in magnitude with an average of 7.5%. The compiler optimizations have a positive effect on most programs, and tend to assist more where the hardware-only approach is ineffective, (*e.g.*, Compress).

The consistent speedup across all programs is a very important property of fast address calculation for it allows the designer a trade-off between a longer cycle time and increased performance for integer

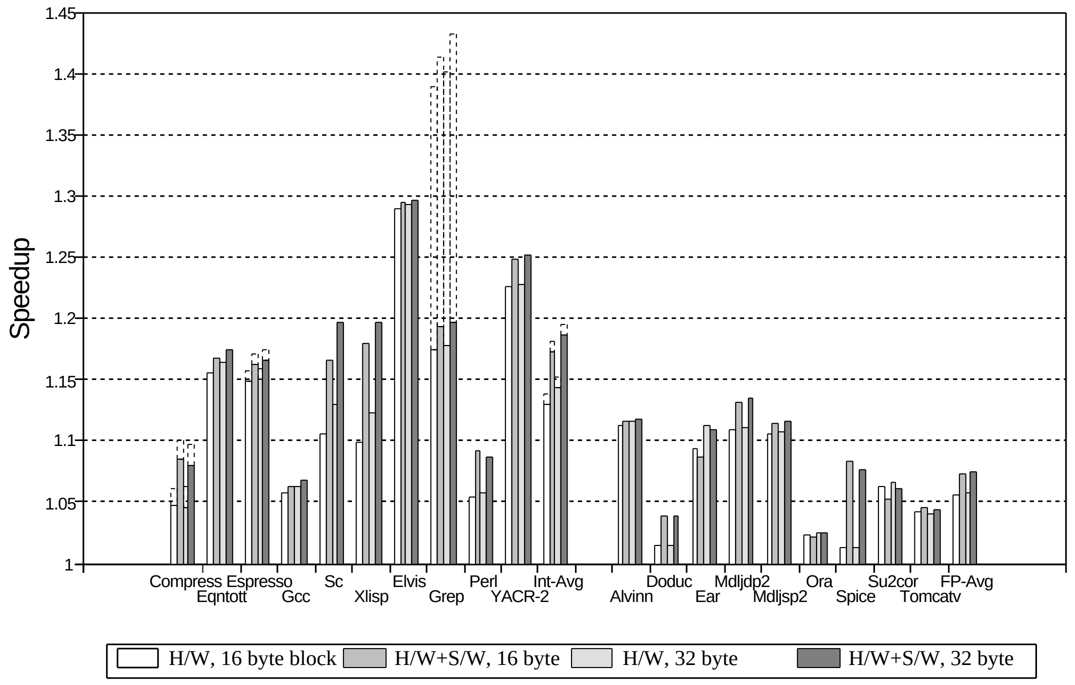


Figure 3.6: Speedups, with and without Software Support. Speedups shown are over baseline model execution time for a 16K byte data cache with 16 and 32 byte blocks. The dashed bars indicate where there was improvement with `register+register` mode speculation.

programs. For example, if fast address calculation increases the cycle time by 5%, the average floating point performance will still improve slightly while the average integer performance will improve by a sizable 13.5%.

All simulations were run with both 16 and 32 byte cache blocks, *i.e.* where the prediction circuitry is able to perform 4 or 5 bits of full addition in parallel with cache access, respectively. The impact of increasing the block size was positive but small in magnitude for most programs, resulting in an overall difference of less than 3% for all experiments. In all cases, the improvement in the average performance was less than 1%.

Considering the prediction failure rate of `register+register` mode addressing, program performance was examined with and without speculation of this mode. The only programs which experienced any change in performance were *Compress*, *Espresso*, and *Grep*. *Grep*'s stellar performance improvement is the result of many `register+register` accesses to small arrays which benefit from limited full addition in the block offset portion of address computation. Average speedup increased less than 1% for the integer codes and was unchanged for the floating point codes. The overall lackluster improvement is the result of high failure rates when predicting `register+register` mode addresses. Without a means to effectively predict `register+register` mode loads and stores, their speculation appears to have little overall benefit, especially in light of increased demand on cache bandwidth.

Benchmark	R+R Speculation		No R+R Speculation	
	H/W only	H/W + S/W	H/W only	H/W + S/W
Compress	24.21	16.47	8.92	+0.00
Eqntott	3.02	1.40	2.95	1.33
Espresso	5.38	3.06	3.95	2.01
Gcc	20.85	7.26	20.31	6.65
Sc	13.10	2.71	13.10	2.65
Xlisp	17.43	1.17	17.43	1.17
Elvis	2.49	1.31	2.46	1.04
Grep	2.63	1.10	1.94	0.40
Perl	19.40	7.98	18.55	7.41
YACR-2	5.07	3.87	4.68	3.49
Alvinn	1.33	1.00	1.33	1.00
Doduc	22.68	13.49	17.37	7.28
Ear	8.95	10.32	8.95	10.32
Mdljdp2	21.11	19.56	7.06	3.28
Mdljsp2	18.81	16.32	2.68	+0.00
Ora	24.72	14.03	21.03	10.97
Spice	45.86	32.44	7.46	3.07
Su2cor	19.92	20.33	7.65	5.72
Tomcatv	32.52	33.56	4.22	2.77

Table 3.4: Memory Bandwidth Overhead. The numbers shown are the total failed speculative cache accesses as a percentage of total references.

Table 3.4 shows the increase in the number of accesses to the data cache (in percent of total accesses without speculation). These numbers reflect memory accesses that were mispredicted and actually made during execution; in other words, these results are the overhead in cache accesses due to speculation. Without compiler support, a large fraction of the speculative memory accesses are incorrect (as shown in Figure 3.5), requiring more cache bandwidth, as much as 45% for *Spice*. The compiler optimizations cut down this extra bandwidth significantly. For most programs the increase in the required cache bandwidth is less than 10%, and the maximum increase in bandwidth is less than 34%; without `register+register` mode speculation the bandwidth increases are at most 11%. Despite the increase in cache accesses due to speculation, the impact of store buffer stalls was surprisingly small, typically less than a 1% degradation in the speedups attained with unlimited cache store bandwidth. (The results in Figure 3.6 include the performance impact of store buffer stalls.)

3.7 Related Work

Golden and Mudge [GM93] explored the use of a load target buffer (LTB) as a means of reducing load latencies. An LTB, loosely based on a branch target buffer, uses the address of a load instruction to predict the effective address early in the pipeline. They conclude the cost of the LTB is only justified when the latency to the first level data cache is at least 5 cycles. Fast address calculation has two distinct advantages over the LTB. First, it is much cheaper to implement, requiring only a small adder circuit and a few gates for control logic. Second, it is more accurate at predicting effective addresses because it predicts addresses using the operands of the effective address calculation, rather than the address of the load. In addition, fast address calculation employs compile-time optimization to further improve performance.

An earlier paper by Steven [Ste88] goes as far as proposing a four stage pipeline that eliminates the address generation stage and executes both memory accesses and ALU instructions in the same stage. Steven proposes the use of an OR function for all effective address computation. Steven's approach was only intended as a method for speeding up stack accesses, all other accesses require additional instructions to explicitly compute effective addresses. The performance of this pipeline organization was not evaluated.

AMD's K5 processor [Sla94] overlaps a portion of effective address computation with cache access. The lower 11 bits of the effective address is computed in the cycle prior to cache access. The entire 32 bit effective address is not ready until late into the cache access cycle, just in time for the address tag check.

The idea of exploiting the two-dimensional structure of memory is being used in several other contexts, such as paged mode DRAM access [HP90]. Katevenis and Tzartzanis [KT91] proposed a technique for reducing pipeline branch penalties by rearranging instructions so that both possible targets of a conditional branch are stored in a single instruction cache line. The high bandwidth of the cache is used to fetch both targets of a branch instruction. The branch condition is evaluated while the instruction cache is accessed and the condition outcome is used to late-select the correct target instruction.

3.8 Chapter Summary

This chapter presented the design and evaluation of fast address calculation, a novel approach to reducing the latency of load instructions. The approach works by predicting early in the pipeline the effective address of a memory access and using the predicted address to speculatively access the data cache. If the prediction is correct, the cache access is overlapped with non-speculative effective address calculation. Otherwise, the cache is accessed again in the following cycle, this time using the correct effective address.

The predictor's impact on the cache access critical path is minimal. The prediction circuitry adds only a single OR operation before cache access can commence. In addition, verification of the predicted effective address is fast and completely decoupled from the cache access critical path.

Detailed timing simulations show that without software support the prediction accuracy of the basic hardware mechanism varies widely. For the programs examined, prediction success rates ranged from 13 to 98%. However, detailed timing simulations of the programs executing on a superscalar processor resulted in consistent program speedups – an average speedup of 14% for the integer codes and 6%

for the floating point codes. With the addition of simple compiler and linker support, prediction accuracy increased significantly, with success rates ranging from 62 to 99%. Simulated performance with software support increased as well, resulting in average speedup of 19% for the integer codes and 7.5% for the floating point codes. Increases in cache bandwidth demand due to speculation were also measures found to be generally very low. With software support, speculation required at most 34% more accesses. By preventing `register+register` addressing mode speculation, extra cache bandwidth requirements drop to at most 11% more accesses, with little impact on overall performance.

The consistent performance advantage of fast address calculation coupled with the low cost of its use, in terms of hardware support, software support, and cache bandwidth demand, should makes the approach an attractive option for reducing load latency.

Chapter 4

Zero-Cycle Loads

4.1 Introduction

In the previous chapter, fast address calculation (illustrated in Figure 4.1a) was introduced as a technique to overlap address calculation with data cache access, thereby eliminating the extra cycle needed for address calculation. For most pipeline designs, address calculation latency comprises at most half of the latency of loads that hit in the data cache, leaving one or more cycles of data cache access latency still exposed to extend execution critical paths or stall instruction issue.

In this chapter, the latency reduction capability of fast address calculation is extended by combining it with an early-issue mechanism, reducing load latency by yet another cycle. As shown in Figure 4.1b, the approach allows loads to complete up to two cycles earlier than traditional designs. These loads, termed *zero-cycle loads* because they have no visible latency to dependent instructions, produce a result by the time they reach the execute stage of the pipeline. Subsequent dependent instructions can enter the execute stage of the pipeline at the same time unencumbered by load instruction hazards. Programs executing on a processor with support for zero-cycle loads will experience significantly fewer pipeline stalls due to load instructions and increased overall performance.

Design of the early-issue mechanism is particularly challenging since it must eliminate one cycle from the time taken to fetch and decode loads and read their register file operands. Instruction cache predecode is used to speed up fetch and decode, permitting these operations to occur when loads are placed in the instruction cache. A base register cache provides early access to register file operands. Even with this support, however, not all loads can execute with zero latency. The early-issue mechanism introduces new register and memory hazards, and fast address calculation will occasionally fail. To ensure correct program execution, the design must also include mechanisms to detect and recover from failed accesses.

The rest of this chapter explores the pipeline support needed to implement zero-cycle loads and evaluates their effectiveness in reducing the impact of load latency. Section 4.2 proposes two pipeline designs that support zero-cycle loads: an aggressive design for pipelines with a single stage of instruction decode, and a less aggressive design for pipelines with multiple decode stages. Section 4.4 presents results of simulation-based studies of the proposed designs. Program performance is examined in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on

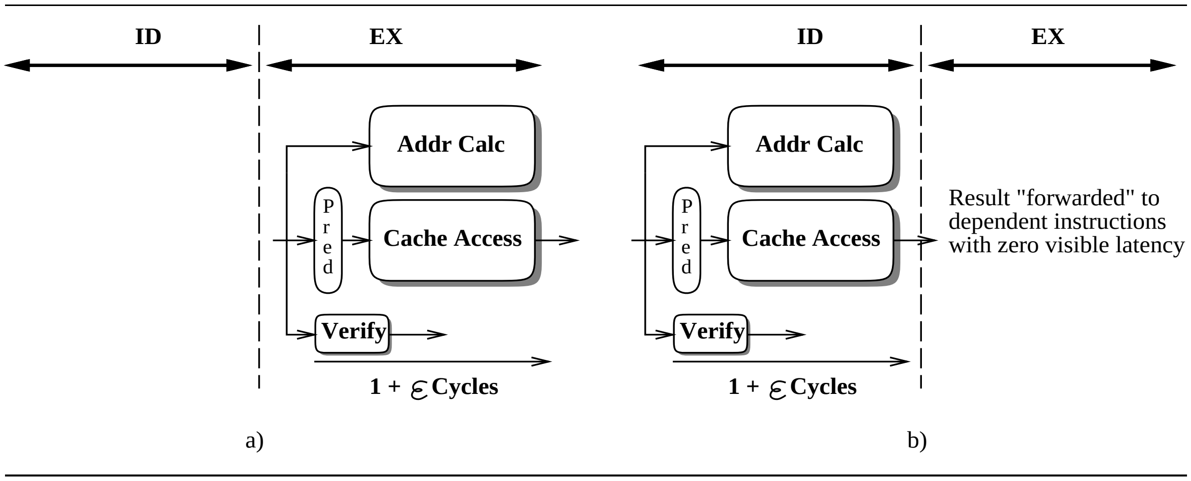


Figure 4.1: Zero-Cycle Loads.

architectures with many and few registers. Section 4.5 describes related work, and finally, Section 4.6 summarizes the chapter.

4.2 Zero-Cycle Loads

4.2.1 Implementation with One Decode Stage

Completing a load by the beginning of the execute stage in a pipeline with only one decode stage is a very challenging task – the load instruction must be fetched, decoded, and executed (including memory access) in only two pipeline stages. Assuming data cache access takes one cycle, all preceeding operations must complete in only a single cycle. Figure 4.2 shows one approach to implementing zero-cycle loads in an in-order issue pipeline with a single decode stage.

Fetch Stage Organization

In the fetch stage of the processor, the instruction cache and *base register and index cache* (or BRIC) are accessed in parallel with the address of the current PC.

The instruction cache returns both instructions and predecode information. The predecode information is generated at instruction cache misses and describes the loads contained in the fetched instructions. The predecode data is supplied directly to the pipes which execute loads, permitting the tasks of fetching, identifying, and aligning loads to complete by the end of the fetch stage.

The predecode data for each load consists of three fields: the addressing mode, base register type, and offset. The addressing mode field specifies either a `register+constant` or `register+register` addressing (if supported in the ISA). The base register type is one of the following: SP, GP, or Other. SP and GP loads use the stack or global pointer [CCH$^+$87] as a base register, respectively. Loads

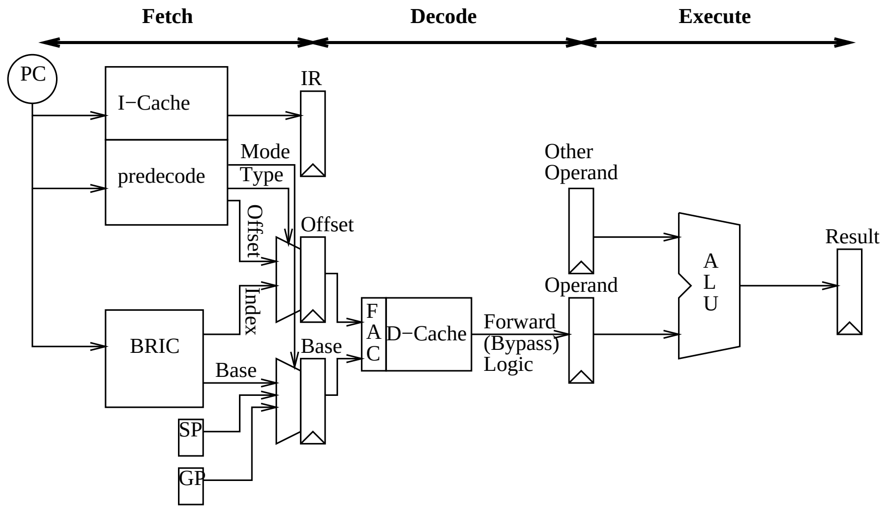


Figure 4.2: Pipelined Implementation with One Decode Stage.

marked "Other" use a register other than the stack or global pointer as a base register. The offset field specifies the offset of the load if it is contained as an immediate value in the instruction.

Predecode costs will vary depending on the specifics of the instruction set, and pipeline and cache designs. For the pipeline shown in Figure 4.2, the predecode information for a single load is 19 bits (assuming an offset of 16 bits). The number of predecode packets attached to an instruction in the data cache depends on the specifics of the pipeline and cache designs. If the necessary network is available to collect predecode from all instructions fetched, only one packet of predecode information is required per instruction. This design would have an instruction cache data array overhead of about 60%, although total cache overheads would be lower because the tag array size would not increase. Without the capability to collect predecode from fetched instructions, predecode for all subsequent loads (that can issue together) must be attached to each instruction in the cache. For a pipeline that can issue two loads per cycle, the predecode overhead would double to 120%. With more time to decode instructions and with more sophisticated pipeline and cache cache designs, predecode costs will decrease.

The BRIC is a small cache indexed by the address of a load, producing a register pair: the base register value and the index register value (unused if a `register+constant` mode load). During execution, the BRIC builds an association between the address of loads executed and their base and index register values. This address-to-register value association allows register access to complete by the end of the fetch stage of the pipeline. If the BRIC misses, an entry is replaced after the base and index register values have been read from the integer register file.

As shown in Section 3.2, loads that use the stack or global pointers are executed quite frequently. The effective capacity of the BRIC can be increased by using an alternate means to supply these loads with a base register value. As illustrated in Figure 4.2, two registers are used to cache copies of the global and stack pointer values. When an access is made, the type field of the predecode data is used to select the correct base register source.

Any cached register value must be updated whenever the corresponding register file value is updated. Since multiple loads may be using the same base and index registers, a value written into the BRIC may have to be stored into multiple locations. Consequently, the BRIC is a complex memory structure, supporting multiple access ports and multi-cast writes. The BRIC will not have to be very large before its access time impacts processor cycle time, hence, only small BRIC sizes are considered – on the order of 4-64 elements. It is also possible to implement zero-cycle loads without a BRIC, a configuration particularly useful to programs with a high frequency of global and stack pointer accesses.

Decode Stage Organization

In the decode stage of the pipeline, the base register and offset pair produced in the fetch stage are combined using fast address calculation (represented by the box labeled FAC in Figure 4.2), and the data cache is accessed.

The pipeline organization in Figure 4.2 requires two new data paths (per pipe). A path must be added to allow forwarding of register values from the BRIC directly to the data cache. This path does not normally exist on traditional pipeline organizations as all values from the fetch/decode stages of the pipeline will first pass through the execute stage before arriving at the data cache ports. In addition, the data path used to write register values to the register file must also be extended to supply register values to the BRIC. All other data paths used to facilitate zero-cycle loads, *i.e.*, D-cache to ALU, ALU to D-cache, ALU to ALU, and D-cache to D-cache, already exist in traditional pipeline organizations.

Pipeline Control

As with most pipeline optimizations, the brunt of the complexity is placed on the pipeline control circuitry. The following logic equation summarizes the condition under which a zero-cycle load will succeed:

$$\begin{aligned} ZCL\_Valid \;\leftarrow\; & BRIC\_Hit \wedge FAC\_Valid \wedge Port\_Allocated \\ & \wedge\; DCache\_Hit \wedge \overline{Reg\_Interlock} \wedge \overline{Mem\_Interlock} \end{aligned}$$

BRIC_Hit indicates if the load address hit in the BRIC.

FAC_Valid indicates if fast address calculation succeeded. Fast address calculation succeeds when no carries are propagated into or generated in the set index part of the effective address computation. (The verification circuit is shown in the lower portion of Figure 3.3 in Chapter 3.)

Port_Allocated indicates if a data cache port is available for the speculative load. This signal is required because accessing the data cache from multiple pipeline stages provides more points of access to the data cache than ports available, necessitating port allocation on a per cycle basis.

DCache_Hit indicates if the load hit in the data cache.

Reg_Interlock indicates whether a data hazard exists between the base and index register values used by the zero-cycle load and the register results of earlier-issued, unfinished instructions.

Mem_Interlock is analogous to *Reg_Interlock*, but detects hazards through memory. A hazard through memory occurs whenever an earlier-issued, unfinished store conflicts with the zero-cycle load. The approaches proposed in Section 3.3.1 to detect load/store conflicts will work here as well.

If a zero-cycle load is not possible or fails due to a mispredicted effective address, there are a number of options available for recovery. If the BRIC misses, the register values read in the decode stage of the pipeline can be used to re-execute the access using fast address calculation in the execute stage of the pipeline. If successful, the load will complete in one cycle. If fast address calculation fails, a non-speculative effective address can be computed in the execute stage of the pipeline, with subsequent data cache access in the memory stage. Alternatively, if an adder is available for use in the decode stage of the pipeline, non-speculative effective address could be performed on the register values from the BRIC, and the failed access could be re-executed in the execute stage of the processor. If an interlock condition exists, the load must stall until it clears, at which point the access can proceed, possibly employing fast address calculation if the interlock condition clears before address generation completes. In the worse case, the BRIC will miss, forcing re-execution in the execute stage, where fast address calculation will fail, resulting in re-execution in the memory stage of the processor – a worse case latency of two cycles (given that mispredictions can be recovered in the following cycle and there is sufficient data cache bandwidth).

In some designs, it may be possible to detect a failure condition early enough to prevent a speculative access. These pipelines will benefit from less wasted data cache bandwidth. For the one decode stage design, it is assumed a BRIC miss or failure to arbitrate a data cache port are the only conditions that can elide a data cache access. While conservative, this strategy ensures that failure detection logic is both simple and fast, resulting in minimal impact on pipeline control critical paths and processor cycle time.

4.2.2 Implementation with Multiple Decode Stages

The increased complexity of instruction decode created by wide issue and faster clock speeds has forced many recent designs to increase the number of pipeline stages between instruction fetch and execute. (Stages collectively referred to as decode stages.) For example, the DEC 21164 [Gwe94a] has three decode stages and the MIPS R10000 [Gwe94b] has two. Adding more decode stages increases the mis-predicted branch penalty, however, architects have compensated for this penalty by increasing branch prediction accuracy through such means as larger branch target buffers or more effective predictors,

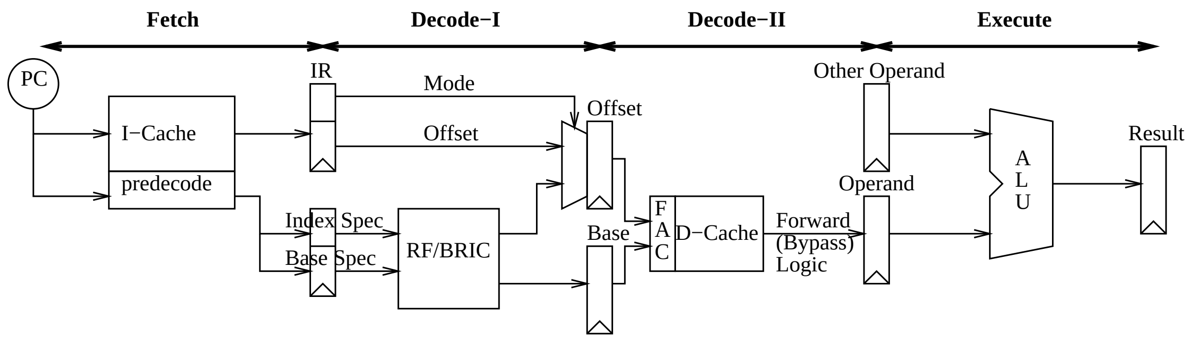


Figure 4.3: Pipelined Implementation with Multiple Decode Stages.

e.g., two-level adaptive [YP93]. Given extra decode stages, the task of implementing zero-cycle loads becomes markedly easier. Figure 4.3 shows one approach to providing support for zero-cycle loads on an in-order issue pipeline with two decode stages.

Register access is delayed to the first decode stage of the pipeline. This modification eliminates the need for a complex address-indexed BRIC in the fetch stage, permitting direct register file access using register specifiers. In the design shown in Figure 4.3, instructions are arbitrarily assumed to be fully decoded by the end of the first decode stage, thus only the base and index register specifiers are supplied by instruction predecode. In some designs, it may be possible to decode the register specifiers and access the register file in a single cycle, eliminating the need for instruction predecode altogether.

In some pipelines, it may not be possible to access the integer register file in the first decode stage without supplying more ports (or multiplexing existing ports), which greatly increases the risk of impacting processor cycle time. A better alternative for these designs may be to adapt the BRIC as a means for caching register values. Previous studies, *e.g.*, [FP91], have found a significant amount of temporal locality in base and index register accesses. A small cache, on the order of 4 to 8 entries should provide the necessary bandwidth to register values without increasing the number of ports on the existing integer register file. Like the original BRIC, a miss initiates a replacement which is available for use after the base and index registers have been read from the integer register file. However, unlike the original BRIC, this storage need not support multi-cast writes, since any register value will reside in at most one location.

The extra decode stage makes it possible to detect more interlock conditions prior to speculative data cache access. For this design, it is assumed register interlock conditions are detected early enough to terminate the speculative access. As a result, a failure in fast address calculation, a memory interlock condition, or a data cache miss are the only signals that do not provide early termination of a speculative access. These conditions cannot be detected early because testing for them requires values that are only available after the start of the data cache access cycle.

By the end of the first decode stage, this design and the single decode stage design converge. In the second decode stage, fast address calculation is used to compute the effective address and the data cache is accessed.

4.2.3 Further Design Considerations

As shown in Section 3.6, fast address calculation fails often when applied to `register+register` mode accesses. Speculation overheads and program performance can likely be improved by not speculating loads using this addressing mode. Instead, accesses using this mode can compute the effective address during the decode stage of the pipeline and access the data cache in the execute stage. Using this design, `register+register` mode accesses that hits in the cache and BRIC and do not have register or memory conflicts will complete in one cycle.

Other failure conditions may manifest due to speculative data cache access. If a fault occurs, *e.g.*, access to an invalid page table entry, it must be masked until the instruction becomes non-speculative. Once the speculative access is verified as correct, posting the fault proceeds as in the non-speculative case.

The fast address calculation mechanism assumes that data cache access can start as soon as the set index part of the effective address is available. If this is not the case, *e.g.*, the cache is indexed with a translated physical address or cache bank selection uses part of the block offset, fast address calculation can not be used. (The early-issue mechanism, however, can still be applied.)

4.3 A Working Example

Figure 4.4 illustrates the performance advantage of zero-cycle loads. Figure 4.4a shows a simple C code fragment which traverses a linked list searching for an element with a matching tag field (often referred to as “pointer chasing”). Figure 4.4b shows the SimpleScalar assembly output as produced by the GNU GCC compiler. This code sequence was selected because it is a very common idiom in C codes, and it is difficult to tolerate the latency of the loads with compile-time scheduling. Moving either load in the loop would require a global scheduling technique because both are preceded by branches. In addition, moving the first load into a previous iteration of the loop would require support for masking faults since a NULL pointer may be dereferenced.

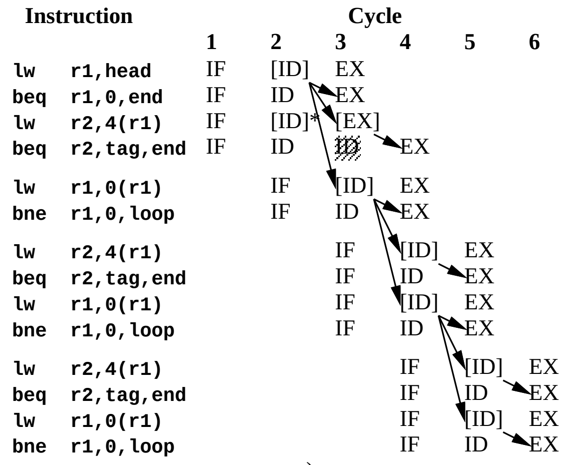
Figure 4.4c and 4.4d depict the code executing on a 4-way in-order issue superscalar processor with and without support for zero-cycle loads, respectively. In both executions, the example assumes perfect branch prediction, one cycle data cache access latency, and unlimited functional unit resources. The stage specifiers contained within brackets, *e.g.*, [ID], denote a data cache access occurred during that cycle. Arrows indicate where values from memory were forwarded to other instructions. The shaded stage specifiers indicate that the instruction was stalled in that stage for one cycle. In the execution with support for zero-cycle loads, all BRIC accesses hit and all fast address calculations

```
for (p = head; p != NULL; p = p->next)
  if (p->tag == tag)
    break;
```

a)

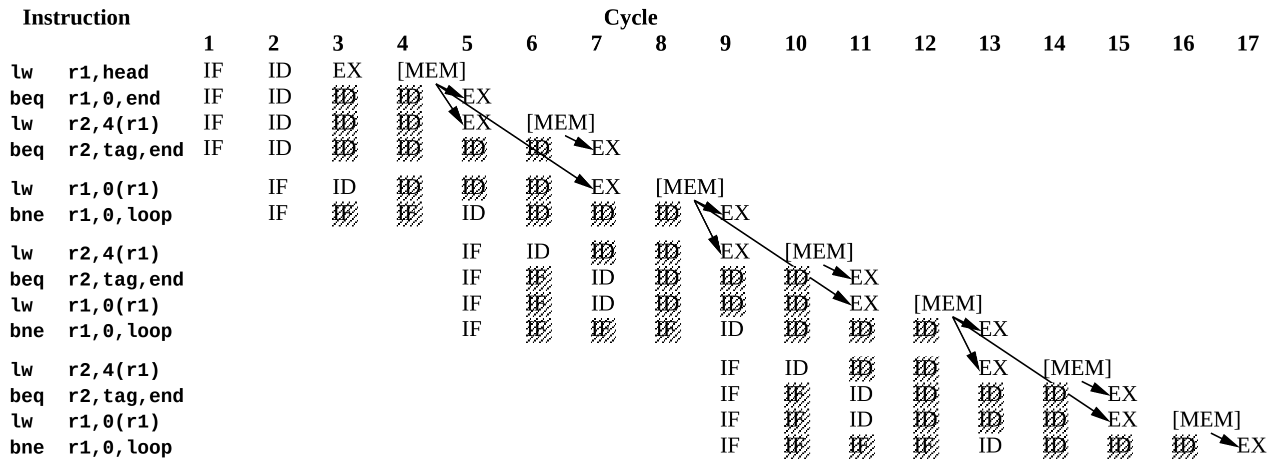
```
        lw   r1,head
        beq  r1,0,end
loop:   lw   r2,4(r1)
        beq  r2,tag,end
        lw   r1,0(r1)
        bne  r1,0,loop
end:
```

b)



Instruction	1	2	3	4	5	6
lw r1,head	IF	[ID]	EX			
beq r1,0,end	IF	ID	EX			
lw r2,4(r1)	IF	[ID]*	[EX]			
beq r2,tag,end	IF	ID	~~ID~~	EX		
lw r1,0(r1)		IF	[ID]	EX		
bne r1,0,loop		IF	ID	EX		
lw r2,4(r1)			IF	[ID]	EX	
beq r2,tag,end			IF	ID	EX	
lw r1,0(r1)			IF	[ID]	EX	
bne r1,0,loop			IF	ID	EX	
lw r2,4(r1)				IF	[ID]	EX
beq r2,tag,end				IF	ID	EX
lw r1,0(r1)				IF	[ID]	EX
bne r1,0,loop				IF	ID	EX

c)



Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw r1,head	IF	ID	EX	[MEM]													
beq r1,0,end	IF	ID	~~ID~~	~~ID~~	EX												
lw r2,4(r1)	IF	ID	~~ID~~	~~ID~~	EX	[MEM]											
beq r2,tag,end	IF	ID	~~ID~~	~~ID~~	~~ID~~	~~ID~~	EX										
lw r1,0(r1)		IF	ID	~~ID~~	~~ID~~	~~ID~~	EX	[MEM]									
bne r1,0,loop		IF	~~IF~~	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX								
lw r2,4(r1)					IF	ID	~~ID~~	~~ID~~	EX	[MEM]							
beq r2,tag,end					IF	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX						
lw r1,0(r1)					IF	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX	[MEM]					
bne r1,0,loop					IF	~~IF~~	~~IF~~	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX				
lw r2,4(r1)									IF	ID	~~ID~~	~~ID~~	EX	[MEM]			
beq r2,tag,end									IF	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX		
lw r1,0(r1)									IF	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX	[MEM]	
bne r1,0,loop									IF	~~IF~~	~~IF~~	~~IF~~	ID	~~ID~~	~~ID~~	~~ID~~	EX

d)

Figure 4.4: Pointer Chasing Example with and without Zero-Cycle Loads. Figure a) shows a simple C code fragment which traverses a linked list searching for an element with a matching tag field. Figure b) shows an assembly version of the code for the SimpleScalar architecture. Figure c) and d) depict the code executing on a 4-way in-order issue superscalar processor with and without support for zero-cycle loads, respectively.

succeed. Incorrectly speculated accesses are denoted with an asterisk.

As seen by comparing the two execution examples, support for zero-cycle loads significantly reduces the number of cycles to execute the code sequence. Without zero-cycle load support, each iteration requires four cycles to execute; with zero-cycle load support, each iteration requires only a single cycle to execute, as both load results are available by the time the two branches reach the execute stage of the processor. Reducing the latency of the load instructions eliminates nearly all stalls. Only one access is misspeculated (marked with an asterisk). This access must be re-executed in the execute stage of the processor because the earlier load issued in the same cycle created a value used by the later load, *i.e.*, a violation of a RAW dependence. (This failure condition would be indicated by the signal *Reg_Interlock*.)

Although not shown in the figure, running the same code on a 4-way out-of-order issue processor without zero-cycle load support requires two cycles to execute each iteration. The out-of-order issue processor cannot achieve one iteration per cycle because the code segment contains a recurrence requiring two cycles per iteration (one cycle for address calculation followed by one cycle to access the data cache). On the same out-of-order issue processor with support for zero-cycle loads, the latency reduction capability of fast address calculation allows each iteration of the recurrence to complete in one cycle.

4.4 Experimental Evaluation

This section examines the performance of programs running on a detailed timing simulator extended to support zero-cycle loads. Key system parameters such as processor issue model, level of software support, and number of architected registers are varied to see what effects these changes had on the efficacy of zero-cycle loads.

4.4.1 Methodology

All experiments were performed on the baseline issue timing simulators detailed in Chapter 2. The simulator implements a detailed timing model of a 4-way superscalar processor and the first level of instruction and data cache memory. The data cache modeled is a dual-ported 16k direct-mapped non-blocking cache. Stores are serviced in two cycles using a 16-entry non-merging store buffer.

A number of modifications were made to the simulator to support zero-cycle loads. To compensate for the cost of generating predecode information, instruction cache miss latency was increased by two cycles. A BRIC was added with a miss latency of three cycles. For the one decode stage designs, the BRIC is address-indexed and the branch penalty is two cycles. For the two decode stage designs, the BRIC is register specified-indexed and the branch penalty is 3 cycles. To reduce the impact of branch penalties for the two decode stage design, the branch target buffer size was doubled. Speculative zero-cycle loads stall until all earlier-issued, unfinished stores are guaranteed to not conflict. If a store's address is unknown, it is assumed to conflict. To reduce the number of stalled loads, stores also employ fast address calculation. The simulator does not attempt fast address calculation on `register+register` mode loads – it instead performs effective address calculation in the decode stage of the pipeline and accesses the data cache in the execute stage. For all experiments, there are only two data cache access ports available each cycle. Data cache ports are arbitrated first to non-speculative loads late in the pipeline, then to the store buffer, and finally, if any are ports are left over, to speculative zero-cycle loads.

All programs were compiled for the SimpleScalar architecture (detailed in Appendix A). The simple software support used to increase the prediction accuracy of fast address calculation (described in Section 3.5) was used for these experiments as well.

4.4.2 Baseline Performance

Table 4.1 shows the impact of zero-cycle loads on program performance. The table lists the IPCs for programs running on pipelines with one and two decode stages and various BRIC configurations. As detailed in Section 4.2, the pipeline implementation with one decode stage (shown in Figure 4.2) indexes the BRIC with a load address and has a 2 cycle branch penalty. The pipeline implementation with two decode stages (shown in Figure 4.3), indexes the BRIC with a register specifier and has a 3 cycle branch penalty. All experiments were performed with the in-order issue processor simulator. The speedups shown are the number of cycles for each program to execute with hardware and software support for zero-cycle loads divided by the number of cycles to execute without hardware and software support for zero-cycle loads. In each experiment, the BRIC simulated is fully associative and uses LRU replacement.

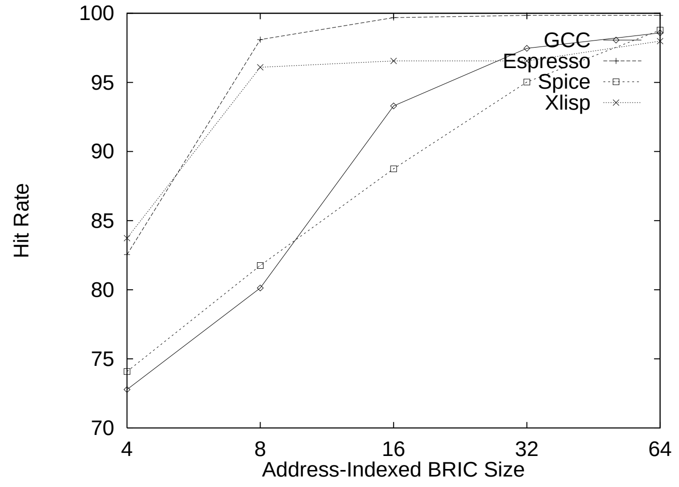
The speedups are quite impressive for both the integer and floating point codes. With an 8 entry address-indexed BRIC (column *BRIC-8*), the run-time weighted average speedups found was 1.45 for the integer codes and 1.26 for the floating point codes. The speedups for the floating point codes are smaller because their executions are heavily dominated by other long latencies, *e.g.*, cache miss latencies and floating point computations, which are mostly unaffected by zero-cycle load support.

Performance with even a small address-indexed BRIC is quite good. The 64 entry BRIC (column *BRIC-64*) only improves performance slightly over the 8 entry BRIC simulations (column *BRIC-8*). This result is very positive, suggesting that keeping the BRIC small to reduce processor cycle time impacts should not have a significant effect on program performance.

Figure 4.5 shows the hit ratios for both address- and register specifier-indexed BRIC of varied size. In each experiment, the BRIC is fully associative with LRU replacement. The figures show only four of the benchmarks, selected as they are representative of the others. The address-indexed BRIC

Benchmark	One Decode Stage Address-Indexed BRIC						Two Decode Stages Register Specifier-Indexed BRIC			
	Base	Speedup					Base	Speedup		
	IPC	BRIC-8	BRIC-64	No GP/SP	GP/SP	Perfect	IPC	BRIC-4	BRIC-8	Perfect
Compress	1.01	1.23	1.23	1.22	1.10	1.40	0.99	1.21	1.22	1.38
Eqntott	1.22	1.41	1.42	1.41	1.03	1.42	1.15	1.37	1.37	1.38
Espresso	1.20	1.43	1.44	1.42	1.03	1.45	1.19	1.40	1.40	1.42
GCC	1.04	1.27	1.28	1.24	1.07	1.41	1.02	1.24	1.25	1.37
SC	1.02	1.48	1.49	1.40	1.16	1.52	1.00	1.45	1.45	1.48
Xlisp	0.99	1.54	1.56	1.48	1.23	1.56	0.98	1.47	1.47	1.48
Grep	0.95	1.95	1.98	1.94	1.01	1.98	0.95	1.96	1.96	1.97
Perl	0.90	1.27	1.28	1.24	1.10	1.33	0.87	1.28	1.28	1.33
YACR-2	1.62	1.37	1.37	1.37	1.12	1.37	1.59	1.36	1.36	1.36
Int Avg	1.11	1.45	1.47	1.42	1.12	1.49	1.08	1.42	1.42	1.44
Alvinn	1.02	1.44	1.44	1.44	1.02	1.45	1.01	1.44	1.44	1.45
Ear	0.84	1.29	1.30	1.29	1.01	1.33	0.81	1.29	1.30	1.34
Mdljdp2	0.83	1.30	1.31	1.29	1.01	1.35	0.82	1.29	1.30	1.35
Mdljsp2	0.78	1.17	1.17	1.17	1.01	1.25	0.78	1.18	1.18	1.25
Spice2g6	0.86	1.28	1.29	1.27	1.20	1.65	0.85	1.27	1.27	1.63
Su2cor	0.76	1.13	1.14	1.13	1.01	1.15	0.75	1.12	1.13	1.15
Tomcatv	0.97	1.21	1.22	1.20	1.02	1.27	0.97	1.21	1.21	1.27
FP Avg	0.87	1.26	1.27	1.26	1.05	1.37	0.86	1.26	1.27	1.37

Table 4.1: Zero-Cycle Load Baseline Performance.



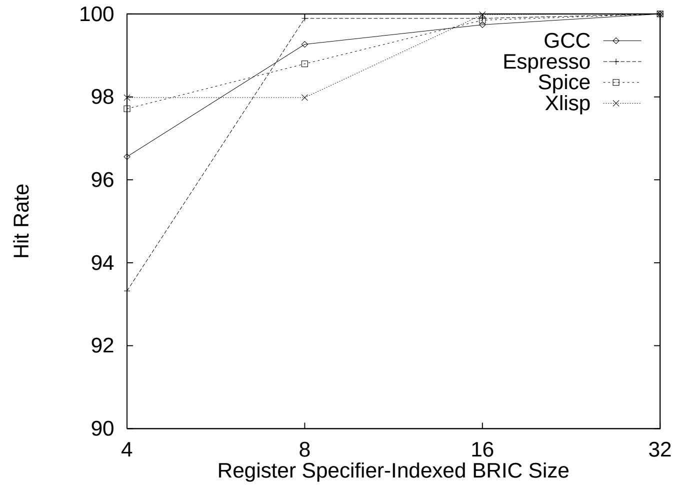


Figure 4.5: Hit Rates for Address- and Register Specifier-Indexed BRIC.

performs well for even small sizes. The performance is quite good for programs with a large number of static loads, *e.g.*, *GCC* and *Spice*, and programs with a large number of dynamic loads, *e.g.*, *Xlisp* and *Spice*. The register specifier-indexed BRIC, used in the two decode stage design, also performed well for all programs. Performance for a even a four entry register specifier-indexed BRIC is very good. This result is to be expected since register file accesses have a significant amount of temporal locality [FP91].

The column labeled *No GP/SP* shows speedups for an 8 entry address-indexed BRIC without separate registers available to cache the global and stack pointer. In this design, global and stack pointer loads reside in the BRIC as well, reducing its effective capacity (*i.e.*, the design in Figure 4.2 without the GP and SP registers). The results suggest that special handling of global and stack references only yields marginal improvements in overall performance, more for the programs with a high dynamic frequency of global and stack loads and stores, *e.g.*, *Xlisp* or *Sc*. If designs omit this option, they may still perform well.

Experiments were also performed to determine the efficacy of the BRIC itself. The column labeled *GP/SP* shows the speedups for a configuration without a BRIC, *i.e.*, only global and stack pointer accesses can execute with zero cycle latency. As expected, only the programs which rely heavily on global or stack pointer accesses show any notable speedups.

The right half of Table 4.1 gives speedups for a pipeline implementation with two decode stages. In this configuration, the branch misprediction penalty is increased to three cycles, and the BRIC is indexed by a register specifier, rather than a load address. Speedups are shown for both a 4 and 8 entry fully associative BRIC with LRU replacement. As suggested by the hit rates in Figure 4.5, the 8 entry BRIC only offers marginal improvement over the 4 entry BRIC. Overall, the speedups are comparable to the one decode stage design. Either of the presented designs should be equally effective at reducing load latency.

Figure 4.6 shows effective latency for loads executing on the two decode stage design with an 8 entry BRIC. Successful zero-cycle loads (labeled *0-cycle*) were most predominant in the integer codes, although for all experiments a significant fraction of loads completed with zero effective latency. Well over 50% of all loads were completed in one or less cycles for all the codes. Two-cycle loads were more common in the floating point codes. When loads could not complete with zero effective latency, the most common causes for failure were register conflicts and fast address calculation failures, with the former case being the most predominant.

Table 4.1 also lists program performance with perfect fast address calculation predictions and no BRIC misses (the columns labeled *Perfect*). The perfect runs provide only slight increases in program performance, more so for the floating point codes. The current designs are exploiting much of the potential performance. Most of the performance loss in each case can be attributed to fast address calculation failures.

4.4.3 Performance with Out-of-Order Issue

An aggressive out-of-order issue execution model provides a built-in mechanism for tolerating load latency. Unconstrained by dependencies, the issue mechanism in an out-of-order issue processor is able to run ahead of executing instructions looking for independent operations with which to tolerate latencies. If branch prediction performs well and there is sufficient parallelism, other latency tolerating or reducing techniques should not be required. To determine the effectiveness of this execution model at negating the benefits of zero-cycle loads, experiments were run on the out-of-order issue simulator (detailed in Chapter 2) extended to support zero-cycle loads. Results of the experiments are shown in Table 4.2. All simulations were performed on a pipeline with two decode stages using an 8 entry fully associative register specifier-indexed BRIC with LRU replacement. All programs are compiled with software support.

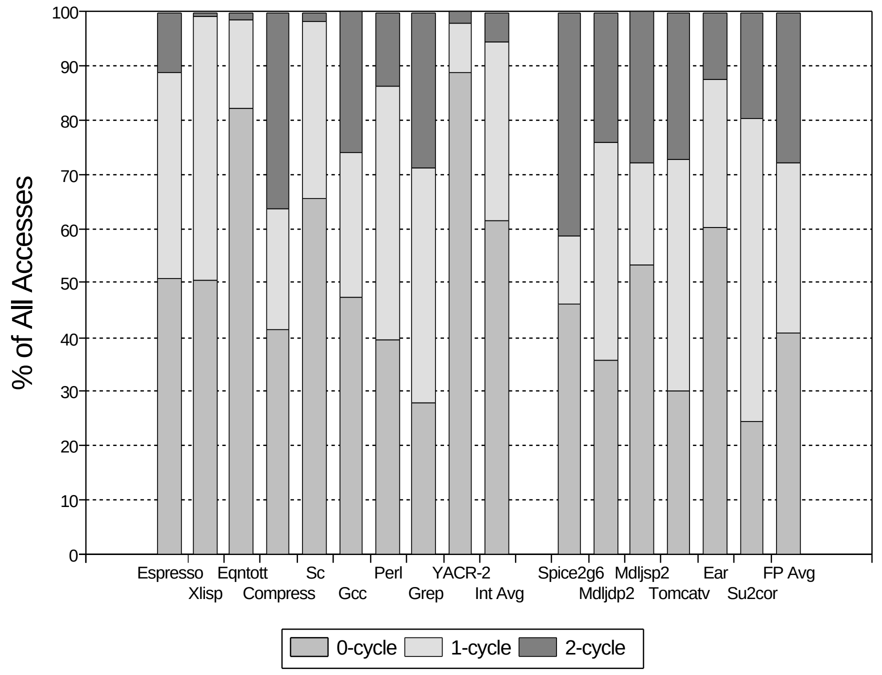


Figure 4.6: Breakdown of All Loads by Latency in Cycles.

Benchmark	Speedup			
	In-Order	Out-of-Order	$\frac{Cycle_{In}}{Cycle_{Out}}$	$\frac{Cycle_{In+ZCL}}{Cycle_{Out}}$
Compress	1.22	1.02	1.63	1.34
Eqntott	1.37	1.16	1.26	0.91
Espresso	1.40	1.15	1.65	1.14
GCC	1.25	1.04	1.46	1.16
SC	1.45	1.16	1.69	1.13
Xlisp	1.47	1.10	1.54	1.03
Grep	1.96	1.42	1.67	0.84
Perl	1.28	1.07	1.40	1.08
YACR-2	1.36	1.07	1.70	1.23
Int Avg	1.42	1.13	1.53	1.06
Alvinn	1.44	1.09	1.80	1.24
Ear	1.30	1.04	2.10	1.61
Mdljdp2	1.30	1.09	1.85	1.42
Mdljsp2	1.18	1.04	1.55	1.47
Spice2g6	1.27	1.01	1.37	1.36
Su2cor	1.13	1.13	1.68	1.40
Tomcatv	1.21	1.08	1.52	1.41
FP Avg	1.27	1.07	1.66	1.39

Table 4.2: Performance with Out-of-Order Issue.

The column labeled *Out-of-Order* in Table 4.1 lists the speedups attained by adding zero-cycle load support to the out-of-order issue processor (over the performance of the out-of-order issue processor without zero-cycle load support). As seen by comparing these speedups with those found for the in-order issue processor (in column *In-Order*, reproduced from Table 4.1), overall speedups are notably less for the processor using out-of-order issue. This result confirms that the out-of-order issue model is tolerating load latency. However, the resulting speedups are not all insignificant, especially for many of the integer codes. These codes likely have less parallelism available to tolerate load instruction latencies, thus they benefit from the latency reduction offered by fast address calculation.

The rightmost two columns of Table 4.2 compare the performance of an in-order issue processor with and without zero-cycle load support to an out-of-order issue processor without zero-cycle load support. The column labeled $Cycle_{In}/Cycle_{Out}$ gives the run-time (in cycles) of programs on the in-order issue processor divided by the run-time on the out-of-order issue processor (neither with zero-cycle load support). This metric quantifies the cycle count advantage of out-of-order issue. Clearly, the programs take fewer cycles to run on the out-of-order issue processor than on the in-order issue processor.

The column labeled $Cycle_{In+ZCL}/Cycle_{Out}$ repeats the experiments, except this time the in-order issue processor has support for zero-cycle loads. For the integer codes, the performance of the two processors is now much closer – both outperforming each other in some cases, with slightly better performance on the out-of-order issue processor. This result is striking when one considers the clock cycle and design time advantages typically afforded to in-order issue processors. It may be the case that for workloads where exposed latency is dominated by data cache access latencies (as is the case for the integer benchmarks), an in-order issue design with support for zero-cycle loads may consistently outperform an out-of-order issue processor. For the floating point codes, the in-order issue processor with support for zero-cycle loads is still much slower than the out-of-order issue processor. The floating point codes have many other long latencies, *e.g.* float point computation and cache misses, which are effectively tolerated by the out-of-order issue mechanism, but benefit little from zero-cycle load support.

4.4.4 Performance with Fewer Registers

A number of architectures in wide-spread use today have few architected registers, *e.g.*, the x86 or IBM System/370 architectures. To evaluate the efficacy of zero-cycle loads for these architectures, the experiments were run again with programs re-compiled to use only 8 integer and 8 floating point registers (one-quarter the normal supply). The results of the experiments are shown in Table 4.3. All simulations were performed on a pipeline using in-order issue with two decode stages. All programs are compiled with software support.

The left side of Table 4.3 shows how a program is affected by reducing the number of architected registers from 32 to 8. The total number of loads increased by as much as 177%, primarily the result

Benchmark	Loads				Speedup		
	Percent	Distribution of Extra Loads					
	More Loads	Global	Stack	General	$\frac{Cycle_{32-reg}}{Cycle_{8-reg}}$	$\frac{Cycle_{32-reg}}{Cycle_{8-reg+GPSP}}$	$\frac{Cycle_{8-reg}}{Cycle_{8-reg+ZCL}}$
Compress	23.34	32.41	44.13	23.45	0.85	1.05	1.32
Eqntott	39.82	92.41	7.59	0.00	0.77	0.86	1.40
Espresso	57.11	9.02	47.90	43.08	0.81	0.96	1.45
GCC	24.02	5.74	36.26	58.01	0.90	1.01	1.28
SC	4.67	0.00	72.46	27.54	0.99	1.13	1.43
Xlisp	0.98	100.00	0.00	0.00	0.99	1.20	1.46
Grep	5.46	0.00	48.86	50.96	0.97	1.01	2.03
Perl	26.46	24.26	32.29	43.45	0.86	1.14	1.38
YACR-2	99.06	0.00	63.14	36.86	0.87	1.23	1.67
Int Avg	26.02	—	—	—	0.90	1.07	1.46
Alvinn	31.79	0.76	99.24	0.00	0.95	1.18	1.34
Ear	112.36	16.45	34.15	49.40	0.78	0.94	1.50
Mdljdp2	81.41	7.66	45.52	46.82	0.69	0.94	1.36
Mdljsp2	110.21	30.35	69.65	0.00	0.77	0.90	1.33
Spice2g6	9.93	13.96	86.04	0.00	0.95	1.16	1.28
Su2cor	47.20	2.26	97.74	0.00	0.79	0.86	1.13
Tomcatv	177.22	1.62	98.38	0.00	0.52	0.56	1.16
FP Avg	64.06	—	—	—	0.82	0.99	1.29

Table 4.3: Performance with Fewer Registers.

of extra accesses needed to spill and reload temporary variables. The increases are notably larger for the floating point codes because they typically use more registers for temporary space. Also shown (in the columns labeled *Distribution of Extra Loads*) is the breakdown of how much (as a percent of total extra loads) each form of addressing contributes to the overhead.

The column labeled $Cycle_{32-reg}/Cycle_{8-reg}$ quantifies the performance impact of fewer architected registers. It shows the run-time (in cycles) of programs compiled to use 32 registers running on the baseline in-order issue simulator divided by the run-time of the 8 register version of the program running on the same processor. For many of the programs, the performance impact of having more registers is quite large, more so in general for the floating point codes.

Next, the performance of 8 register designs with limited and full support for zero-cycle loads was examined. The design with limited zero-cycle load support only allows global and stack pointer accesses to execute with zero cycle latency. Implementing this support is less costly than full support for zero-cycle loads. (This design is essentially the one in Figure 4.2 without a BRIC.) This design should perform well considering the predominance of stack and global accesses in the 8 register executions. The speedups in the column labeled $Cycle_{32-reg}/Cycle_{8-reg+GPSP}$ show this implementation's performance with respect to an architecture with 32 registers and no support for zero-cycle loads. For most programs, limited support for zero-cycle loads more than compensates for the lack of registers in the architecture. In many cases, the performance of the register-limited architecture is better than its 32 register counterpart. Not only does the zero-cycle load support perform well on the extra accesses due to spills and reloads, but it also performs well on the stack and global pointer accesses that both programs execute. For a few of the floating point codes, *e.g.*, *Tomcatv*, the improvements rendered still do not approach the performance of the 32 register architecture. These codes suffer from an excessive number of dynamic loads and stores which saturate available data cache bandwidth and limit overall performance improvements.

The experiments in column $Cycle_{8-reg}/Cycle_{8-reg+ZCL}$ show the performance found with full support for zero-cycle loads on the register-limited architecture. The speedups are shown with respect to the program running on the register-limited architecture without support for zero-cycle loads. As expected, the speedups are better than those found on the 32 register architecture due to the excellent performance of the many extra stack and global accesses.

4.5 Related Work

The application of early issue as a means of reducing load latency has been gainfully applied in a number of previous works [BC91, EV93, GM93]. All use an address predictor mechanism, which is a variant of the *load delta table* [EV93], to generate addresses early in the pipeline, allowing loads to be initiated earlier than the execute stage. The load delta table tracks both the previous address accessed by a particular load and one or more computed stride values used to predict a load's next effective address. Considering the frequency of strided accesses and accesses with no (zero) stride, the load delta table is an effective approach to predicting load addresses. The approach described in this chapter uses a stateless address predictor, eliminating the cost and complexity of the load delta table. Compared to the other approaches, zero-cycle loads features the tightest level of pipeline integration, yielding fewer register and memory interlocks and better potential performance. Tighter pipeline integration, however, does limit the extent to which load latency can be reduced (two cycles in the proposed designs). To have an effect on longer load latencies, like those occurring during data cache misses, a load delta table approach may be more effective if used instead or in conjunction with zero-cycle loads. This observation is further supported by the possibility that the load delta table may perform better on codes where fast address calculation performs poorly (*e.g.*, poorly structured numeric codes).

Instruction scheduling [KE93, PS90, GM86] is essentially the software dual of the early-issue mechanism, as both work to increase distance between loads and their first use. There are tradeoffs to using either approach. With compile-time scheduling, it is possible for the compiler to hoist loads much further than possible with hardware-based early-issue mechanisms. Hardware-based early-issue of loads, on the other hand, benefits from run-time information and the ability to dynamically mask faults, making it possible to hoist loads past stores and branches that would otherwise form barriers for the compiler.

The C Machine [DM82] used a novel approach to implement zero-cycle access to stack frame variables. At cache misses, memory operand specifiers within instructions were replaced with direct stack cache addresses. When the partially decoded instructions were executed, operands in the stack cache could be accessed as quickly as registers.

The *knapsack* memory [AVS93] component included support for zero-cycle loads. Software support was used to place data into the power-of-two aligned knapsack region, providing zero-cycle access to these variables when made with the architecturally-defined knapsack pointer. This optimization was limited primarily to global data.

Jouppi [Jou89] proposed a pipeline that performed ALU operations and memory access in the same stage. The pipeline employs a separate address generation pipeline stage, pushing the execution of ALU instructions and cache access to the same pipeline stage. This organization increases the mispredicted branch penalty by one cycle. It also removes the load-use hazard that occurs in traditional five stage pipelines, instead introducing an address-use hazard. The address-use hazard stalls the pipeline for one cycle if the computation of the base register value is immediately followed by a dependent load or store. The R8000 (TFP) processor [Hsu94] uses a similar approach. Zero-cycle loads can be viewed as essentially a variation on this pipeline design – employing early issue rather than late execute. In the designs presented in this chapter, early issue does not aggravate branch penalty, and fast address calculation allows memory access to execute one stage earlier, thereby eliminating more load latency without incurring any more exposure to hazards.

Comparing the baseline results of this chapter with those in Chapter 3 reveals roughly a two-fold improvement in performance for the integer codes, and nearly a four-fold improvement for the floating point codes. These performance improvements follow from the increased latency reduction afforded by zero-cycle loads (*i.e.*, from one cycle with fast address calculation alone to zero cycles) in combination with better overall support for speeding up `register+register` mode accesses, an addressing mode which the floating point codes rely on more heavily.

4.6 Chapter Summary

Two pipeline designs supporting zero-cycle loads were presented: an aggressive design for pipelines with a single stage of decode and a less aggressive design for pipelines with multiple decode stages. The designs make judicious use of instruction cache predecode, base register caching, and fast address calculation to produce load results two cycles earlier than traditional pipeline designs. The design with multiple decode stages was markedly simpler because more of the component operations of a load could be performed after fetching the load instruction.

The proposed designs were evaluated in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on architectures with many and few registers.

Overall, the speedups afforded by zero-cycle loads, for either pipeline design, were excellent. For the one decode stage design with in-order issue, simulations showed a cycle-weighted speedup of 1.45 for the integer codes and 1.26 for the floating point codes. Good speedups were found for even small BRIC sizes. Software support was generally effective, more so on the integer codes, but even without software support speedups were still quite good.

Speedups on the out-of-order issue processor were significantly less due to the latency tolerating capability of the execution model. However, some programs still showed notable speedups, likely because the executions lacked sufficient parallelism to tolerate all load latency, and thus benefited from the latency reduction capability of fast address calculation. An in-order issue processor with zero-cycle load support compared favorably in performance to an out-of-order issue processor for programs with good cache performance and significant exposed load instruction latency.

With fewer registers, the frequency of loads and their impact of program performance increases significantly, especially for floating point codes. Providing an 8 register architecture with zero-cycle load support for only global and stack pointer references, resulted in performance comparable to a 32 register architecture. This result suggests limited support for zero-cycle loads is one avenue available to improving the performance of legacy architectures with few registers. With full support for zero-cycle loads, speedups were quite good, slightly better than for the 32 register architecture due to excellent prediction performance for the many extra global and stack accesses present.

Chapter 5

High-Bandwidth Address Translation

5.1 Introduction

In support of the proposed cache hit optimizations, this chapter examines four new high-bandwidth and low-latency address translation mechanisms. These new mechanisms feature better latency and area characteristics than current TLB designs, providing architects with effective alternatives for keeping address translation off the critical path of loads.

Address translation is a vital mechanism in modern computer systems. The process provides the operating system with the mapping and protection mechanisms necessary to manage multiple large and private address spaces in a single, limited size physical memory [HP90]. In practice, most microprocessors implement low-latency address translation with a translation lookaside buffer (TLB). A TLB is a cache, typically highly-associative, containing virtual memory page table entries which describe the physical address of a virtual memory page as well as its access permissions and reference status (*i.e.*, reference and dirty bits). The virtual page address of a memory access is used to index the TLB; if the virtual page address hits in the TLB, a translation is quickly returned. On a TLB miss, a hardware- or software-based miss handler is invoked which “walks” the virtual memory page tables to determine the correct translation to load into the TLB.

The latency of address translation is generally not thought of as a latency component of loads. The usual goal of address translation design is not to minimize load latency, but rather to keep address translation off the critical path of data cache access. Typically, this task is accomplished by allowing data cache access and address translation to proceed in parallel using, for example, a virtually-indexed cache. The address translation mechanism is then constructed to be at least as fast as the data cache.

Today, however, architectural and workload trends are placing increased demands on the address translation mechanism, making it increasing difficult to keep address translation off the critical path of data cache access. Processor designs are continually exploiting higher levels of instruction-level parallelism (ILP), which increases the bandwidth demand on TLB designs. The nature of workloads is also changing. There is a strong shift towards codes with large data sets and less locality, resulting in poor TLB hit rates. Notable examples of this trend include environments that support multitasking, threaded programming, and multimedia applications.

Together, architectural and workload trends are pushing architects to look for TLB designs that possess low-latency and high-bandwidth access characteristics while being capable of mapping a large portion of the address space. The current approach used in most multiple-issue processors is a large multi-ported TLB, typically dual-ported with 64-128 entries. A multi-ported TLB provides multiple access paths to all cells of the TLB, allowing multiple translations in a single cycle. The relatively small size of current TLBs along with the layout of the highly-associative storage lends itself well to multi-porting at the cells [WE88].

Although a multi-ported TLB design provides an excellent hit rate at each access port, its latency and area increase sharply as the number of ports or entries is increased. While this design meets the latency and bandwidth requirements of many current designs, continued demands may soon render it impractical, forcing tomorrow's designs to find alternative translation mechanisms. Already, some processor designs have turned to alternative TLB organizations with better latency and bandwidth characteristics; for example, Hal's SPARC64 [Gwe95] and IBM's AS/400 64-bit PowerPC [BHIL94] processor both implement multi-level TLBs. Many processors implement multi-level TLBs for instruction fetch translation as well [CBJ92].

This chapter extends the work in high-bandwidth address translation design by introducing four designs with better latency and area characteristics than a multi-ported TLB. Using detailed timing simulation, the performance of the proposed high-bandwidth designs was compared to the performance of a TLB with unlimited bandwidth. A number of designs are clear winners – their use results in almost no impact on system performance. Any latency and area benefits these designs may afford will serve to improve system performance through increased clock speeds and/or better die space utilization.

The scope of this work is limited to address translation for physically tagged data caches. Instruction fetch translation is a markedly easier problem, since instruction fetch mechanisms typically restrict all instructions fetched in a single cycle to be within the same virtual memory page, requiring at most one translation per cycle. Instruction fetch translation is well served by a single-ported instruction TLB or by a small micro-TLB implemented over a unified instruction and data TLB [CBJ92].

The remainder of this chapter is organized as follows. Section 5.2 describes a performance model for address translation and qualitatively explores the impact of address translation latency and bandwidth on system performance. Section 5.3 details the proposed mechanisms for high-bandwidth address translation, and Section 5.4 presents extensive simulation-based performance studies of address translation designs using the proposed mechanisms. Finally, Section 5.5 summarizes the chapter.

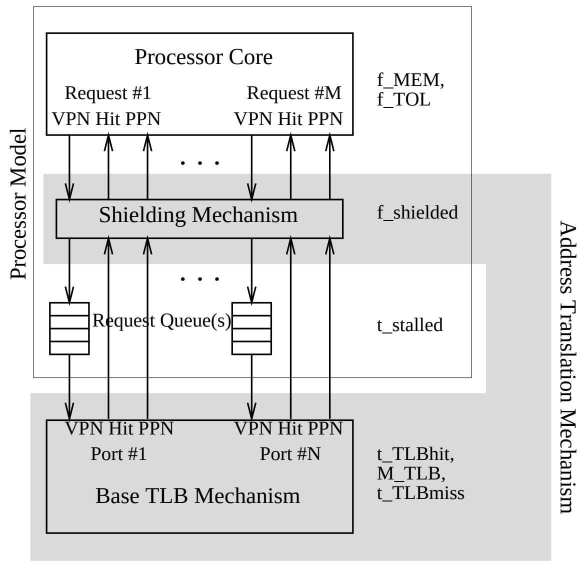


Figure 5.1: A System Model of Address Translation Performance. VPN is the virtual page number, PPN is the physical page number.

5.2 Impact of Address Translation on System Performance

Before delving into the details of the proposed high-bandwidth designs or their evaluation, it is prudent to first develop a performance model for address translation. The model presented in this section is strictly qualitative in nature. It is not used to derive the performance of a particular address translation mechanism; this is done empirically with detailed timing simulations in Section 5.4. Instead, the model serves as a framework for address translation. By casting the proposed designs into this framework, one can readily see which features affect address translation performance, and consequently, how address translation performance affects system performance.

Figure 5.1 illustrates the performance model for address translation. At the highest level, a processor core executes a program in which a fraction f_{MEM} of all instructions access memory. Each cycle, the processor core makes as many as M address translation requests. A fraction $f_{shielded}$ of these requests are serviced by a *shielding mechanism.* A shielding mechanism is a high-bandwidth and low-latency translation device that can satisfy a translation request without: 1) impacting the latency of memory access, or 2) forwarding the request to the base TLB mechanism. Hence, the shielding mechanism acts as a shield for the base TLB mechanism, filtering out $f_{shielded}$ of all translation requests. An effective shielding mechanism can significantly reduce the bandwidth demands on the base TLB mechanism; three shielding mechanisms are examined in detail: *L1 TLBs*, *piggyback ports*, and *pretranslation.*

Requests not handled by the shielding mechanism are directed to the base TLB mechanism which can service up to N requests per translation cycle. The base TLB mechanism functions identically to a traditional TLB, providing fast access to page table entries using a low-latency caching structure. However, the organization used in this work may be non-traditional, *e.g.*, interleaved, for the purpose of providing increased bandwidth. If a base TLB port is immediately available, the translation proceeds immediately. If a port is not available, the request is queued until a port becomes available, at which time it may proceed. The queuing mechanism employed is dependent on the processor model, *e.g.*, an out-of-order issue processor queues requests in a memory reorder buffer, while an in-order issue processor queues requests by stalling the pipeline. Requests are queued waiting for a port for an average latency of $t_{stalled}$. The magnitude of $t_{stalled}$ is determined by the bandwidth of the address translation mechanism – with unlimited bandwidth $t_{stalled}$ will be zero, with limited bandwidth it may be non-zero. How bandwidth affects queueing latency in the processor is very complex, since it depends on the frequency and distribution of requests to the translation device. This relationship is not derived analytically; instead, it is measured precisely with detailed timing simulations in Section 5.4. Once a request is serviced by the base TLB mechanism, $(1 - M_{TLB})$ requests will hit in the TLB and be serviced with latency t_{TLBhit}. The remaining M_{TLB} of all requests will miss in the TLB and be serviced with latency $t_{TLBmiss}$.

Under this model of address translation, the average latency of a translation request (as seen by the processor core), t_{AT}, is:

$$t_{AT} = (1 - f_{shielded}) * (t_{stalled} + t_{TLBhit} + M_{TLB} * t_{TLBmiss})$$

The impact of address translation latency on system performance is tempered by two factors: 1) the processor's ability to tolerate latency, and 2) the relative impact of memory access latency compared to other latencies. The impact of address translation on system performance, measured as the average latency for address translation per instruction, TPI_{AT}, is:

$$TPI_{AT} = f_{MEM} * (1 - f_{TOL}) * t_{AT}$$

f_{TOL} is the fraction of address translation latency that is tolerated by the processor core. The workload and processor model both affect the degree to which the processor core can tolerate latency. If the workload exhibits sufficient parallelism and the execution model provides latency tolerating support, the impact of address translation latency on overall performance will decrease. Processor models with high levels of latency tolerating capability include those that support out-of-order issue, non-blocking memory access, and speculative execution.

Finally, f_{MEM} is the dynamic fraction of all instructions that access memory. This factor is affected by the workload, the number of architected registers, and the compiler's ability to effectively utilize registers. Programs that access memory often will need better address translation performance for good system performance.

In summary, the performance of the address translation mechanism is affected: 1) by its ability to shield requests from the base translation mechanism, and 2) by the latency and bandwidth of the base translation device. The impact address translation on system performance is affected: 1) by the processor's ability to tolerate translation latency, and 2) by the relative impact of address translation latency compared to other latencies.

5.3 High-Bandwidth Address Translation

This section presents new mechanisms for high-bandwidth address translation. The proposed designs fall into two categories: designs that extend traditional high-bandwidth memory design to the domain of address translation, and designs crafted specifically for high-bandwidth address translation.

Techniques for delivering high-bandwidth memory access are well developed, both in the literature and in practice. The common approaches are multi-ported [SF91], interleaved [Rau91], and multi-level [JW94] memory structures. Multi-ported TLBs are already widely used; this work develops and evaluates *interleaved* and *multi-level* TLBs as well. In addition, *piggyback ports* are introduced as a technique to exploit the high level of spatial locality in simultaneous translation requests. This approach allows simultaneous accesses to the same virtual memory page to combine their requests at the TLB access port. *Pretranslation* is introduced as a technique for attaching translations to base register values, making it possible to reuse a single translation many times.

All of the proposed high-bandwidth address translation designs are targeted towards systems with physically tagged caches, *i.e.*, those which require a translation for each memory access. Virtual-address caches, on the other hand, do not require a translation for each memory access. Because there are no physical addresses in the cache, address translation can be pushed off until data is fetched from physical storage, *e.g.*, when a physically addressed second-level cache or main memory is accessed. Such a design eliminates both bandwidth and latency concerns. Virtual-address caches have, however, two significant drawbacks which discourage their use in real systems: 1) synonyms, and 2) lack of support for protection.

Synonyms can occur in virtually-indexed caches when storage is manipulated under multiple virtual addresses. In a multiprogrammed environment, shared physical storage can end up in multiple lines of a virtually-indexed cache, creating a potential coherence problem. In a multiprocessing environment, cache coherence operations must first be reverse-translated to remote virtual addresses before data can be located in the remote cache. Many solutions have been devised to eliminate synonyms, including alignment restrictions on shared data [Che87], selective invalidation [WBL89], and single address space operating systems [KCE92]. However, these approaches have yet to come into widespread use due to performance and/or implementation impacts on application and system software. Moreover, these solutions do not solve the second problem that arises with virtual-address caches, efficient implementation of protection.

Traditionally, protection information has been logically attached to virtual memory pages. As a result, their implementation has been naturally integrated into the TLB. If the TLB is eliminated through use of a virtual-address cache, the problem of implementing protection still remains. One solution is to integrate protection information into cache blocks [WEG$^+$86]. However, the page-granularity of protection information makes managing these fields both complicated and expensive. Another solution is to implement a TLB minus the physical page address information, called a protection lookaside buffer (PLB) [KCE92]. This TLB-like structure, however, still requires high-bandwidth and low-latency access (although, latency requirements are somewhat relaxed).

In light of these drawbacks, virtual-address caches have seen little use in real systems. In addition, it is likely that if virtual-address caches are adopted they may still employ TLB-like structures to implement protection, which requires a high-bandwidth mechanism like the ones describe here. Consequently, virtual-address caches are not considered further.

5.3.1 Multi-ported TLB

A multi-ported TLB, shown in Figure 5.2a, provides high-bandwidth access using a brute force approach. Each port is provided its own data path to every entry in the TLB, implemented by either replicating the entire TLB structure (one single-ported TLB for each port) or multi-porting the individual TLB cells. Since every entry of the TLB is accessible from each port of the device, this design provides a good hit rate for each port (low M_{TLB}). However, the capacitance and resistance load on each access path increases as the number of ports or entries is increased [WE88], resulting in longer access latency (t_{TLBhit}). In addition, this design has a large area due to the many wires and comparators needed to implement each port. (In CMOS technology, the area of a multi-ported device is proportional to the square of the number of ports [Jol91].)

Independent of access latency and implementation area considerations, this design provides the best bandwidth and hit rate of all the designs, hence, it provides a convenient standard for gauging the performance of the other proposed designs.

5.3.2 Interleaved TLB

An interleaved TLB, shown in Figure 5.2b, employs an interconnect to distribute the address stream among multiple TLB banks. Each TLB bank can independently service one request per translation cycle. This design provides high-bandwidth access as long as simultaneous accesses map to different banks.

The mapping between virtual page addresses and the TLB banks is defined by the *bank selection function.* This function influences the distribution of the accesses to the banks, and hence, the bandwidth delivered by the device. The experimental evaluations consider both *bit selection*, which uses a

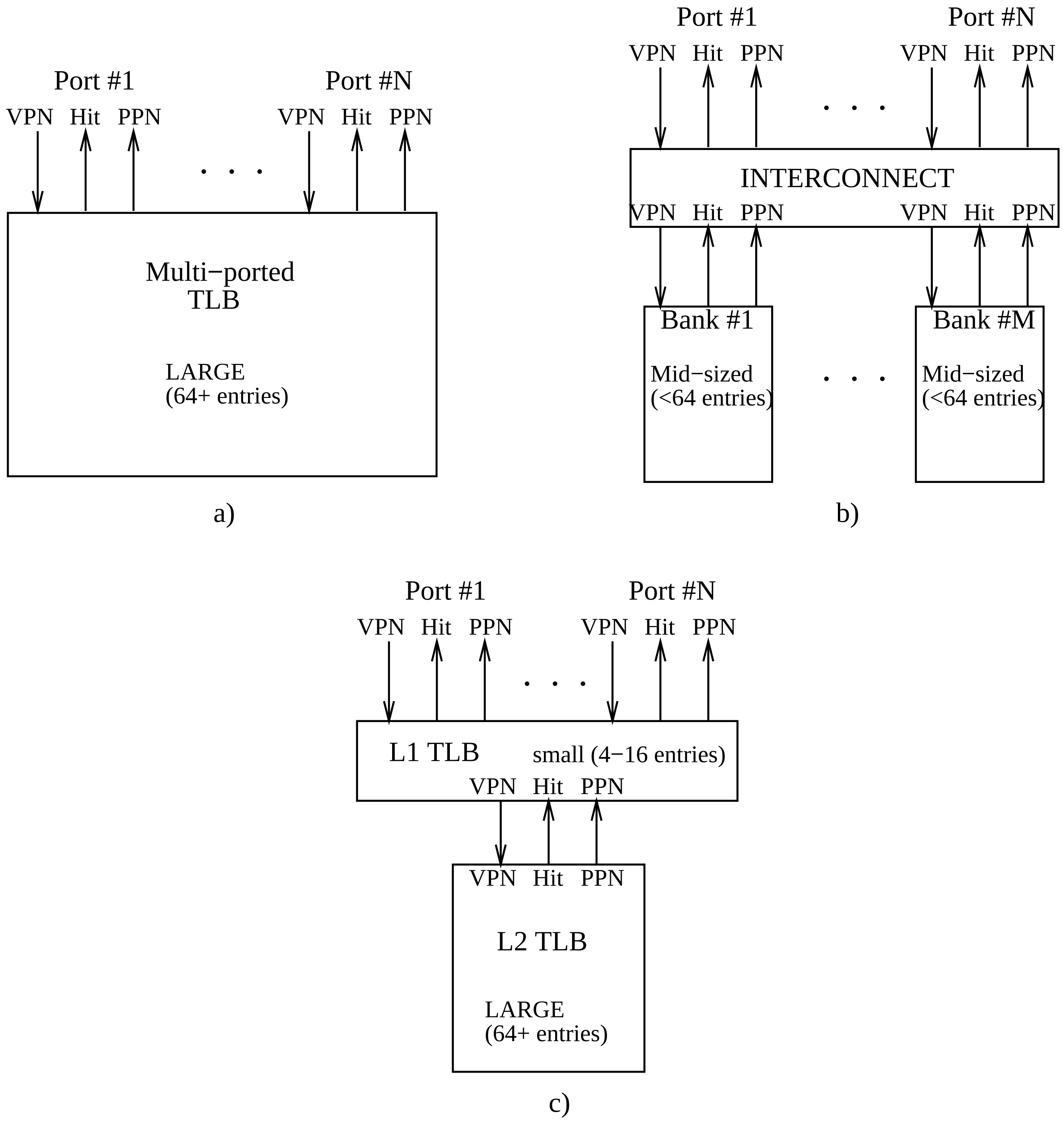


Figure 5.2: Traditional High-Bandwidth Memory Designs: a) multi-ported, b) interleaved, and c) multi-level.

portion of the virtual page address to select the bank, as well as an XOR-folding scheme, which randomizes the bank assignment by XOR'ing together portions of the virtual page address. (XOR-folding functions have been shown to provide better bank distribution [KJLH89].)

By its construction, an interleaved TLB cannot be fully-associative, since any particular page can reside in only one bank. It must have at least as many sets as banks. As a result, M_{TLB} for this design may be higher than a same-size design with a more associative organization, possibly resulting in longer average translation latency. The impact should be minimal, however, if the interleaved TLB remains highly-associative.

This design will likely have better latency and area characteristics than a multi-ported TLB, especially for large TLBs. While the interconnect, typically a full crossbar, adds some latency to the access path, this latency is mitigated by the shorter access latency of the smaller, single-ported banks. The area overhead is concentrated in the interconnect; for a full crossbar, the implementation area is proportional to the square of number of access ports. For small numbers of ports, sizes should not be prohibitively large.

5.3.3 Multi-level TLB

A multi-level TLB, shown in Figure 5.2c, provides high-bandwidth and low-latency address translation by exploiting locality in program references. When an entry from the base TLB mechanism (L2 TLB) is referenced, it is placed into in a small upper-level TLB (L1 TLB). An L1 TLB acts as a shielding mechanism; if it offers a good hit rate, it will shield the L2 TLB from all accesses that hit in the L1 TLB, significantly reducing the bandwidth demand on the L2 TLB.

When an access misses in the L1 TLB, it must forward the request to the L2 TLB, where L2 TLB access port contention, L2 TLB access latency, and L2 TLB miss latency may increase the latency of the access. Since the L1 TLB is small, it may be possible to use a more effective replacement policy (*e.g.*, LRU replacement in the L1 TLB vs. random replacement in the L2 TLB), which should improve the hit rate of the L1 TLB.

If the processor supports hardware-based TLB consistency operations [BRG$^+$89], multi-level inclusion should be enforced in the L1 TLB during L2 TLB replacements or invalidations, *i.e.*, the entries in the L1 TLB should be a subset of the entries in the L2 TLB. This implementation strategy will eliminate the need for consistency operations to probe the L1 TLB, which may be expensive if it is tightly integrated into the processor pipeline.

The L1 TLB is a multi-ported TLB with enough ports to handle all simultaneous requests from the processor core. By keeping the L1 TLB small, it is possible to provide both high-bandwidth and low-latency access to all its entries. The additional area overhead of this design is concentrated in the implementation of the L1 TLB, which for small sizes and few ports should be much smaller than the L2 TLB.

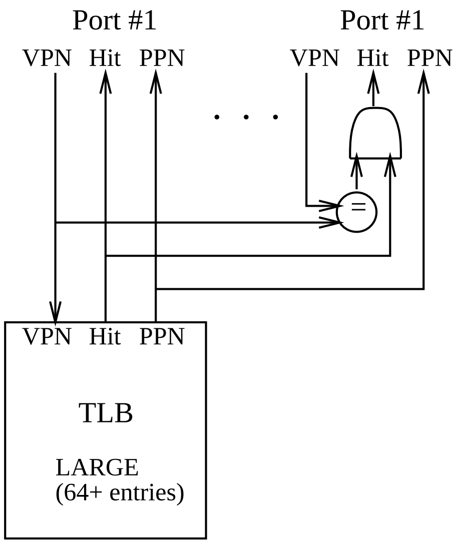


Figure 5.3: Piggyback Port.

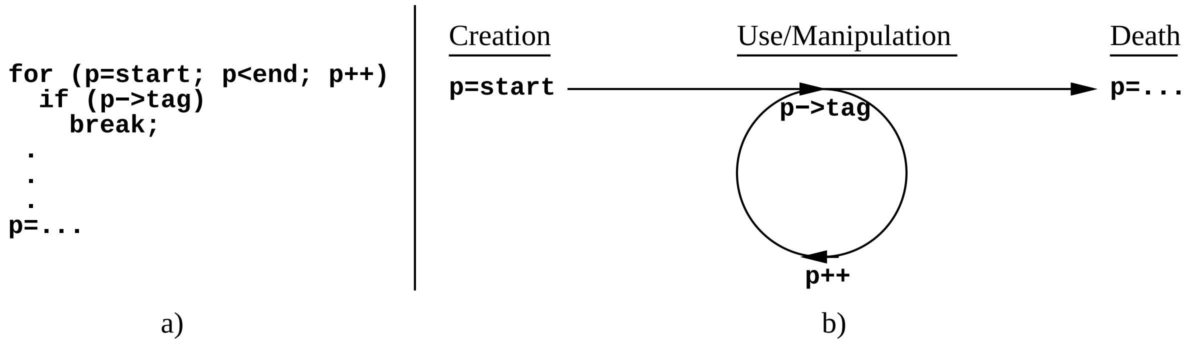
At least two commercial processors have explored the use of multi-level TLBs; Hal's SPARC64 [Gwe95] and IBM's AS/400 64-bit PowerPC [BHIL94] processors both implement multi-level TLBs to meet the latency and bandwidth needs of their respective designs. Multi-level TLB designs have long been used for reducing the latency of instruction fetch translations [CBJ92].

5.3.4 Piggyback Ports

Piggyback ports, shown in Figure 5.3, exploit spatial locality in simultaneous address translation requests. When simultaneous requests arrive at a TLB port, requests with identical virtual page addresses may be satisfied by the same TLB access.

To implement piggybacking, the virtual page addresses of blocked requests are compared to the virtual page addresses of requests in progress. A blocked request may use the result of a translation in progress if their virtual page addresses match. If both requests are executing under the same protection domain, the other fields of the translation request, *i.e.*, protection and page status information, may be shared as well. The approach is similar to read combining in multiprocessor interconnection networks [LS94, GGK$^+$83].

Piggyback ports have minimal impact on translation latency. Once a request is submitted to the TLB, all other requesters can compare their virtual addresses in parallel with TLB access. As a result, the impact on translation latency is limited to a single gate on the TLB hit signal. Area costs are also small, being limited to a single comparator and hit signal gate per piggyback port.



```
for (p=start; p<end; p++)
  if (p->tag)
    break;
 .
 .
 .
p=...
```

Figure 5.4: The Life of a (Register) Pointer. Figure a) shows a C code fragment in which pointer p strides through an array. Figure b) illustrates the operations that occur over the lifetime of pointer p.

5.3.5 Pretranslation

Pretranslation is a shielding mechanism that allows a single translation request to be used for multiple memory accesses. Figure 5.4 illustrates the basis for this approach. Loads and stores access memory through register pointers: global accesses through the global pointer [CCH+87], stack accesses through the stack pointer, and all other references through general purpose register pointers. Pointers are created whenever a variable is referenced, its address is taken, or when dynamic storage is allocated. During the lifetime of a pointer, it is dereferenced at loads and stores, and manipulated using integer arithmetic. Over the lifetime of the pointer, it may be dereferenced and manipulated many times.

Studies have shown, *e.g.*, Eickemeyer and Vassiliadis' work [EV93], that when pointers are manipulated, it is often the case that small constant values are added to or subtracted from the pointer. The end result, which this design exploits, is that successive dereferences of a pointer often yield accesses to the same virtual memory page.

In traditional TLB-based address translation mechanisms, an address translation request is made to the TLB each time a pointer is dereferenced, often requesting the same translation on subsequent requests. With pretranslation, a translation is attached to a register *value* at the first dereference of the value, *i.e.*, at the first load or store to use the register value as a base address. On subsequent dereferences, loads and stores may use the translation (termed a pretranslation) attached to the register value provided that the virtual page address of the memory access matches the virtual page address of the attached translation. When pointers are manipulated with arithmetic operations, any attached translation is propagated to the destination register value. Pretranslation yields high bandwidth as long as register pointers are reused often and point to the same virtual memory page.

Figure 5.5 illustrates how to integrate pretranslation into a processor pipeline. Pretranslations are accessed in parallel with register file access in the decode stage of the pipeline, making the pretranslation available by the start of instruction execution. If the instruction is an arithmetic operation, the pretranslation is attached to the result register value. For loads and stores, the pretranslation, if

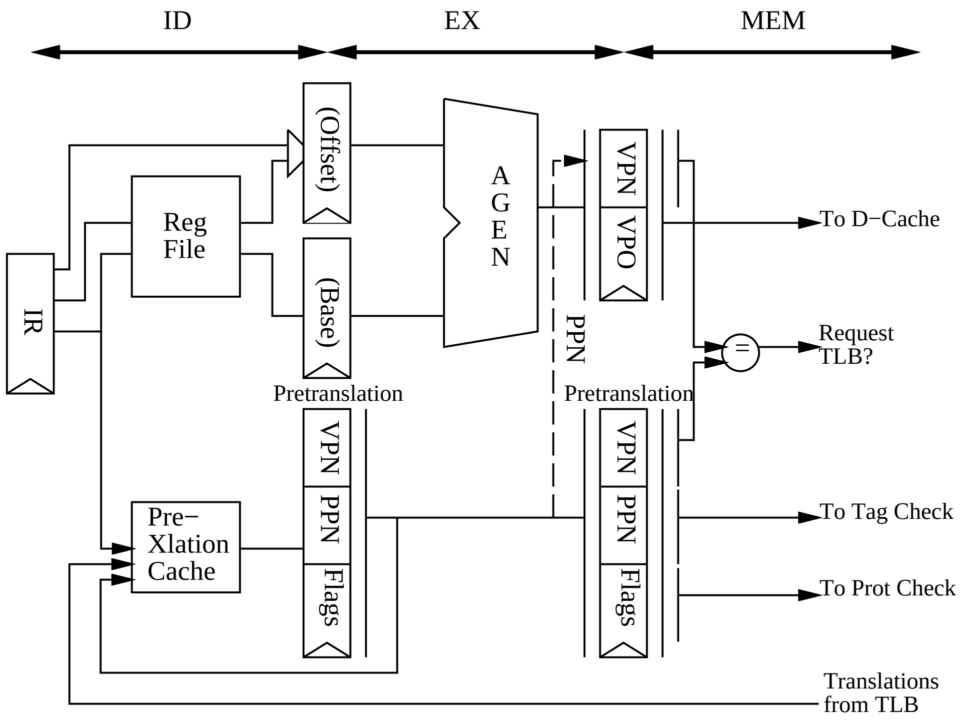


Figure 5.5: Pipeline Support for Pretranslation.

available, is used to elide TLB access if the virtual page addresses match. If the virtual page addresses do not match, a translation request is forwarded to the base translation mechanism. The result of the translation is attached to the base register value.

Two important considerations affect the design of the mechanism used to attach pretranslations to register values. First, a single base register value may be combined with multiple offsets, allowing it to reference multiple pages. A suitable mechanism to attach multiple translations to a single register may improve performance, *e.g.*, a few bits from the offset could be combined with the base register identifier to form the identifier of a pretranslation. Second, only a fraction of all registers will be pointer values at any one time, thus, storage need not be allocated for each register. It suffices to use a small cache, called the *pretranslation cache*, to hold pretranslations. If this cache is kept small, it will facilitate high-bandwidth and low-latency access to pretranslations.

Any changes in virtual memory state, *e.g.*, address mapping, page size, or access permission, must be reflected in the pretranslation cache, otherwise, invalid accesses may go undetected. If virtual memory state changes are infrequent, it may be sufficient to simply flush the pretranslation cache whenever changes occur.

The VAX IPA register used a similar technique to reuse translations during instruction fetch [LE89]. The current PC physical address translation is stored in the Instruction Physical Address (IPA) register, and this translation is used to access the cache until: 1) the PC crosses a page boundary, or 2) a branch is taken. On either of these events, the previous translation is invalidated and another address translation of the PC is initiated. The translation hit buffer (THB) [BF92] further extends this idea to include a prediction of the next translation as well.

Pretranslation can be viewed as an extension of Chiueh and Katz's *branch address cache* (BAC) [CK92], which was applied as a mechanism to reduce access latency of physically indexed caches. (A similar mechanism was proposed in an earlier paper [HHL$^+$90].) The proposed pretranslation design includes a number of modifications to the original BAC mechanism. High-bandwidth address translation is accommodated by attaching the virtual page address to register values. Using this approach, the base TLB mechanism does not have to be accessed to validate use of an attached physical page address. In addition, instructions that create pointer values propagate any attached pretranslation to the result register value. This optimization is important for good performance on optimized code where register copies occur often, for example, during instruction scheduling or loop unrolling. Finally, pretranslations are stored in a small cache, instead of the larger BAC. Since only a fraction of all registers contain pointer values at any one time, the small pretranslation cache provides an excellent hit rate. Like the BAC, pretranslation provides the physical page address by the end of instruction decode. Thus, it may be used to access a physically indexed cache without an added latency for address translation.

5.4 Experimental Evaluation

This section examines the performance of programs running on a detailed timing simulator extended to support high-bandwidth translation designs. Key system parameters such as page size, processor issue model, and number of architected registers are varied to see what effect these changes have on the performance of the designs. All the results presented in this section are run-time weighted averages across all the benchmarks. Individual results for all experiments are included in Appendix B.

5.4.1 Methodology

All experiments were performed on the baseline out-of-order issue timing simulator detailed in Chapter 2. The simulator executes only user-level instructions, performing a detailed timing simulation of an superscalar processor and the first level of instruction and data cache memory. To increase instruction issue rates and accordingly the bandwidth demand on the address translation mechanism, the issue width of the baseline simulator was extended to 8-way issue with a a 64 entry reorder buffer and 32 entry load/store queue. The branch predictor was changed to a 2-level adaptive predictor with an 8 bit global history pattern indexing a 4096 entry pattern history table (*i.e.* the GAp method from [YP93]) with 2-bit saturating counters. In addition, a limited variant of the collapsing buffer (described in [CMMP95]) was added to increase fetch bandwidth. The data cache modeled is a four-ported 32k

Mnemonic	Description
T4	4-ported fully-associative TLB, 128 entries, random replacement
T2	2-ported fully-associative TLB, 128 entries, random replacement
T1	1-ported fully-associative TLB, 128 entries, random replacement
I8	8-way bit-select interleaved TLB, 128 entries (16 entry fully-associative bank), random replacement in bank
I4	4-way bit-select interleaved TLB, 128 entries (32 entry fully-associative bank), random replacement in bank
X4	4-way XOR-select interleaved TLB, 128 entries (32 entry fully-associative bank), random replacement in bank
M16	4-ported 16-entry fully-associative L1 TLB w/LRU replacement, 128-entry fully-associative L2 TLB, random replacement
M8	4-ported 8-entry fully-associative L1 TLB w/LRU replacement, 128-entry fully-associative L2 TLB, random replacement
M4	4-ported 4-entry fully-associative L1 TLB w/LRU replacement, 128-entry fully-associative L2 TLB, random replacement
P8	4-ported 8-entry fully-associative pretranslation cache w/LRU replacement, 128-entry fully-associative L2 TLB, random replacement
PB2	2-ported fully-associative TLB w/2 piggyback ports, 128 entries, random replacement
PB1	1-ported fully-associative TLB w/3 piggyback ports, 128 entries, random replacement
I4/PB	4-way bit-select interleaved TLB w/4 piggybacked banks (32 cells/bank), random replacement in bank

Table 5.1: Analyzed Address Translation Designs.

2-way set-associative non-blocking cache.

A number of changes were made to the simulator to support the proposed high-bandwidth address translation mechanisms. Table 5.1 lists the designs examined, with their mnemonic designations.

For all configurations, TLB access is assumed to be fully overlapped with data cache access. Thus, address translation does not create a visible latency unless the translation mechanism cannot immediately service a translation request, *i.e.*, due to insufficient TLB bandwidth or a TLB miss. When multiple requests meet at a single TLB port, the port is allocated first to the earliest issued instruction. For all designs, the base TLB mechanism has 128 entries. TLB misses are serviced in 30 cycles after the processor pipeline has completely drained. The baseline page size is 4k bytes.

The interleaved schemes, *i.e.*, I8 and I4, use bit selection to select the TLB bank; the three or two address bits immediately above the page offset portion of the virtual address are used to select the proper TLB bank. The configuration X4 uses an XOR-folding of the three least significant groups of two address bits immediately above the page offset portion of the virtual address. In the two-level designs, *i.e.*, M16, M8, and M4, the L1 TLB can service up to four hits per cycle. L1 TLB misses are sent in the following cycle to the L2 TLB, where they may queue if other requests are being serviced by the L2 TLB. The minimum latency for an L1 TLB miss is 2 cycles. The pretranslation cache design (P8) has a hit latency of one cycle; misses are not detected until the cycle immediately following address generation, resulting in at least one more cycle latency for access to the single-ported base TLB. Like the multi-level TLB designs, requests to the single-ported base TLB may have to queue waiting for the port. The pretranslation cache tags are composed of the register identifier (5 bits) concatenated with

Program	Insts	Loads	Stores	Inst/Cycle		(Ld+St)/Cycle		Br Pred
	(Mil.)	(Mil.)	(Mil.)	Issue	C'mit	Issue	C'mit	Rate (%)
Compress	62.0	15.8	6.1	3.65	1.96	1.30	0.69	89.7
Doduc	1,375.1	330.4	130.2	2.16	1.76	0.71	0.59	86.6
Espresso	517.5	116.5	32.7	4.48	2.90	1.32	0.84	90.2
GCC	110.6	26.4	16.5	3.56	1.87	1.32	0.72	80.2
Ghostscript	625.2	109.1	53.3	2.76	2.18	0.73	0.55	93.3
MPEG_play	529.6	114.9	47.9	4.10	2.82	1.19	0.87	85.9
Perl	231.5	57.7	37.2	2.85	1.43	1.10	0.57	81.2
TFFT	959.8	136.6	89.4	2.69	1.79	0.62	0.42	79.9
Tomcatv	359.7	90.9	18.3	3.64	2.72	1.00	0.83	86.6
Xlisp	962.7	289.2	171.6	4.17	2.52	1.86	1.21	87.9

Table 5.2: Program Execution Performance. Instruction, load, and store counts include only non-speculative operations. The columns labeled *Issue* and *C'mit* indicate the average number of operations issued and committed per cycle, respectively, on the baseline 8-way out-of-order issue processor simulator.

the upper 4 bits of the offset of a load or zero for any other instruction. In the piggybacked designs, *i.e.*, PB2 and PB1, requests that do not receive a translation port may piggyback off any other translation performed in the same cycle. For the I4/PB configuration, piggyback ports are provided at each bank of the TLB, thus, simultaneous requests that meet at the same bank may be serviced at the same time if their virtual page addresses match.

In the multi-level TLBs and pretranslation design, *i.e.*, M16, M8, M4, and P8, page status information (*i.e.*, reference and dirty bits) is propagated into the upper-level caching structures. However, when a change must be made to the page status (*e.g.*, first reference or write to a page), the change is immediately sent to the base TLB, where the access may be queued if a port is not available immediately. This write-through strategy for page status information simplifies flushing of the upper-level TLB structure, since any status in the upper-level cache structure is fully replicated in the base TLB. Immediately propagating page status changes to the base TLB has little impact on performance, because page status changes require little bandwidth. Multi-level inclusion is enforced in the L1 TLBs, *i.e.*, M16, M8, and M4, by loading TLB misses into both the L1 TLB and the L2 TLB, and by selectively invalidating from the L1 TLB any entry replaced in the L2 TLB. Coherence is enforced in the pretranslation cache by flushing it whenever an entry in the base TLB is replaced.

The selected benchmarks include programs with varied memory system performance, *i.e.*, programs with large and small data sets as well as high and low reference locality. Table 5.2 details the programs analyzed and their execution statistics. The programs are described in Chapter 2. *Ghostscript* and *TFFT* have the largest data sets, roughly 10 and 40 Mbytes, respectively. *Compress*, *MPEG_play*, and *TFFT* have notably little locality in their reference streams; small data caches and TLBs perform very poorly for these three programs.

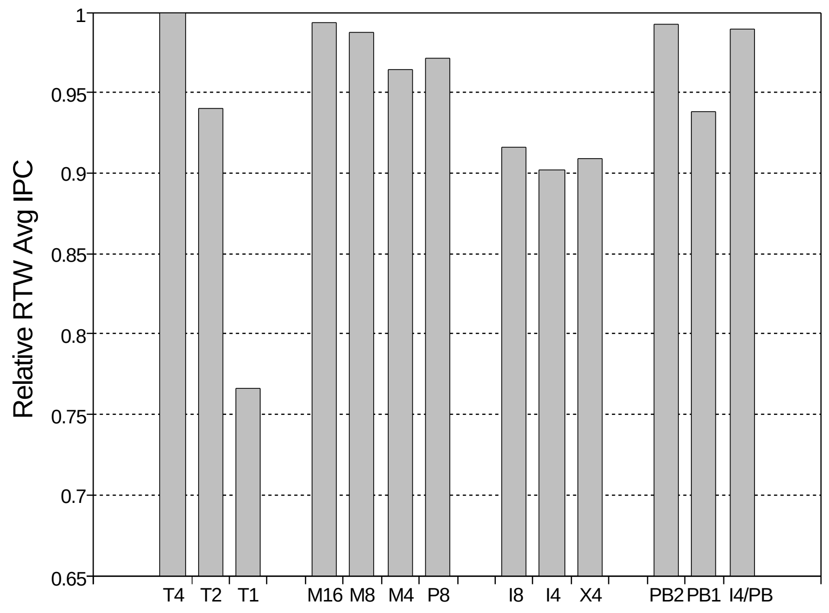


Figure 5.6: Relative Performance on Baseline Simulator. All results are run-time weighted average IPCs normalized to the performance of design T4.

5.4.2 Baseline Performance

Figure 5.6 shows the performance of all the designs running on the baseline processor model, an aggressive 8-way out-of-order issue processor with 32 registers and 4k virtual memory pages. The run-time weighted average IPC (weighted by the run-time of T4 in cycles) is shown for each design. The IPCs are normalized to the IPC of the four-ported TLB design (T4). The T4 design provides a convenient benchmark, since it can service up to four translation requests per cycle, thus no latency is introduced into the results due to insufficient translation bandwidth. (The baseline simulator has a four-ported data cache, so cache bandwidth is never a bottleneck.) Since the timing simulations only count cycles, any clock cycle effects that a poorly scalable design (such as T4) might introduce are ignored. On this common ground, the relative performance of a particular design indicates the cycle time improvement required to make the design worth implementing. For example, the average IPC of the 2-ported TLB design (T2) is 94.1% of the 4-ported design (T4), as a result, for a T2-based design to be a win, the average time per instruction must be at least 0.941 times that of the T4 design.

The leftmost group of bars in Figure 5.6 are the multi-ported TLB designs, *i.e.*, T4, T2, and T1, with 4, 2, and 1 port(s), respectively. These results demonstrate how sensitive the simulated system is to address translation bandwidth. Since the four-ported TLB design (T4) provides all the translation bandwidth the processor needs, its performance is always the best. With half as much translation

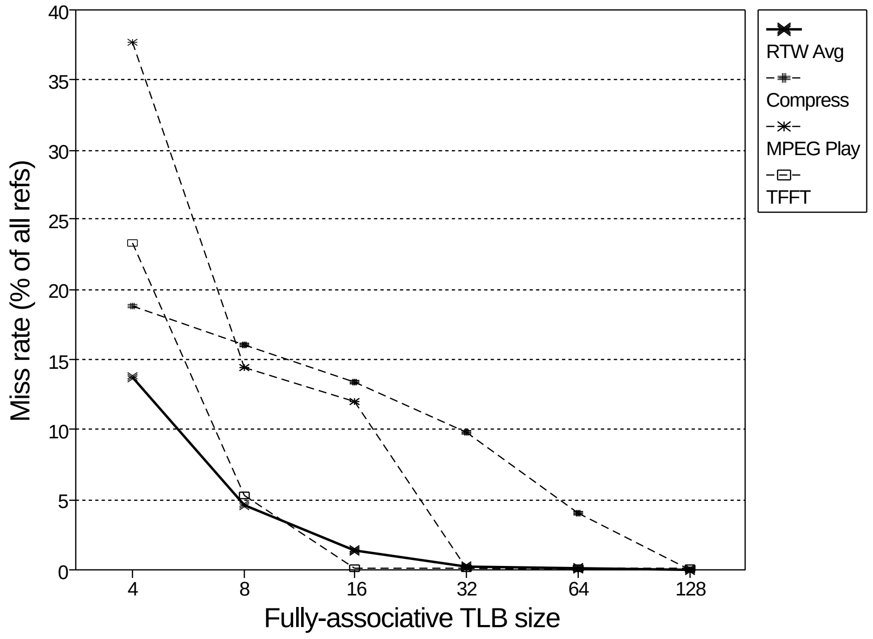


Figure 5.7: TLB Miss Rates. All values shown indicate percent of all references that miss in a fully-associative TLB. The line labeled RTW Avg is the run-time weighted average miss rate over all the benchmarks.

bandwidth, *i.e.*, the dual-ported TLB (T2), the average IPC drops by 6%. With a single-ported TLB (T1), performance drops off sharply to 76% of the performance of the four-ported TLB (T4) design. Clearly, to not impact system performance, a translation device will have to provide at least two translations per cycle.

The second group of bars in Figure 5.6 are the multi-level (*e.g.*, M16, M8, and M4) and pretranslation (P8) designs. The performance of multi-level TLBs is quite good. An L1 TLB with as few as four entries over a single-ported L2 TLB suffers less than a 4% degradation in average IPC. Figure 5.7 indicates why the multi-level designs perform so well. This figure shows the run-time weighted average miss rates (labeled RTW Avg) for fully-associative TLBs from 4 to 128 entries. The 4, 8, and 16 entry TLBs use LRU replacement (as done for the 4, 8, and 16 entry L1 TLBs), while the 32, 64, and 128 entry TLBs employ random replacement (as done for the 128 entry base TLB mechanisms). A four entry L1 TLB with LRU replacement shields all but 13.8% of the translation requests from reaching the L2 TLB. This shielding effect significantly reduces the bandwidth demand on the L2 TLB. The few references that do reach the L2 TLB have only slightly longer latency which is effectively tolerated by the out-of-order issue processor. A few of the programs, most notably *Compress*, *MPEG_play*, and *TFFT*, have poor performance on the multi-level designs. These programs have very low locality in the data reference stream, as can be seen by their large TLB miss rates in Figure 5.7.

While the pretranslation design (P8) performs well, *i.e.*, less than a 3% degradation in average

IPC, its overall performance is worse than a same-sized L1 TLB. The reason for this difference lies in the mechanism by which each design reuses translations. The pretranslation design is only able to reuse a translation whenever a register pointer is reused. The multi-level TLB design, on the other hand, is able to reuse a translation whenever an address is reused. The latter case is more common; when a new register pointer is first used on the pretranslation designs it must be translated, while on the multi-level designs, a new pointer address may already be in the L1 TLB. It is interesting to note that reference locality and register reuse are sometimes orthogonal. In a few specific instances, *e.g.*, *Compress* and *GCC*, the pretranslation designs performed better than a same-sized L1 TLB. This contradictory behavior is likely due to better cache management for the pretranslation design. When new pointer values are created, they are re-inserted into the pretranslation cache, which places the entry on the tail of the LRU queue. Other benefits of the pretranslation cache, such as early presentation of the physical page address should further motivate the use of this design. (The simulations presented do not take advantage of early presentation of the physical page address.)

The interleaved designs did not perform as well as the multi-level designs, providing on the average less bandwidth than a dual-ported TLB (T2). This rather lackluster performance was not due to the set-associative organization required by the interleaved configurations. All of the configurations analyzed were at least 16-way set-associative and possessed excellent hit rates. Poor performance was due to bank conflicts which delayed requests. Increasing the number of banks (I8) or use of an XOR-folding bank selection function (X4) provided only marginal benefit, suggesting that many simultaneous accesses were to the same page, thus no increase in interleaving or change in bank selection function could eliminate conflicts.

The piggybacked designs, *i.e.*, PB2 and PB1, performed better than the interleaved designs. Piggybacking a single-ported TLB (PB1) resulted in only a 6% worse average IPC than the four-ported TLB design (T4). Clearly, many simultaneous accesses are to the same virtual page. However, not all concurrent accesses reference the same page as seen in the improved performance of the piggybacked dual-ported TLB design (PB2). This design can perform two independent translations per cycle, all other requests may use the result of either translation. The piggybacked dual-ported TLB design (PB2) performs nearly as well as the four-ported TLB design (T4).

Design I4/PB is an interleaved TLB with piggyback ports at each bank. This design leverages off the complementary benefits of the interleaved and piggybacked approaches. For an address stream with little spatial locality, requests will be steered to different banks and be serviced in parallel. For an address stream with good spatial locality, requests to the same page will be steered to the same bank and can share the translation result using the piggyback ports. As shown in the Figure 5.6, this design performs very well, resulting in only a 1% degradation in average IPC.

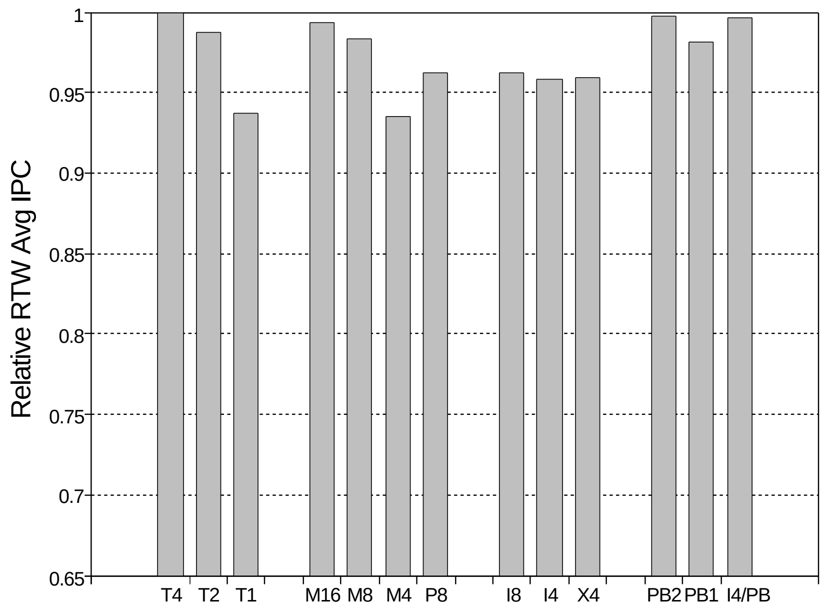


Figure 5.8: Relative Performance with In-order Issue.

5.4.3 Performance with In-Order Issue

Figure 5.8 shows the performance of the designs under the same conditions as Figure 5.6 except the processor is constrained to use an in-order issue model. This modification has two competing effects on the results. First, the average IPC of the in-order issue processor is markedly lower than that of the out-of-order issue processor, *i.e.*, 1.156 vs. 2.094, respectively. Consequently, the bandwidth demand on the address translation mechanism is reduced. Second, the in-order issue processor model cannot tolerate latency as effectively as the out-of-order issue processor. Thus, it is much more sensitive to address translation latency introduced by insufficient bandwidth.

Figure 5.8 shows the results of the experiments running on the 8-way in-order issue processor. The multi-ported TLB designs, *i.e.*, T4, T2, and T1, demonstrate the reduced bandwidth demand on the address translation. With only a single-ported TLB (T1), performance only degrades 6% below performance with a four-ported TLB (T4). The multi-level designs still perform well, although the performance of the 4 entry L1 TLB (M4) was affected more by the in-order issue model than the 8 (M8) and 16 (M16) entry designs. This result is likely due to the reduced latency tolerating capabilities of the in-order issue model, which cannot tolerate the 2 or more cycle latency incurred for the 13.8% of all memory accesses that must be serviced by the L2 TLB. The out-of-order issue model tolerates this latency much better than the in-order model, resulting in better overall performance. The interleaved designs perform much better on the in-order issue model. The degradation in IPC dropped from 10% to about 5% for these experiments. The reduced bandwidth demands on the interleaved designs reduces the number of bank conflicts. The piggybacked designs all perform better, with the PB2 and I4/PB designs experiencing virtually no degradation in average IPC.

5.4.4 Performance with Increased Page Size

A recent trend in TLB design has been to increase page sizes [TH94]. This trend is prompted by workloads with large data sets and/or little locality. Increased page size has a number of effects on the performance of the designs. With the same number of TLB entries, more memory can be mapped, which can reduce the number of TLB misses for both the base and L1 TLBs. Increased page size will increase the lifetime of pretranslations, allowing a pointer to stride further before leaving a page. Larger pages will also affect bank selection in the interleaved TLB designs, address bits formerly used to select the bank will become part of the page offset of the larger page. Changing the bank selection function will affect the distribution of accesses to the TLB banks.

If the system supports variable page sizes [TH94], there are implications on the proposed designs. For the interleaved design, bank selection must be a function of only the virtual page address; as a result, changes in the page size could require a change in the bank selection function. All TLB entries that are no longer in the correct bank would have to be invalidated or moved to the correct bank. Page size changes in the multi-level and pretranslation designs must be kept coherent with the base TLB mechanism. Variable page sizes should not affect piggybacking as long as all requests are in the same virtual address space.

Figure 5.9 shows the performance of the translation mechanisms running on the baseline 8-way out-of-order issue processor, except with 8k pages instead of 4k pages. The performance of the multi-ported designs is mostly unchanged, because the TLB miss rates were unchanged. The miss rates with a 128 entry TLB with 4k pages are already very low. The multi-level and pretranslation designs benefited from the larger page size. The L1 TLBs can map more memory and hence have better hit ratios, while the pretranslation cache benefited from longer pretranslation lifetimes. The interleaved designs performed roughly the same as with 4k pages, although there were some large variations in individual program performance due to changes in the bit selection function. As expected, the larger page size improved the performance of the piggybacked designs, *i.e.*, PB2 and PB1 and I4/PB, since the larger page size provides more opportunity to piggyback requests.

5.4.5 Performance with Fewer Registers

A number of architectures in wide-spread use today have few architected registers, *e.g.*, the Intel x86 or IBM System/370 architectures. To evaluate the efficacy of the proposed translation mechanisms for these architectures, the benchmarks were recompiled to use only 8 integer and 8 floating point registers (one-quarter the normal supply), and the experiments were run again. The primary effect of reducing

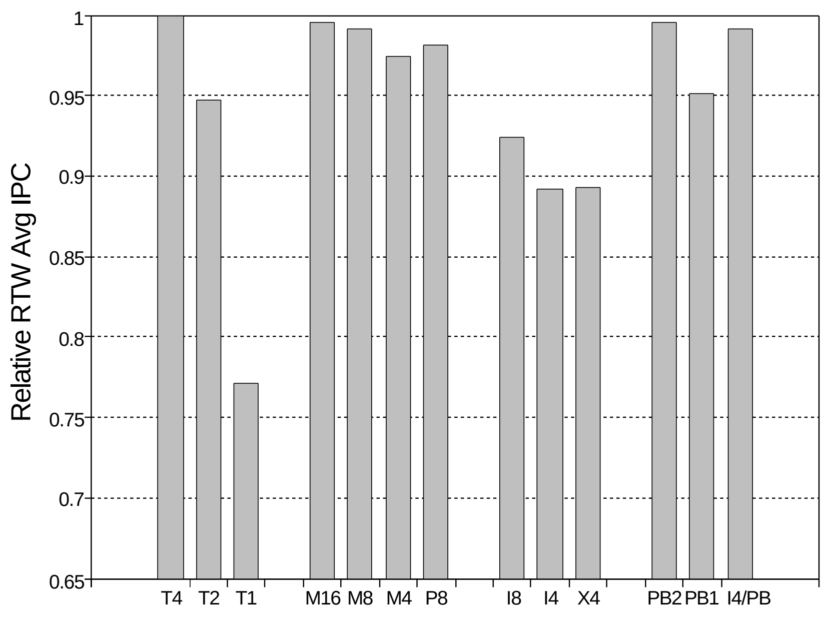


Figure 5.9: Relative Performance with 8k Pages.

the number of registers is an increased number of loads and stores executed. Most of these references are directed to the stack and global regions of memory with a high degree of spatial and temporal locality. The results of the experiments are shown in Figure 5.10. All simulations were performed on the baseline 8-way out-of-order issue processor with 4k pages.

Even with the many extra memory accesses, the multi-level designs performed well. The pretranslation design (P8) performance suffered because (with few registers) pointer register value lifetimes were severely shortened due to many extra spills. When a pointer is spilled to the stack its pretranslation is lost, thus, another translation request must be made to the single-ported base TLB when it is reloaded. The performance of the interleaved designs was impacted significantly, dropping nearly 10% overall. Comparing the multi-level performance to the interleaved designs supports the conclusion that the many extra references have a high degree of locality. However, as shown by the poor performance of the piggybacked single-ported TLB designs (PB1), the locality is not always to the same virtual memory page. The interleaved and piggybacked design (I4/PB) performed slightly worse, suggesting that the extra accesses may have spatial locality spanning a page, which could occur for very large stack frames or many extra accesses directed to a large global region.

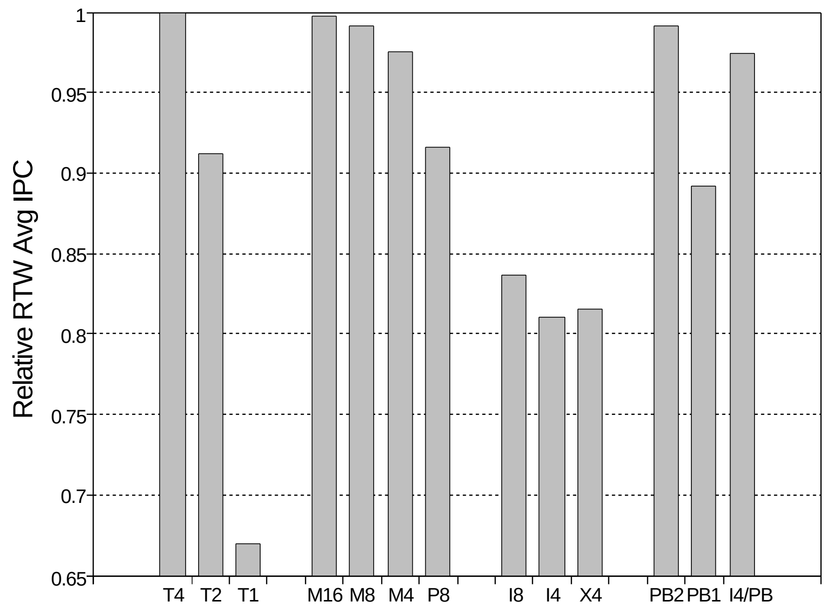


Figure 5.10: Relative Performance with Fewer Registers (8 int/8 fp).

5.5 Chapter Summary

Four new mechanisms for high-bandwidth address translation were presented: interleaved TLBs, multi-level TLBs, piggyback ports, and pretranslation. These address translation mechanisms have better latency and area characteristics than a multi-ported TLB. Various designs using the proposed translation mechanisms were developed and evaluated with detailed timing simulations. Key system parameters such as processor issue model, page size, and number of architected registers were varied to see their effects on the performance of the proposed designs.

Overall, several several designs performed on par with a four-ported TLB. The multi-level TLB designs performed well except for programs with poor reference locality. The interleaved and piggybacked designs complement each other; an interleaved TLB with piggybacking at each bank performed well for all programs. Alone, the interleaved designs performed poorly due to many simultaneous accesses to the same bank, which without support for piggybacking are serialized at the bank. Piggybacking alone also performed poorly over a single-ported TLB due to many accesses occurring simultaneously to different pages. A piggybacked dual-ported TLB appears to be an adequate substitute for a four-ported TLB.

The pretranslation design also performed well, although its performance was slightly worse than a same-sized multi-level TLB design. Other benefits of this design should motivate its use. Pretranslations are available early in the pipeline, facilitating the use of upper-level physically indexed caches.

In addition, attaching address information to physical registers prior to reception of their results could have other benefits, *e.g.*, classifying computation as access or execute.

With in-order issue, bandwidth demand on the translation mechanism is reduced, but it still must perform well to provide good system performance due to the reduced latency tolerating capability of the in-order issue processor. The reduced bandwidth appears to be the stronger force, resulting in better overall performance for all the translation designs.

With larger pages (*i.e.*, 8k vs. 4k), the multi-level, pretranslation, and piggybacked designs performed better. The L1 TLBs can map more memory, resulting in a higher hit rate. Pretranslations performed better because pointers can stride further before their pretranslations are invalidated. With a larger page, piggybacking has more opportunity to combine requests.

With few registers (*i.e.*, 8 int/8 fp vs. 32 int/32 fp), bandwidth demands on the translation mechanism rose sharply. All but the multi-level designs suffered worse performance. The high degree of reference locality in the extra references generated allowed a small L1 TLB to service most of the load. Pretranslation performed worse with fewer registers due to shorter register lifetimes.

Clearly, there exist many effective alternatives to the brute force approach of multi-porting the TLB. The designs presented in this chapter should give architects plenty of choices when multi-ported TLB designs become impractical.

Chapter 6

Cache-Conscious Data Placement

6.1 Introduction

In the previous chapters, techniques were examined for reducing cache hit latencies. Cache hit optimizations provide the most benefit to codes with good cache performance. Many workloads, however, have working sets that are too large or lack the locality necessary for good data cache performance. In other cases, their reference streams perform poorly on commonly used cache geometries. Whatever the case may be, these programs see little benefit from cache hit optimizations since they spend much of their execution time waiting for cache misses to be serviced. For these codes, cache miss optimizations will have a much greater impact on program performance.

Much effort has been invested in reducing the impact of cache misses on program performance. As with any other latency, cache miss latency can be tolerated using compile-time techniques such as instruction scheduling [KE93, PS90, GM86], or run-time techniques including out-of-order issue, decoupled execution [Smi82b], or non-blocking loads [FJ94]. It is also possible to reduce the latency of cache misses using techniques that include multi-level caches [JW94, BKW90, WBL89], victim caches [Jou90], and prefetching [RL92, CBM+92, MLG92]. Reducing the frequency of cache misses also works to reduce the performance impact of cache misses; approaches along these lines include set-associative caches [KJLH89, Hea86], column-associative caches [AP93], stride tolerant address mappings [Gao93, SL93, Sez93], page coloring [Kes91, DS91, BLRC94], and program restructuring to improve data [CMT94, LRW91] or instruction cache performance [Wu92, PH90, McF89].

In this chapter, a novel software-based variable placement optimization, called *cache-conscious data placement*, is introduced as a technique for reducing the frequency of data cache misses. To apply the approach, a program is first profiled to characterize how its variables are used. The profile information then guides heuristic variable placement algorithms in finding a variable placement solution that decreases predicted inter-variable conflict, and increases predicted cache line utilization and block prefetch. The generated placement solution specifies the location of global, stack, and heap variables. Placement of global variables is implemented at compile time using a modified linker, stack and heap variable placement is implemented at run time using modified system libraries, *e.g.*, `malloc()`.

The remainder of this chapter details the design, implementation, and analysis of cache-conscious data placement. Section 6.2 motivates the approach by demonstrating how variable placement can affect data cache performance. Section 6.3 presents an overview of the cache-conscious data placement framework. Section 6.4 describes the implementation of the experimental framework, including details on the profiling methodologies, variable placement heuristics and placement mechanisms. Section 6.5 presents analyses of programs optimized with cache-conscious data placement, and Section 6.6 details related work. Finally, Section 6.7 summarizes the chapter.

6.2 How Variable Placement Affects Data Cache Performance

Variable placement is simply the process of assigning (virtual) addresses to variables. In the context of this work, a variable is any region of memory that the program views as a single contiguous space, *e.g.*, scalars or entire arrays in the global, stack, and heap segments. A variable is assigned an address when it is created. For global variables, addresses are assigned at compile time, typically when the program is linked; for stack and heap variables, addresses are assigned at run time, when the dynamic storage is allocated.

The address assigned to a variable affects its location in the data cache. A variable's address modulo the data cache block size determines its location within a cache block. For a virtually-indexed cache, a variable's address modulo the data cache set size determines the cache set into which the variable will reside.[1] Consequently, variable placement can be used as a mechanism to control which variable's reside in a cache block, and for virtually-indexed caches, which variables map to a cache set.

With variable placement to control the contents of data cache blocks, it becomes possible to influence the performance of the data cache. To see how this is possible, consider how changing a variables placement affects a data cache miss from each of the three miss classes [HS89]:

Conflict Misses: Conflict misses occur when the number of frequently referenced blocks mapping to the same cache set is greater than the associativity of the cache. Blocks that do not fit into the cache set will displace other blocks each time they are referenced. By placing frequently referenced variables into the same cache block or blocks that map to different sets of the cache, inter-variable conflict misses may be eliminated.

Capacity Misses: Capacity misses result when the working set of the program does not fit in the cache. Referenced cache blocks will displace other blocks because there is simply not enough space in the cache to contain all the frequently accessed blocks. By moving infrequently referenced variables out of cache blocks and replacing them with more frequently referenced variables, cache line utilization can be increased. With better utilization of cache lines, the working set of the cache (in cache blocks) may be decreased, and capacity misses may be eliminated.

[1] The address assigned to a variable affects its location in a physically index cache as well, but only to the extent that the cache is indexed with the page offset portion of addresses.

Compulsory Misses: Compulsory misses occur the first time a variable is referenced. If the variable's cache block has not been previously fetched into the cache, a miss will occur. By grouping variables that are used at roughly the same time into the same cache block, cache block prefetches will be used more effectively, and compulsory misses may be eliminated.

Of course, changes in variable placement can either improve or degrade cache performance. The challenge addressed in this work is: How can variable placement be used to consistently improve data cache performance? In the following section, cache-conscious data placement is developed as a variable placement approach that will be shown to consistently improve data cache performance.

6.3 Cache-Conscious Data Placement

Early on in this work, it became apparent that there were a vast number of options available when designing the optimization framework. With little previous work to cull the design space, limits had to be placed on the scope of this work.

First, all variable placement decisions are made at compile time. This restriction made the optimization framework fairly simple to implement and debug. In addition, this approach possesses the lowest variable placement overheads. Global variables are placed by a modified linker with zero run-time cost; stack and heap variables are placed by modified system routines with very low run-time costs. With low run-time overheads, any reduction in the data cache miss ratio afforded by better variable placement is likely to yield program speedups. Compile-time placement, however, has its drawbacks. Since a variable's placement is decided only once at compile time, the approach lacks the ability to adapt to any particular run of a program. This inflexibility may limit the overall impact of the approach. In addition, the approach requires that profile information collected in one run of the program be used to direct variable placement in another run of the program. Heap variables lack a convenient name to implement this binding, necessitating development of a heap naming strategy.

Second, all placement decisions are directed with profile information. Profile information constitutes the best information available to direct placement decisions, thereby eliminating any questions as to the quality of the analyses used to direct variable placement. In the future, studies should consider other possibly more convenient and less expensive methods for characterizing variable usage.

Finally, the designed heuristic placement strategy works to find variable placement solutions optimized for small and low-associativity virtually-indexed data caches. These cache geometries are typical of those found in the first level of the data memory hierarchy of current generation microprocessors. Limiting the study to virtually-indexed caches ensures that the compiler has complete control over the placement of variables in the data cache. Future studies might examine variable placement optimizations for other levels of the memory hierarchy, *e.g.*, L2 caches or main memory. These levels, however, have distinctly different miss frequencies and latencies, indexing strategies, and caching geometries,

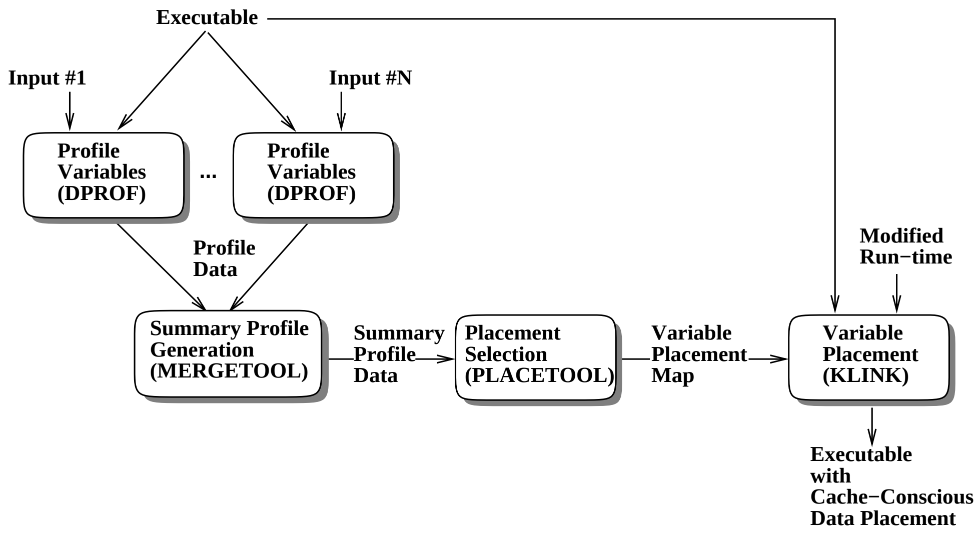

Executable
Input #1
Input #N
Profile Variables (DPROF)
...
Profile Variables (DPROF)
Profile Data
Summary Profile Generation (MERGETOOL)
Summary Profile Data
Placement Selection (PLACETOOL)
Variable Placement Map
Variable Placement (KLINK)
Modified Run−time
Executable with Cache−Conscious Data Placement

Figure 6.1: Cache-Conscious Data Placement.

thus, they will likely benefit from modifications to the proposed placement strategy.

The following subsections give an overview of the cache-conscious data placement framework. The current implementation of the framework is detailed in Section 6.4. The related work section (Section 6.6) includes a chronology of the framework's development.

6.3.1 Optimization Framework

Figure 6.1 illustrates the cache-conscious data placement optimization framework. A program to be optimized is first profiled to gather information characterizing its variable usage. The variable profiler, DPROF, monitors the execution of the program, tracking the location of all variables and any accesses made to them, producing a *reference profile* for each variable. A reference profile indicates how often, when, and where a variable was accessed during an execution of the program. The profiler is run multiple times on representative inputs, and the summary profile generator, MERGETOOL, then aggregates the collected profiles into a summary profile. The placement tool, PLACETOOL, uses the summary profile to guide heuristic variable placement algorithms in finding a variable placement solution that decreases predicted inter-variable cache conflicts, and increases predicted cache line utilization and block prefetch. When the placement tool finds an acceptable placement solution, a placement map is generated. The placement map specifies where each variable should reside in the virtual address space. It is used at compile time by a special linker, KLINK, to place global variables, and at run time by modified system libraries to place stack and heap variables.

The implemented optimization framework can accommodate placement of global, stack, and heap variables. Global variables are placed at an absolute address. Stack and heap variables, on the other hand, lack absolute addresses until run time, thus, their placement is specified as a preferred location in the data cache. Modified system codes enforce their placement at run-time by aligning the allocations to the specified point in the cache.

6.3.2 Heuristic Placement Algorithm

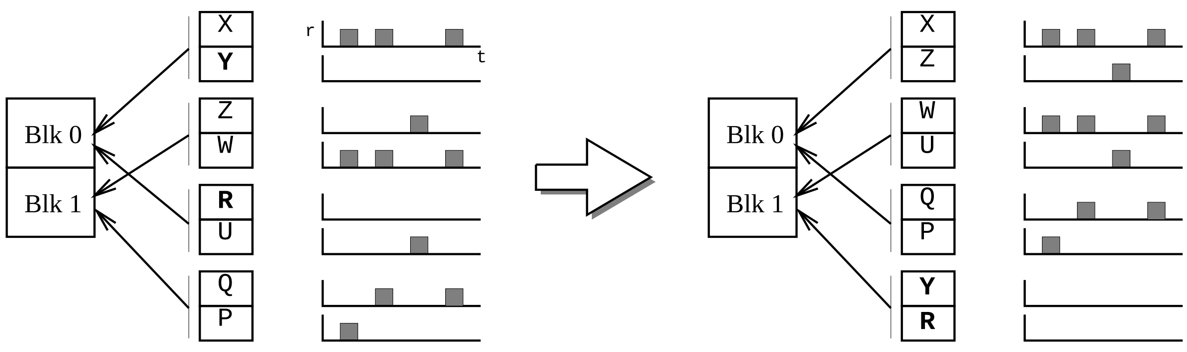
At the heart of cache-conscious data placement is the variable placement algorithm. The algorithm, illustrated with examples in Figures 6.2, 6.3, and 6.4, searches for variable placement solutions that optimize predicted cache performance for a user-specified target cache geometry. This is a challenging task, for even with perfect and complete profile information, finding a variable placement solution with optimal predicted performance is an intractable problem. The search space has at least $N_{var}!$ possible solutions, where N_{var} is the number of variables in the program. As a result, a heuristic search technique is used to locate near optimal solutions.

The heuristic placement algorithm employs three phases of operation, executed in the order shown in the examples. Each of the three phases improves a different aspect of cache performance by: 1) increasing cache line utilization, 2) increasing block prefetch utility, and 3) decreasing inter-variable conflict.

The first phase of operation, *active/inactive variable partitioning*, works to increase cache line utilization by splitting frequently and infrequently referenced variables into separate regions of memory. The partitioned variable placement concentrates more references to a smaller region of memory, resulting in a smaller working set for the program. A frequently referenced, or active, variable is any variable with an average reference density (*i.e.*, references/byte over the entire program execution) larger than a user-specified parameter. The partitioning process is illustrated in the example in Figure 6.2. The two variables `Y` and `R` have no profiled references, thus, they are relocated to a region of memory after the frequently referenced variables. After the transformation, the new variable placement needs only three cache blocks to hold the referenced variables, whereas previously four cache blocks were required.

In the second phase of placement, *temporal affinity set generation*, variables are grouped into *temporal affinity sets*. A temporal affinity set is a collection of variables that share common reference characteristics over time. By grouping these variables such that they are placed in the same cache block, it is possible to increase the effectiveness of cache block prefetch. The grouping process is illustrated in the example in Figure 6.3. The variables pairs `X` and `W`, and `Z` and `U` share common reference patterns over time. As a result, they are placed into the same temporal affinity set (shown by the dark boxes), and the necessary relocations are made to place the variables in the same cache block.

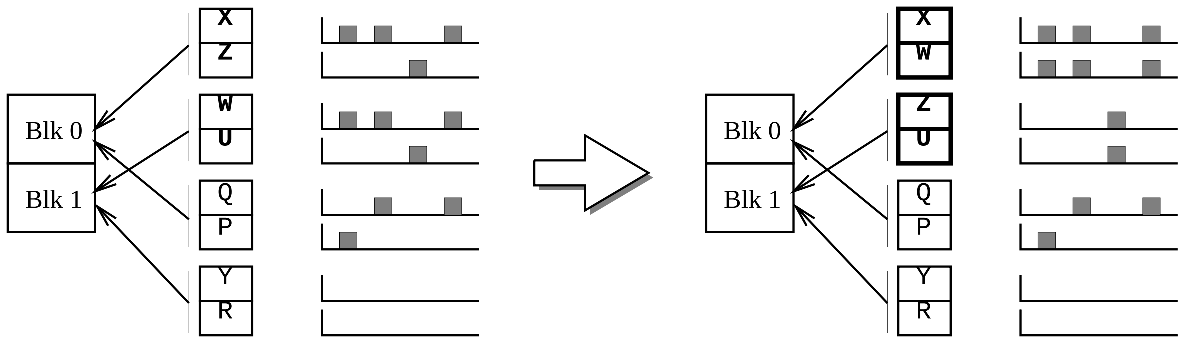
In the final placement phase, *address assignment*, variables are sorted to minimize inter-variable conflicts for the target cache geometry. Since variable placement occurs only once at compile time, the target cache geometry should be selected as the smallest cache size on which the developer expects to attain good (or tolerable) performance for the program being optimized. If the target cache size



Goal: increase cache line utilization

Method: move infrequently used variables out of region of memory used by frequently used variables

Figure 6.2: Active/Inactive Variable Partitioning. The labeled boxes represent variables allocated sequentially in memory. The arrows denote the mapping of variables to the target cache blocks. The target cache block size is large enough to hold two variables. Shown to the right of each variable is its reference profile which indicates total references over time.



Goal: increase block pretech utility

Method: group variables that share similar reference characteristics over time

Figure 6.3: Temporal Affinity Set Generation.

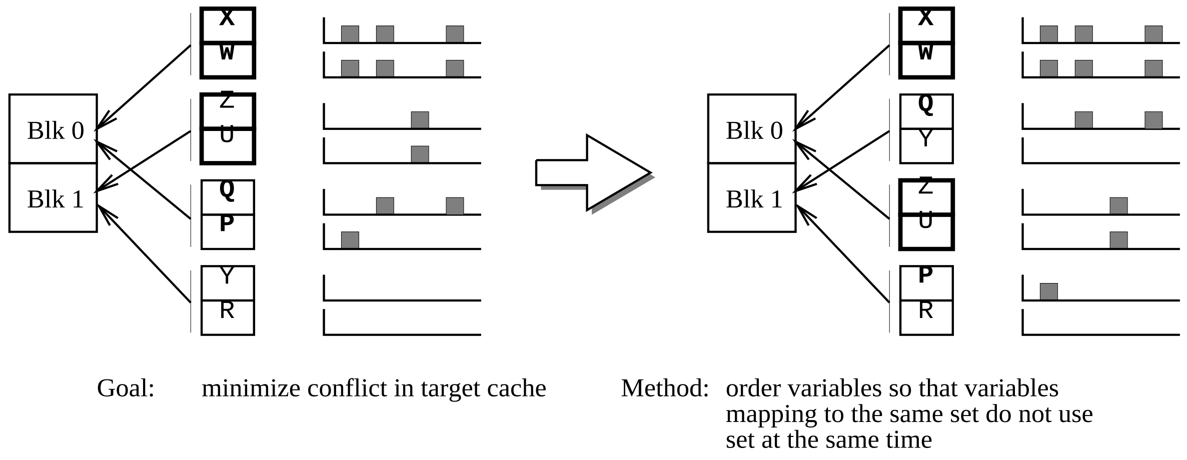

Blk 0
Blk 1
X
W
Z
U
Q
P
Y
R
Blk 0
Blk 1
X
W
Q
Y
Z
U
P
R
Goal: minimize conflict in target cache
Method: order variables so that variables mapping to the same set do not use set at the same time

Figure 6.4: Address Assignment.

selected is too small, the resulting placement will be so strapped by conflict that the placement algorithm will probably not find a good placement solution. If the target cache size selected is too large, the placement algorithm may produce a variable placement solution without considering potentially expensive conflicts in smaller cache sizes.

The conflict between two variables is computed by mapping the variables onto the target cache and computing the fraction of time both variables are resident in the same cache sets at the same time (as predicted by their reference profiles). As shown in the example in Figure 6.4, the profiles for variable group `X/W` and variable `Q` indicate they conflict over two time intervals. The assignment phase attempts to minimize predicted conflict by sorting variables into cache sets containing variables with which they have little predicted conflict. The conflict reduction process is illustrated in the example in Figure 6.4. After detecting the conflict between the variable group `X/W` and `Q`, `Q` and `Y` are swapped with the variable group `Z/U`, thereby eliminating the predicted conflicts.

The last placement transformation demonstrates a phase interaction problem endemic to phasic heuristic search algorithms. In trying eliminate a cache conflict, the assignment algorithm used an inactive variable (*i.e.*, variable `Y`) to fill a hole in the placement. As a result, the capacity reduction afforded by the first placement phase was lost. In addition to phase interactions, phasic heuristic search algorithms also have phase ordering issues. The phases described above could be performed in any order. They are run in the order described because this ordering has to date performed best for the target workload and cache geometries.

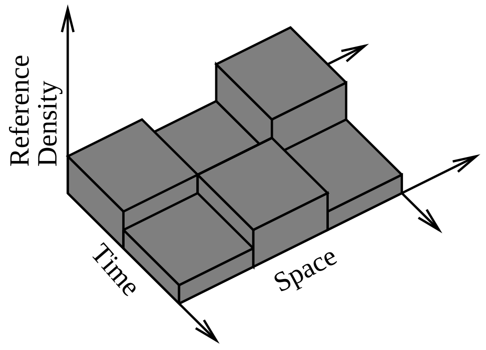


Figure 6.5: Variable Reference Profile.

6.4 Detailed Methodology

This section details the implementation of the cache-conscious data placement framework. The following subsections detail variable profiling, variable placement, and the placement mechanisms used to enforce variable placement decisions.

6.4.1 Variable Profiling

Variable Reference Profiles

Variable reference profiles track how often, when, and where variables are accessed during execution of a program. As shown in Figure 6.5, profiles track reference density over time and space, yielding reference counts for a particular location in a variable over a specific period of time. A convenient implementation for profiles is a two-dimensional array. Since profiler memory is limited, any implementation should make available options to limit the number and size of profile buckets in either dimension, permitting users to locate a suitable balance between profile accuracy and memory overheads.

Variable Profiling

The profiler monitors the execution of the program, tracking all storage management and memory access events. Any profiler implementation will require: 1) a mechanism to monitor program memory accesses, and 2) an efficient technique to bind an accesses to variable reference profiles. Figure 6.6 illustrates the variable profiling methodology implemented in the data profiler, DPROF.

In this work, execution event monitoring is implemented using a modified functional simulator. In the main loop of the simulator, the necessary “hooks” were added to detect storage management and memory accesses. A simulator-based profiler, while simple to extend for event monitoring, results in poor execution performance. In a production environment, performance could be significantly improved using some form of monitoring that supports direct execution of the profiled program. Suitable

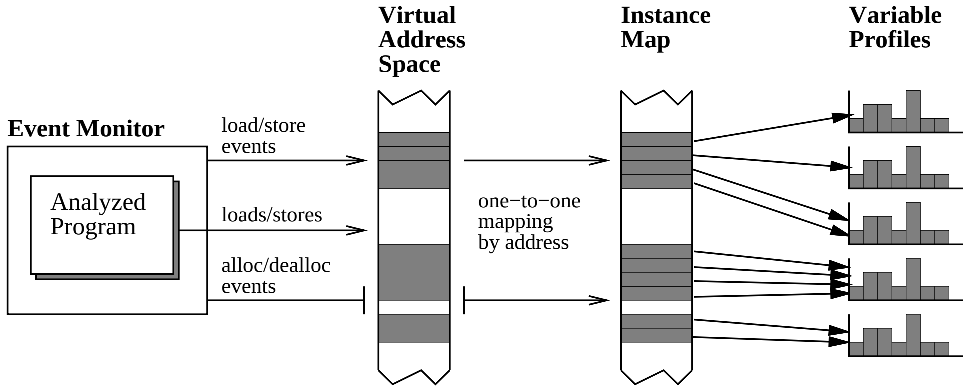


Figure 6.6: Profiling Methodology (DPROF).

approaches along these lines include dynamic compilation [CK93] and executable editing [LS95].

Efficient binding of memory reference addresses to variable profiles is implemented though the use of the *instance map*. The instance map shadows the entire virtual address space, providing a pointer to a profile for each allocated address in the virtual address space. When the profiler detects a memory access event, the *instance map* is indexed with the address of the reference, which returns a pointer to a variable profile that is then updated.

The instance map is updated by the profiler at storage management events. When storage is allocated, the profiler locates the profile associated with the allocated variable (possibly allocating a new profile) and updates all entries in the instance map within the address range of the variable to point to the profile. Instance map entries for global variables are initialized by the profiler at program startup.

To correctly update variable profiles, the profiler needs an indication of elapsed program time. The implemented profiler tracks time using a program cycle counter maintained by the functional simulator. In environments where a cycle counter is not available, a real-time clock or a timer interrupt-incremented counter will suffice.

Variable Naming Strategy

The optimization framework requires that profile information collected in one run of the program be used to direct variable placement in another run of the program. To implement this binding, profile and placement tools must assign names to all variables. Design of the variable naming strategy is an important consideration because it has a profound effect on the quality of profile information and the effectiveness of variable placement. There are many strategies to choose from, *e.g.*, variable address. The one chosen should best meet the following two constraints: 1) variable names should not change between runs of a program, and 2) computing variable names should incur minimal run-time overheads.

In the implemented framework, global variables are named using their address. This approach

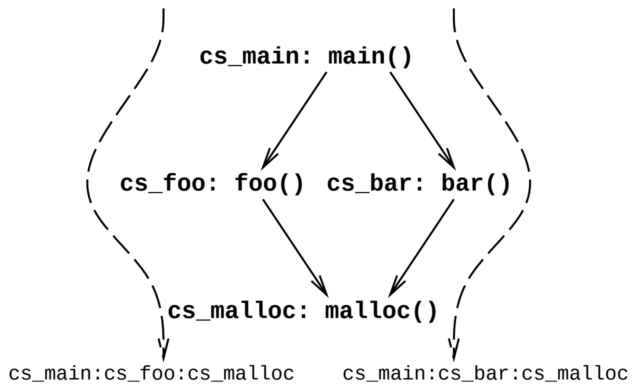


Figure 6.7: Heap Allocation Names. The labels on function calls, *e.g.*, `cs_main`, designate call size addresses. The dashed lines signify function call paths leading to an allocation request at the call to the function `malloc`.

works well to satisfy the above listed constraints. A variable at address `X` in one run of the program is the same variable at address `X` in another run, provided the program is not recompiled between runs. In addition, global variable names can be computed at compile time with no run-time cost. A similar naming strategy is used for naming stack variables. Stack variables are not named individually, instead, the entire stack is assigned a single name, and it is profiled and placed as a single variable. Since most programs have excellent temporal and spatial locality in stack references, this approach has worked well.

Generating names for heap variables is a much more challenging task. Heap variable addresses change between runs of the program, making their address an unsuitable name. The approach implemented in this work is illustrated in Figure 6.7. Heap variables are named when they are created (*e.g.*, at calls to `malloc()`) using the address of the call site to `malloc()` combined (with XOR-folding) with a few return addresses from the stack. (Similar heap naming schemes were employed by Lebeck and Wood [LW94] and Barrett and Zorn [BZ93].) This naming approach does a reasonably good job of satisfying the constraints listed above. Since the addresses of calls sites and function returns do not change between runs of a program (provided the program is not recompiled), heap variable names do not change between runs. Computing heap allocation names is very efficient, requiring only a few instructions. (The code to compute heap names is listed in Figure 6.14.) This approach does, however, have complications that do not arise with global variables. It is possible for concurrently live heap variables to possess the same name. The placement algorithms recognize this possibility and limit the placements on these variables to prevent possibly expensive cache conflicts.

Summary Profile Generation

Collected profile information should be representative of "typical" program variable usage, otherwise the placement algorithm may make poor placement decisions. In this work, this task is accomplished

```
Input:      summary profiles
Output:     placement map

Method:     /* read inputs */
            read_summary_profiles();

            /* PHASE 1: split into active and inactive groups */
            split_active_inactive();

            /* PHASE 2: group global variables with similar temporal characteristics */
            group_globals();

            /* PHASE 3: place stack, global, and heap variables to minimize conflict */
            stack.alignment = 0;
            merge(conflict_map.profile, stack.profile, 0);
            assign_global_addresses();
            assign_heap_alignments();

            /* finished placing variables, write placement map */
            write_placement_map();
```

Figure 6.8: Variable Placement Algorithm (PLACETOOL). Function `merge(A, B, N)` combines profile A and B starting at index N within the space dimension of A and returns the combined profile in A.

through the use of summary profiles. The summary profile generator, MERGETOOL, constructs summary profiles by simply summing together the collected reference profiles for each variable.

6.4.2 Variable Placement Algorithm

Figure 6.8 lists the variable placement algorithm, implemented in PLACETOOL. The placement algorithm reads the summary profiles created during profiling, and then performs variable placement using a three-phase heuristic search algorithm. After a suitable variable placement is found, a variable placement map is created. The following subsections detail the three placement phases.

Active/Inactive Variable Partitioning

In the first phase of placement, the function `split_active_inactive()`, shown in Figure 6.9, performs active/inactive variable partitioning. Variables are split into global and heap sets, and within these groups variables are split into *active* and *inactive* sets. A variable is considered active if it has an average reference density (*i.e.*, total references/bytes) of at least ACTIVE_THRESHOLD. After completion, all inactive variables have been relocated away from the active variables, possibly resulting in better cache line utilization for the active variables.

Temporal Affinity Set Generation

After partitioning active and inactive variables, the active global variables are grouped by the function `group_globals()` into temporal affinity sets. Figure 6.10 shows the algorithm used to generated temporal

```
Procedure:  split_active_inactive()
Input:      all_variables
Output:     active_global, inactive_global, active_heap, inactive_heap

Method:     for var ∈ all_variables {
              if (var.ref_density > ACTIVE_THRESHOLD) {
                if (var.type == GLOBAL)
                  active_globals = active_globals ∪ var;
                else
                  active_heap = active_heap ∪ var;
              }
              else {
                if (var.type == GLOBAL)
                  inactive_globals = inactive_globals ∪ var;
                else
                  inactive_heap = inactive_heap ∪ var;
              }
            }
```

Figure 6.9: Active/Inactive Variable Partitioning.

```
Procedure:  group_globals()
Input:      active_globals
Output:     active_globals, grouped into temporal affinity sets

Method:     for var1 ∈ active_globals {
              for var2 ∈ active_globals {
                if (temporal_affinity(var1, var2) > AFFINITY_THRESHOLD) {
                  active_globals = active_globals - var2;
                  var1.group = var1.group ∪ var2;
                  merge(var1.profile, var2.profile, 0);
                }
              }
            }
```

Figure 6.10: Temporal Affinity Set Generation. Function **temporal_affinity(A, B)** returns the temporal affinity of A to B, defined to be the fraction of execution time in which both variables were active at the same time.

affinity sets. The algorithm selects a global variable and scans all other global variables looking for others with a high temporal affinity. Temporal affinity is computed as the fraction of execution time in which both variables were active at the same time. If two variables have temporal affinity greater than AFFINITY_THRESHOLD, they are placed in the same temporal affinity set. This process continues until all variables have been grouped into temporal affinity sets. By grouping variables with similar reference characteristics into the same same cache block, cache block prefetch utility may increase.

Only global variables are grouped into temporal affinity sets. Heap variables cannot be grouped because the heap allocator used has a minimum allocation size comparable to the target cache block sizes, leaving insufficient space within cache blocks for multiple heap allocations. For target caches or levels of the memory hierarchy with larger blocks, this algorithm should be applied to heap variables as well.

Address Assignment

After all active variables have been grouped into temporal affinity sets, the process of address assignment begins. Address assignment determines the final address for variables while trying to minimize inter-variable conflict. During address assignment, global variables are assigned an absolute address. Stack and heap variables are assigned a preferred alignment in the data cache, which is later enforced at run time by the modified system libraries. As shown in Figure 6.8, the entire stack is placed first, aligned to the first set of the cache. After the stack is placed, global variables are assigned addresses by the function `assign_global_addresses()`, and finally heap variables are assigned cache alignments by the function `assign_heap_alignments()`.

The global variable assignment algorithm is shown in Figure 6.11. The global variables are scanned in descending order of reference density (*i.e.*, most frequently used variable first) for the variable that if placed at the next available address in the data segment will induce the least number of predicted conflict misses. Conflict misses are predicted with the *conflict map*. The conflict map is a reference profile for the target cache, updated after variables are placed to show how many profiled references are being made to a particular cache set at a particular time. With this information, the placement algorithms can predict the conflict induced by any variable placement.

The function `conflict(A, B, C)` computes the conflict between profiles `A` and `B` by first projecting profile `B` onto profile `A` starting at index `C` in the space dimension of profile `A`. The function then computes the product of the reference counts that overlap in both the time and space dimensions.

Using this approach, the estimated conflict will be small if the variable is placed into the cache and the cache in infrequently reference when the variable is frequently referenced. Conversely, when the variable projects onto an frequently referenced portion of the cache, the estimated conflict will be higher.

The conflict map is updated with the reference profiles of placed variables only after an entire pass through the data cache. This strategy ensures that variables from the same image of the cache (in memory) do not conflict with each other during placement. In the assignment algorithm, the variable `pass_var` tracks which variables have been placed but are awaiting integration into the conflict map.

The heap allocation placement algorithm, shown in Figure 6.12, determines the preferred cache alignments for heap allocations. Unlike global variables, heap allocations may be aligned to any set in the cache. Accordingly, the algorithm scans the entire conflict map looking for the point of minimum estimated conflict, and places the heap variable at that point. (If the algorithm determines the variable should not be aligned, it is assigned an alignment of -1.)

While the algorithm computes the preferred alignment for all heap variables, the heap allocation placement algorithm only forces cache alignments (by setting the variable property `force_alignment` to `TRUE`) on heap variables with the following characteristics:

```
Procedure   assign_global_addresses()
Input:      conflict_map, active_globals, inactive_globals
Output:     active_globals, inactive_globals, with all variables assigned an address

Method:     addr = DATA_BASE;
            pass = addr / BASE_CACHE_SIZE;
            for 1 to |active_globals| {
               min_conflict = MAX_INT;
               for var ∈ active_globals in descending ref_density {
                  c = conflict(conflict_map.profile, var.profile, addr % BASE_CACHE_SIZE);
                  if (c < min_conflict) {
                     min_conflict = c;
                     min_var = var;
                  }
               }
               min_var.addr = addr;
               addr = addr + min_var.size;
               active_globals = active_globals - min_var;
               pass_vars = pass_vars ∪ min_var.group;
               if ((addr / BASE_CACHE_SIZE) != pass) {
                  for var ∈ pass_vars {
                     merge(conflict_map, var.profile, var.addr % BASE_CACHE_SIZE);
                     pass_vars = pass_vars - var;
                  }
                  pass = addr / BASE_CACHE_SIZE;
               }
            }

            /* place inactive global variables */
            for var ∈ inactive_globals {
               var.addr = addr;
               addr = addr + var.size;
            }
```

Figure 6.11: Global Variable Address Assignment. DATA_BASE is the base address of the data segment. The function `conflict(A,B,C)` computes the conflict between `A` and `B` starting at index `C` in the space dimension `A`.

- The heap variable's name cannot have more than INSTANCE_THRESHOLD concurrent instances during any profiled execution.
- The heap variable must have a reference density (as indicated in the summary profile) of at least HEAP_THRESHOLD.

The first requirement ensures that alignments are not placed on heap variable names with many concurrent instances, which could induce many intra-variable conflict misses. The profiler tracks the maximum number of concurrent instances created for any variable name. If the value is greater than INSTANCE_THRESHOLD, the allocation is never aligned. The second condition reduces variable alignment memory overhead (incurred during placement of heap variable) by only forcing alignment on variables that are frequently referenced. These variables have the best chance of negating any performance penalties due to increased memory usage incurred when aligning heap allocations. Variables not marked with forced alignment may also be allocated with the correct alignment, however, this will occur only if no extra memory overheads are incurred.

```
Procedure:  assign_heap_alignments()
Input:      conflict_map, active_heap, inactive_heap
Output:     active_heap, inactive_heap, with all variables assigned an alignment

Method:     for var ∈ active_heap in descending ref_density {
              min_conflict = MAX_INT;
              for i = 0 to BASE_CACHE_SIZE-1 {
                c = conflict(conflict_map.profile, var.profile, i);
                if (c < min_conflict) {
                  min_conflict = c;
                  min_index = i;
              }
              if (var.max_instance < INSTANCE_THRESHOLD) {
                if (var.ref_density >= HEAP_THRESHOLD)
                  var.force_alignment = TRUE;
                var.alignment = min_index;
                merge(conflict_map, var.profile, var.alignment);
              }
              else
                var.alignment = -1;
            }

            /* place inactive heap allocations */
            for var ∈ inactive_heap
              var.alignment = -1;
```

Figure 6.12: Heap Variable Address Assignment.

Before completion, both placement algorithms place the inactive variables. The placement algorithm does not attempt to minimize conflicts when placing inactive variables. Since inactive variables have few accesses, they induce few conflict misses irregardless of their placements. To preserve any spatial locality that existed in the original program, inactive global variables are placed in the same order as found in the original program. Inactive heap variables are marked as not requiring a cache alignment.

The order of placement, *i.e.*, first stack, then global, and finally heap, is important for good placement performance. The stack typically has a large area of active use and would be difficult to place effectively after the global or heap variables. Heap allocations are best placed after the global variables because they may be placed anywhere in the cache, whereas global variables must be placed sequentially in the data segment. Placing variable candidates with the most placement freedom last appears to produce better placement solutions.

The running time of the placement algorithms can be very long. The global assignment routine, *i.e.*, `place_global_variables()`, is the most expensive algorithm with a running time $O(G^2ST)$, where G is the number of global variables, S is the set size of the cache, and T is the number of buckets in the time dimension of profiles. To reduce running time, a branch and bound variation of the assignment algorithms was employed in the implementation of the placement algorithms. While searching for variables with minimal conflict, the minimum conflict yet found is tracked and passed to the function that estimates conflict, *i.e.*, `conflict()`. If the current call to `conflict()` computes a conflict value

```
Input:    executable, with symbol table including variable addresses and size,
          reference relocations, and variable placement map
Output:   executable, with relocated data segment and modified system libraries

Method:   /* read inputs */
          read_executable_segments();
          read_symbol_table();
          read_relocation_table();
          read_placement_map();

          /* attach each relocation to variable it references */
          bind_relocations_to_symbols();

          /* do the placement */
          update_variable_placement();

          /* fix up images */
          fixup_relocations();
          rearrange_data_segment();
          replace_startup()
          replace_malloc()
          link_dynamic_map();

          /* write output */
          write_executable_segments();
```

Figure 6.13: Post-pass Link Algorithm (KLINK).

higher than the passed minimum conflict, conflict evaluation is immediately terminated, and the search continues with another variable. This modification significantly improved the running time of the placement algorithm, *e.g.*, placement for GCC dropped from a running time of 20 minutes to less than 30 seconds.

6.4.3 Placement Mechanisms

The placement mechanisms enforce the placement decisions made by the placement tool. For global variables, placement can be enforced at link time; for stack and heap variables, placement must be enforced at run time with modified system codes, *e.g.*, `malloc()`. In the implemented framework, global variable placement is performed by a modified linker, and stack and heap variable placement is performed by modified system libraries.

Figure 6.13 lists the algorithm used by the modified linker KLINK. The modified linker places global variables as specified by the placement map (generated by PLACETOOL), and replaces the startup module (*i.e.*, `crt0.o`) and heap allocation routines (*i.e.*, `malloc.o`) with versions capable of enforcing stack and heap allocation alignments. KLINK is implemented as a post-pass link phase within GNU GLD.

To place global variables, KLINK first binds relocations (*i.e.*, references to variables in the text and data segments) to the global variables they reference. Global variables are then relocated to their new addresses and all relocations affected are fixed. Finally, the data segment is shuffled to match the new placement map, and the updated segments are written to the executable.

The standard ECOFF object file format used in the SimpleScalar tool set (detailed in Chapter 2) was unable to accommodate global variable placement in its current form. First, the symbol tables have no complete record of the start and ending addresses of each variable. Moreover, many global variables are not even listed in the symbol table. To reduce the size of object files, any variable that is not externally referenced (*e.g.*, local labels, static variables, private literals) is not recorded in the symbol table. Any references to these variables are converted to a more compact segment-relative reference format. Finally, reordering can mix initialized and uninitialized variables, forcing some uninitialized variables to be initialized if they are placed amid initialized variables.

After much judicious modification of the compiler, assembler, and linker, all these problems were overcome. All variable symbols were extended to include variable size information, and all variable references were required to be symbol-relative, rather than segment-relative. In addition, all variables up to and including the last initialized variable in the data segment are initialized. These changes did increase executable sizes, sometimes more than 100%. However, this effect results from the limited extensibility of ECOFF object files; most executable overheads could be eliminated with a more accommodating object file design.

Heap allocation placement is implemented at run time using the modified `malloc()` implementation shown in Figure 6.14. The modified `malloc()` first computes the heap allocation name, an integer value, by XOR-folding NAME_DEPTH return addresses from the stack (returned by the function `return_address()`) and truncating the value to be smaller than MAX_NAME. The name generated is used to index the placement map (generated by PLACETOOL), which indicates the alignment requirements for this particular allocation. If the requested alignment is greater than or equal to zero, the allocation is forced to the specified alignment using `forced_aligned_malloc()`. The modified heap allocator is based on a power-of-two heap allocator, thus, it is often the case that sufficient internal fragmentation exists in an allocation to accommodate a requested alignment without incurring memory overheads. Since an aligned allocation may begin anywhere within the power-of-two size allocation block, a back pointer is required immediately preceding the aligned block to indicate the start of the allocation block.

6.5 Experimental Evaluation

This section evaluates the performance of cache-conscious data placement by examining the cache performance and run-times of programs compiled with various placement strategies.

```
Input:    allocation request size, heap allocation map
Output:   heap allocation, possibly aligned to requested cache alignment

Method:   /* compute heap allocation name */
          name = 0;
          for (i=0; i < NAME_DEPTH; i++)
             name = hash(name, return_address(i));
          name = name % MAX_NAME;

          /* determine allocation priority, force alignment, if needed */
          if (place_map[name].force) {
             p = forced_aligned_malloc(place_map[name].alignment, place_map[name].set_size);
          else
             p = aligned_malloc(place_map[name].alignment, place_map[name].set_size);
          q = align_to(p, size, heap_map[name].alignment);
          update_backpointers(p, q);
          return q;
```

Figure 6.14: Modified Heap Allocator. The function `return_address(i)` returns the return address in the `i`th stack frame up the stack. The functions `forced_aligned_malloc()` and `aligned_malloc()` allocate storage with the requested cache alignment. The former always returns aligned storage, increasing the allocation size if needed.

6.5.1 Methodology

All profiling and placement tools were implemented within the SimpleScalar tool set (described in Chapter 2). The implementation, while challenging due to the many system components impacted, was manageable by a single person in only four months time, totaling less than 6000 lines of code including comments.

All experiments were performed with compilers and simulators described in Chapter 2. For each program, five executables were generated using the following placement strategies:

Natural: Natural placement is the variable placement resulting from use of the standard compiler, linker, and system libraries.

Random: Random placement shuffles all global variables in the data segment, and aligns the stack and heap allocations to a randomly selected cache block.

CCDP Real: Cache-conscious data placement (CCDP) employs the data placement algorithms and mechanisms detailed in Section 6.4 with the placement parameters specified in Table 6.1. The placement parameters were chosen after an extended period of tuning the placement algorithms. Variable placement is directed with summary profiles generated with the profile inputs listed in Table 6.2. A different input, also listed in Table 6.2, is used during program analysis.

CCDP Ideal: This placement strategy is identical to *CCDP Real* placement, with the exception that profiling and analysis occur on the same input, *i.e.*, the analyzed inputs listed in Table 6.2. (Note, ideal is not intended to imply optimal placement, rather it is the case where the heuristic placement algorithms are directed with perfect profile information.)

Parameter	Value
NAME_DEPTH	4 frames
MAX_NAME	256
HEAP_THRESHOLD	0.25 refs/byte
ACTIVE_THRESHOLD	0.25 refs/byte
AFFINITY_THRESHOLD	90%
BASE_CACHE_SIZE	8192 bytes (direct-mapped)
INSTANCE_THRESHOLD	1
LOW_THRESHOLD	2.344E-4 refs/byte-cycle
MID_THRESHOLD	4.688E-4 refs/byte-cycle
PROFILE_ARRAY_SIZE	256 buckets
PROFILE_BUCKET_SIZE	32 bytes

Table 6.1: Placement Algorithm Parameters. LOW_THRESHOLD and MID_THRESHOLD are the reference densities used by the data profiler (DPROF) to compress data profiles, *i.e.*, each profile bucket is reduced to the two-bit values NONE, LOW, MID, and HIGH. PROFILE_ARRAY_SIZE and PROFILE_BUCKET_SIZE specify profile array sizes (number of profile buckets) and bucket sizes (in bytes) for the time and space dimension of all variable profiles, respectively.

Program	Arguments and Inputs	
	Analyzed	Profiled
Compress	in	compress, in.Z
Doduc	doducin	smallin, tinyin
GCC	1stmt.i	1toplev.i, 1varasm.i
Ghostscript	FAC-page-4.ps	FAC-page-1.ps, FAC-page-10.ps
Go	50 9 2stone9.in	2 21 5stone.in, 40 8
MPEG_play	coil.mpg	ts2.mpg, moonland.mpg
Perl	tests.pl	sort.pl, grep.pl
Vortex	vortex25.in	vortex25a.in, vortex25b.in
YACR-2	input2.in	input1.in, input3.in

Table 6.2: Analyzed and Profiled Inputs.

Simple: Simple placement employs a simpler, less expensive data placement algorithm. Profile information is limited to a single reference density for each variable. During placement, only active/inactive partitioning is applied, and variables within a partition are placed in natural order. Stack and heap allocations employ natural placement.

Nine programs were analyzed, selected because they exhibit moderately high data cache miss rates for cache geometries found in current-generation microprocessors. The programs are described in Chapter 2. Table 6.2 lists their inputs and arguments used for profiling and analysis. Table 6.3 gives the baseline execution statistics for programs compiled with natural placement. All execution statistics were collected using the baseline simulators described in Section 2.

Figure 6.15 shows the cache performance of the programs with natural placement for 2k, 8k, and 32k direct-mapped data caches. The benchmarks have relatively high data cache miss rates on small caches, with a few performing poorly on caches as large as 32k bytes. The bars show the breakdown of inter- and intra-variable misses. An intra-variable miss is defined as any miss where the replaced and fetched data both come from the same variable. These misses represent a special class of misses

Program	Insts (Mil.)	Loads (Mil.)	Stores (Mil.)	Miss Rate (% of refs) I-cache	D-cache	D-TLB	Mem Usage (K bytes)
Compress	62.4	15.0	7.6	+0.00	16.22	4.241	442
Doduc	1,599.9	544.0	196.2	3.90	4.12	+0.000	148
GCC	123.3	29.1	20.2	3.26	4.93	0.013	1,223
Ghostscript	306.3	73.6	38.7	0.93	3.10	0.030	5,166
Go	315.3	84.9	30.0	3.65	7.95	0.019	591
MPEG_play	529.5	119.9	47.4	1.77	18.35	0.132	3,032
Perl	200.9	53.1	34.7	5.48	7.52	0.189	3,810
Vortex	688.7	182.9	156.0	5.11	6.51	0.222	13,016
YACR-2	429.7	59.3	19.0	+0.00	4.11	+0.000	199

Table 6.3: Program Statistics before Cache-Conscious Data Placement.

for this study, since they cannot be eliminated through variable placement.[2] The remaining miss component, the inter-variable misses, are the result of interactions between variables, and represent an upper bound on the performance improvements that could be achieved with cache-conscious data placement. Clearly, inter-variable interactions account for a significant fraction of data cache misses. As cache size increases, the fraction of misses due to inter-variable interactions grows as well.

Figure 6.16 also shows baseline program cache performance on 2k, 8k, and 32k direct-mapped data caches. The bars in this graph show miss rates broken down by the three miss classes: conflict, capacity, and compulsory misses.[3] Conflict and capacity misses dominate the misses, both decreasing when cache sizes increase. The compulsory miss component is very small for all programs.

6.5.2 Cache Performance

Figure 6.17 shows the miss rates found with an 8k direct-mapped data cache for each of the five placement strategies. The first bar, labeled *Natural*, is the cache performance of the baseline programs using natural placement. Comparing natural placement to random placement (*i.e.*, the bars labeled *Random*) reveals natural placement to be a consistently better placement strategy.

Figure 6.18 gives insights into why this may be. The graph shows the percent change in the miss ratio when going from natural to random placement (a positive change means more misses for random placement). The change in miss rates is also broken down by the three miss class, *i.e.*, conflict, capacity, and compulsory misses. As shown in the graph, random placement increases both the conflict and capacity misses for most of the programs. The compulsory miss component is also increased, although this is difficult to see because the component is very small. Clearly, properties of natural placement

[2]This may not the case when the last block of a variable is interacting with itself and the variable does not completely fill either cache block. In this case, it may be possible to remove the interaction by relocating the variable. However, this anomaly is limited to at most one pair of cache blocks in variables with sizes close to a multiple of the cache set size, making this case very infrequent.

[3]Conflict misses are computed as the difference between the miss rate of the target cache and a fully-associative cache of the same size. Compulsory misses are computed by tracking the frequency storage is first touched. The remaining misses are classified as capacity misses.

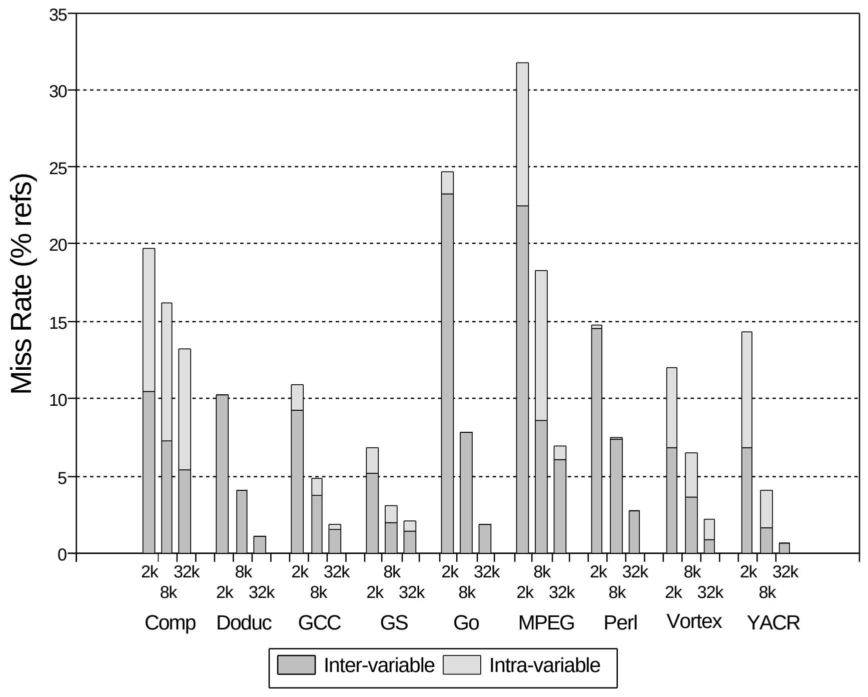


Figure 6.15: Program Miss Rates Broken Down by Variable Interaction.

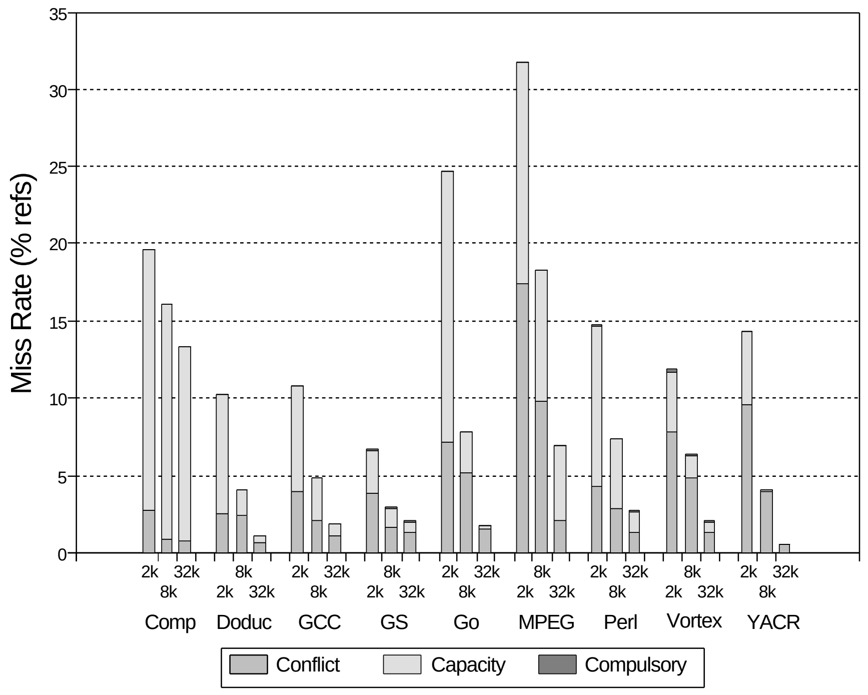


Figure 6.16: Program Miss Rates Broken Down by Miss Class.

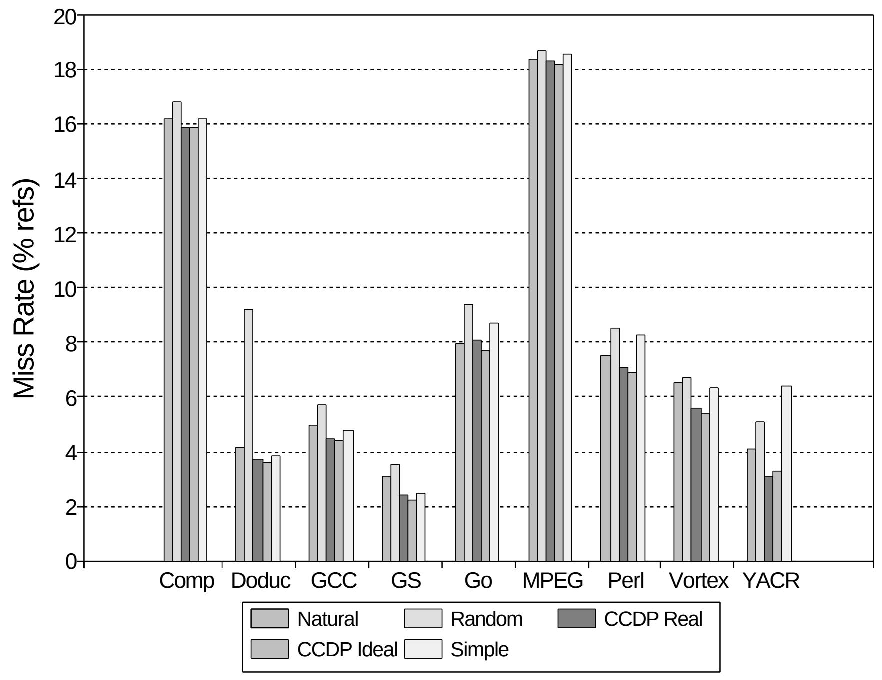


Figure 6.17: Performance of Placement Algorithms – 8k Direct-Mapped Cache.

work to reduce all classes of misses.

The better performance of natural placement appears to be the result of how programmers write programs. Logically related global and stack variables are often defined next each other in the program source. Logically related heap allocations are often allocated near each other in time. Both these actions work to keep logically related variables close to each other in memory. This simple process of grouping related variables serves to reduce all classes of misses. Conflict misses are reduced because the grouped variables are close enough that they do not conflict with each other. Capacity misses are reduced because the grouped variables have significant temporal locality which serves to increase cache line utilization. Compulsory misses are reduced because grouped variables are often placed in the same cache block which more effectively utilizes cache line prefetch. The performance of natural placement clearly sets the bar high for any artificial placement scheme.

The bars labeled *CCDP Real* in Figure 6.17 shows the performance of programs compiled with cache-conscious data placement. For these experiments, different inputs were used for profiling and analysis. Cache performance improvements over natural placement were fairly good, with many of the experiments resulting in more than a 10% reduction in total cache misses. The best improvement was a 25% reduction in data cache misses for *YACR-2*. The only program that suffered from more misses was *Go*, although the increase was very small. Many of the misses in *Go* are the result of accesses to the playing board array. The summary profiles did not do a good job of capturing typical accesses to

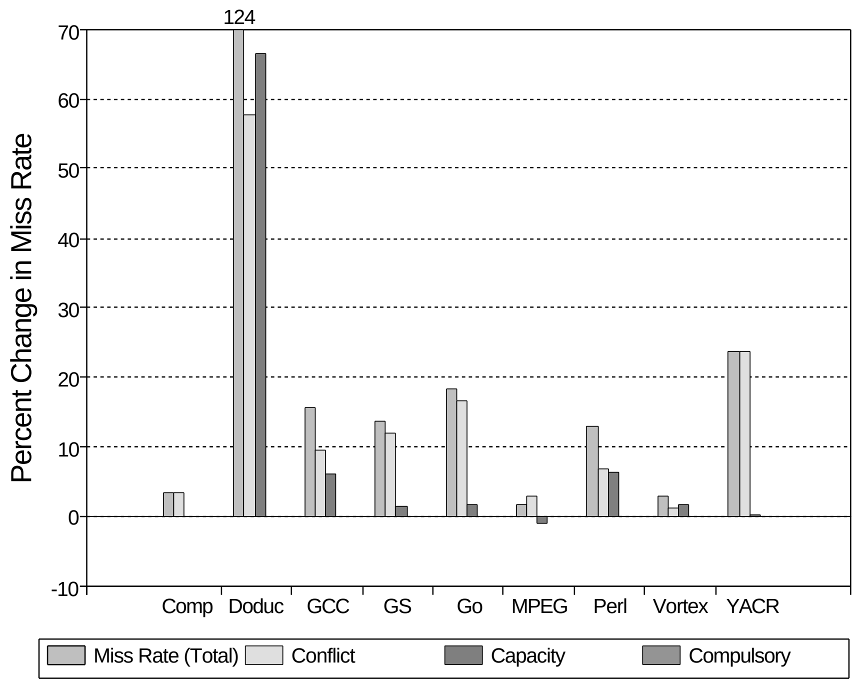


Figure 6.18: Impact of Random Placement for 8k Direct-Mapped Cache.

this variable. When optimized and run with the same input, shown by the bars labeled *CCDP Ideal* in Figure 6.17, the placement optimizations were able to improve cache performance.

Figure 6.19 shows the percent change in miss ratio broken down by miss class when going from natural to cache-conscious data placement. The placement optimizations removed a significant fraction of misses for many of the programs. As shown by the results for *Doduc* and *MPEG_play*, the heuristic placement algorithm appears to be favoring placement solutions that minimize conflict misses, sometimes at the expense of capacity misses. This effect is likely the result of the conflict minimization phase of placement running last, thereby giving preference to placements with low inter-variable conflict. The heuristic placement algorithm would likely benefit from modifications that limit interactions between placement phases.

Overall, summary profiles appear to working well. As the bars labeled *CCDP Ideal* in Figure 6.17 show, placement with perfect profile information only results in small improvements in cache performance. With *YACR-2*, cache performance is actually degraded with perfect profile information – this anomaly is due to the heuristic search algorithms. Since the heuristic placement algorithm works to find good placement solutions, not optimal ones, it is entirely possible for a summary profile input to produce a better placement solution than a perfect profile input.

Table 6.4 quantifies the instruction count and memory overheads for cache-conscious data placement. All overheads are expressed as a percent change in the baseline execution statistics given in in

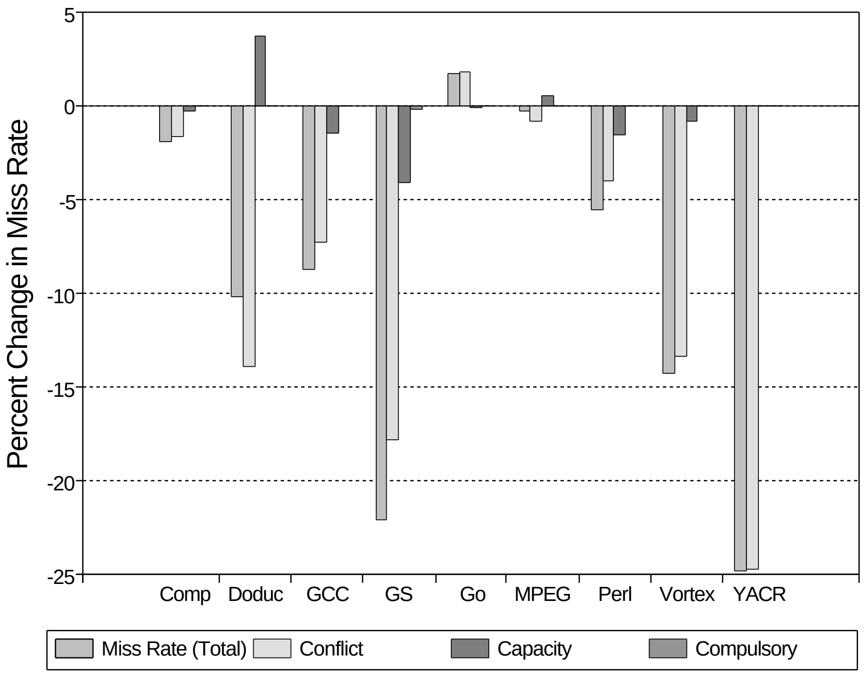


Figure 6.19: Impact of CCDP Placement for 8k Direct-Mapped Cache.

Table 6.3. Execution overheads are the result of extra instructions and memory usage needed to implement stack and heap variable placement. Instruction execution overheads are very small. Memory overheads are much larger. These large overheads are the result of many small, frequently accessed storage allocations being aligned to a specific point in the cache. To place these allocations, the modified storage allocator (detailed in Section 6.4) can increase their size up to the set size of the cache. Data TLB miss rates rates, however, suggest that while the programs are mapping more memory to align dynamic allocations, little of the actual memory is being touched. In most experiments, the data TLB miss rates decreased. Other virtual memory impacts, *e.g.*, page fault rates, were not measured, but given the small absolute increases in program memory usage and the good performance of the data TLB, they are expected to be small.

The last set of bars in Figure 6.17 labeled *Simple*, show the performance of the simple placement algorithm. Simple placement employs a less complex, less expensive version of the cache-conscious data placement algorithm. It performs active/inactive partitioning, placing all global variables within a partition in natural order. Heap and stack variable allocations employ natural placement. As shown in Figure 6.17, this placement strategy has significantly less stability than the more complex cache-conscious data placement algorithm. The reduced performance is primarily due to increased conflict misses. It appears that for consistent cache performance improvements on a direct-mapped cache, the placement algorithm needs to consider how placement decisions affect inter-variable conflict.

Program	Insts	Loads	Stores	Miss Ratio			Mem Usage
				I-cache	D-cache	D-TLB	
Compress	+0.00	+0.00	+0.00	+0.00	-1.97	-0.06	+8.14
Doduc	+0.00	+0.00	+0.00	+0.00	-10.19	0.00	0.00
GCC	+0.04	+0.04	+0.00	+0.00	-8.72	-0.02	+24.20
Ghostscript	+0.00	+0.00	+0.00	+0.00	-22.58	-0.06	+1.86
Go	+0.08	+0.05	+0.01	+0.00	+1.76	-0.03	+4.74
MPEG_play	+0.00	+0.02	+0.00	+0.00	-0.27	-0.00	+1.85
Perl	+0.16	+0.14	+0.15	+0.09	-5.59	-0.03	+7.14
Vortex	+0.01	+0.02	+0.00	+0.01	-14.29	+0.01	+0.03
YACR-2	+0.01	+0.00	+0.00	+0.00	-24.57	+0.00	+70.35

Table 6.4: Percent Change in Program Statistics after Cache-Conscious Data Placement.

6.5.3 Impact of Varied Cache Geometry

Since variable placement optimizations occur at compile time, the placement algorithm must optimize cache performance for a single target cache geometry. This section examines the performance of programs running on cache geometries other than the target cache geometry.

Figure 6.20 shows the performance of each program (optimized for execution on an 8k direct-mapped cache) running on an 8k 2-way set-associative data cache. This configuration eliminates many conflicts found in the direct-mapped cache of the experiments in Figure 6.17. Performance improvements afforded by placement optimization is smaller than for the direct-mapped cache since many of the conflict misses eliminated by cache-conscious data placement no longer exist in the set-associative cache geometry. Simple placement fairs notably better in this configuration, performing nearly as well as cache-conscious data placement for all programs, since the approach is no longer strapped with conflict misses.

Figures 6.21 and 6.22 show the performance of the programs running on 2k and 32k direct-mapped caches, respectively. Performance improvements vary more so the smaller cache configuration; performance improvements for *Doduc* and *Perl* are lost, while the performance improvements for *Ghostscript* and *Go* increased. This variation is likely due to conflict misses induced in the smaller cache sizes that were not considered during variable placement. With a larger cache configuration, *i.e.*, 32k, cache-conscious data placement still improved cache performance for most programs, although improvements were small due to better overall cache performance. The performance of *Perl* degraded on the larger direct-mapped cache. This case may be the result of more conflict being eliminated (when increasing the cache size) for natural placement than for cache-conscious data placement; the exact cause is still under investigation.

6.5.4 Program Performance

Improvements in cache miss rate do not translate directly into program run-time improvements. The impact the placement optimizations have on program performance is affected by the program's overall

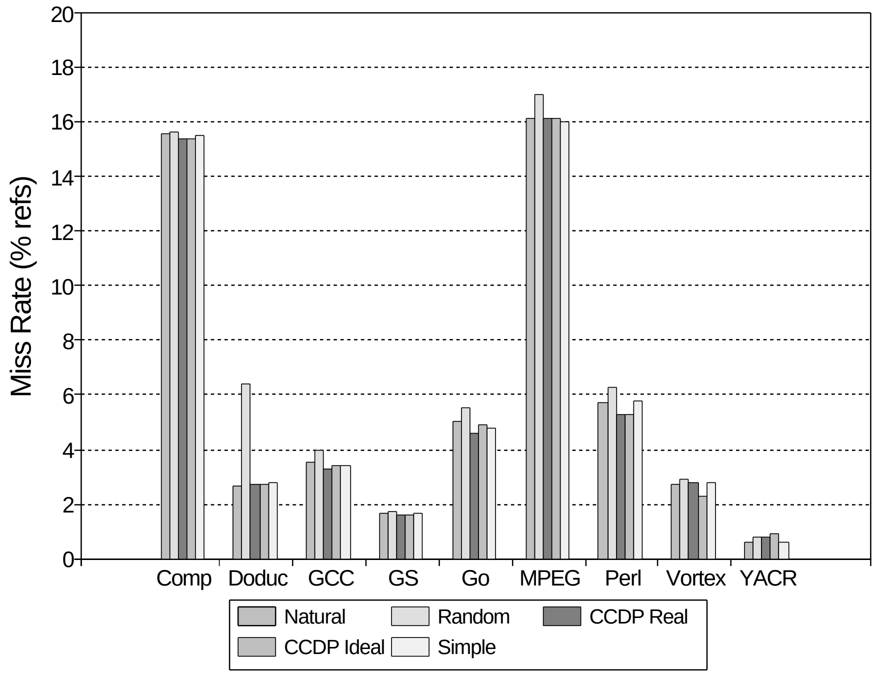


Figure 6.20: Performance of Placement Algorithms – 8k 2-Way Set-Assoc. Cache.

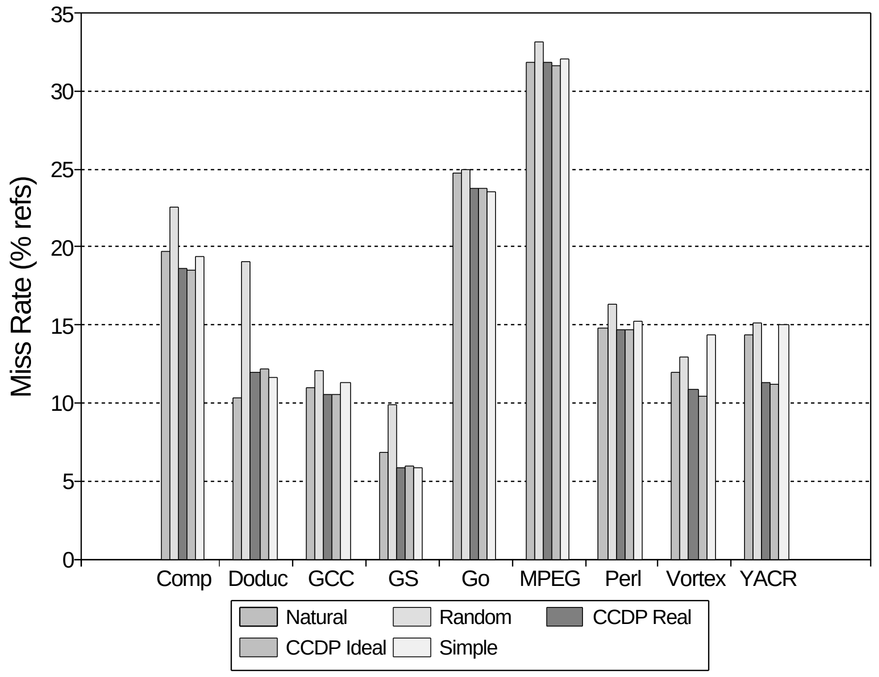


Figure 6.21: Performance of Placement Algorithms – 2k Direct-Mapped Cache.

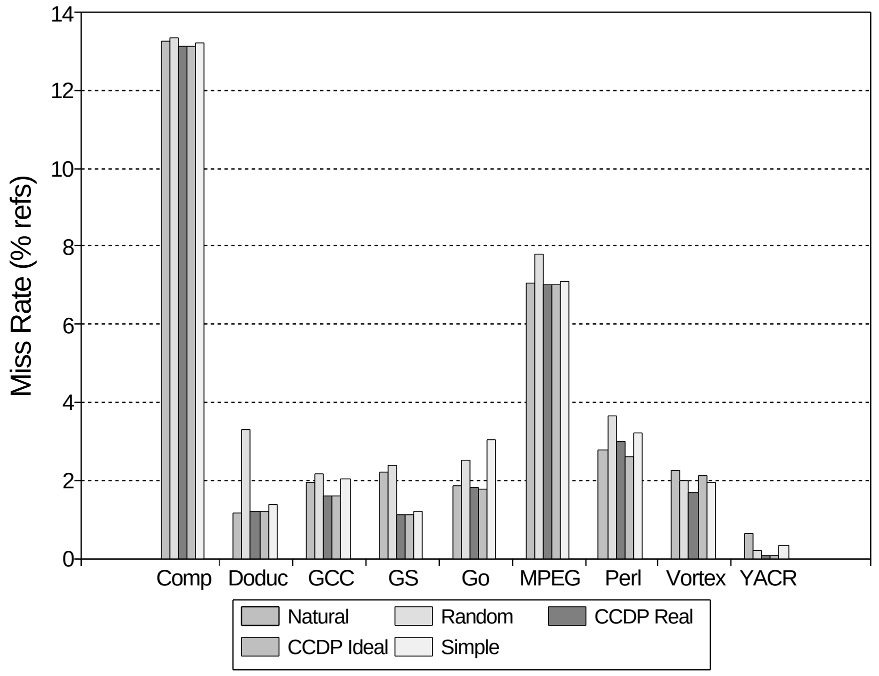


Figure 6.22: Performance of Placement Algorithms – 32k Direct-Mapped Cache.

cache performance and the processor's ability to tolerate cache miss latency. A better measure of program performance impacts requires a more detailed analysis. To gauge the performance of cache-conscious data placement in the context of a realistic processor model and memory system, program performance was examined running on the baseline in-order and out-of-order superscalar timing simulators detailed in Chapter 2.

Table 6.5 lists the baseline program performance (*i.e.*, compiled with natural placement) on the in-order and out-of-order issue simulators. The column, labeled *Total Miss CPI*, shows the total data cache miss latency per instruction, tolerated or otherwise. For each processor model, the table gives the CPI, and for the data cache miss latency the total exposed latency per instruction, the percent it is of total CPI, and the percent of total miss latency CPI tolerated by the processor model. While the total miss latency CPI is a sizeable fraction of the CPI for many of the programs, both processor models do a fairly good job of tolerating the latency, with the out-of-order issue processor model tolerating more cache miss latency for all the programs.

Figure 6.23 shows program speedups while running on the baseline in-order and out-of-order timing simulators. Performance impacts are small for both models. All speedups were less than 6% for the in-order issue processor, all less than 2% for the out-of-order issue processor. Performance for *Go* was worse due to its poorer cache performance after cache-conscious data placement. Performance impacts were small because both processors are tolerating much of the cache miss latency eliminated. Since

Program	Total	In-order				Out-of-order			
	Miss CPI	CPI	Miss CPI			CPI	Miss CPI		
			Exposed	% CPI	% Tolerated		Exposed	% CPI	% Tolerated
Compress	0.358	0.889	0.221	24.84	38.31	0.655	0.118	18.00	67.07
Doduc	0.113	1.131	0.041	3.60	63.97	0.905	0.015	1.60	87.19
GCC	0.112	0.901	0.050	5.53	55.51	0.763	0.034	4.45	69.68
Ghostscript	0.066	0.864	0.042	4.85	36.51	0.633	0.028	4.41	57.70
Go	0.162	1.153	0.087	7.57	46.12	0.954	0.059	6.20	63.49
MPEG_play	0.330	1.021	0.188	18.42	43.01	0.729	0.136	18.62	58.87
Perl	0.181	1.091	0.068	6.24	62.39	0.944	0.043	4.59	76.06
Vortex	0.186	0.927	0.111	11.99	40.24	0.805	0.079	9.80	57.59
YACR-2	0.043	0.637	0.030	4.67	30.82	0.427	0.020	4.73	53.03

Table 6.5: Baseline Program Performance.

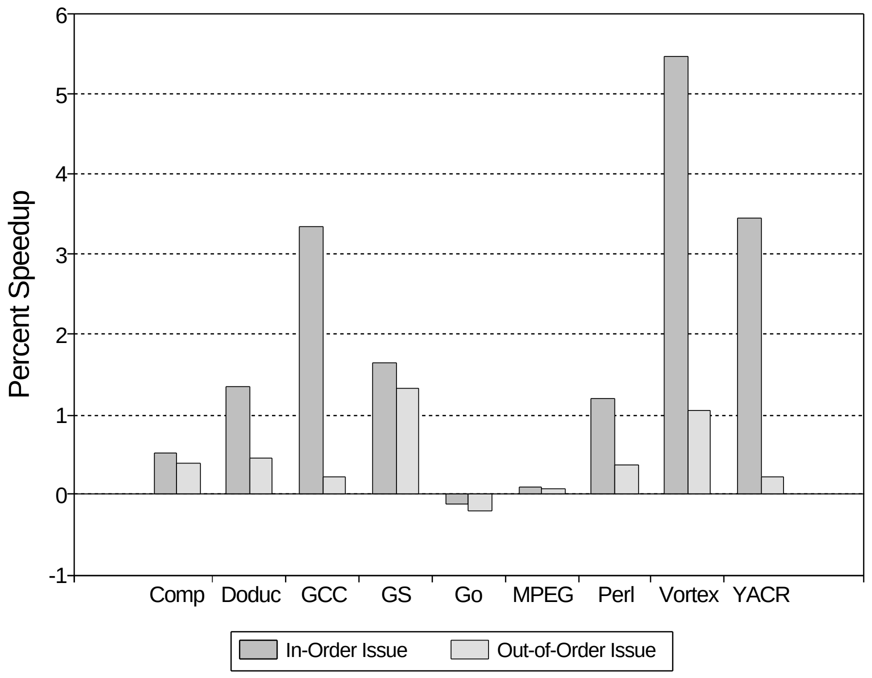


Figure 6.23: Performance on Detailed Timing Simulator – 8k Cache.

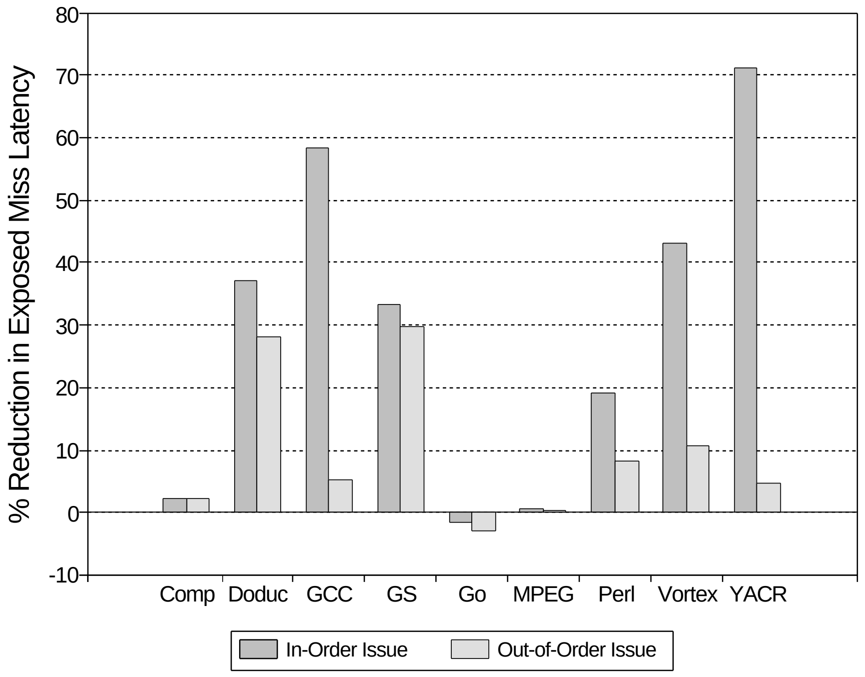


Figure 6.24: Reduction in Exposed Miss Latency – 8k Cache.

the out-of-order issue processor has better latency tolerating capabilities, speedups are smaller. Figure 6.24 shows the percent reduction in the exposed data cache miss latency component. In this light, performance impacts are much larger. The placement optimizations were able to eliminate much of the exposed cache miss latency. For the in-order issue processor, many of the programs saw more than a 30% reduction in exposed cache miss latency; reductions were less for the out-of-order issue processor.

6.6 Related Work

Many parallels to this work can be found in software techniques developed for improving instruction cache performance. Techniques such as basic block re-ordering [Dav95, Wu92, PH90], function grouping [Wu92, PH90, Fer76, Fer74, HG71], and text ordering based on control structure [McF91] have all been shown to significantly improve instruction cache performance. Like this work, the approaches usually rely on profile information to guide heuristic algorithms in placing instructions to minimize instruction cache conflicts, and maximize cache line utilization and block prefetch. Unlike this work, some of the studies concentrate on virtual memory system performance, *i.e.* TLB miss and page fault rates. The approaches presented in this paper would certainly extend to lower levels of the memory hierarchy as well; this is a potential area for future exploration.

A number of peripheral works employ data relocation to improve data cache performance. Page coloring [BLRC94, LBF92, Kes91] techniques have leveraged the memory mapping capability of virtual memory to reduce conflicts in physically indexed caches. User-programmable cache set mappings [DS91] have been proposed to provide the same benefits. Compiler-directed dimension extension [CL89] works to relocate data within large arrays, giving opportunity to improve data cache performance when a large array conflicts with itself. Data placement optimizations have been used to reduce false sharing in shared memory multiprocessors [JE95]. Compiler-directed variable partitioning has been proposed as an approach to reduce inter-variable interactions [Mue95] for the purpose of improving the predictability of cache access latencies in real-time systems. The Scout operating system [MMO$^+$94] employs data placement to reduce data cache conflict between active protocol stacks.

The approach used to name heap variables was adopted from [BZ93].

The history of the development of this work is relevant related work. The initial placement strategy tried to fill the padding space created by variable alignment with other variables. By packing variables, the data segment size could be reduced, sometimes by as much as 10%, with the thought that cache blocks would be better utilized and cache performance would improve. It quickly became evident this was a poor placement strategy, nearly all experiments resulted in an increase in cache misses. This result led to the development and evaluation of random placement and the observation that natural placement performs very well. Next, the active/inactive variable partitioning algorithm (*i.e.*, the *simple* placement strategy in Section 6.5) was tried. This algorithm used profile information to improve cache line utilization. While this algorithm performed well for some programs, others performed very poorly, especially on direct-mapped caches due to increased inter-variable conflicts. It soon became evident that any effective approach must also work to minimize conflict misses. This observation led to the cache-conscious data placement algorithm described in this chapter. Initially, only global variable placement was optimized, but this approach also lacked good stability due to occasional conflicts that arose with stack and heap variables. The algorithms were then expanded to accommodate placement of stack and heap variables as well, this produced the current generation algorithms which have been shown to consistently improve cache performance for the targeted workloads and cache geometries.

6.7 Chapter Summary

Cache-conscious data placement was introduced as a software-based technique to improve data cache performance by relocating variables in the virtual memory space. The approach employs data profiling to characterize variable usage. Profile information then guides heuristic variable placement algorithms in finding a variable placement solution that decreases predicted inter-variable conflict, and increases predicted cache line utilization and block prefetch. The generated placement solution is implemented using a modified linker and system libraries.

Four placement strategies were examined in detail using detailed timing simulation: natural, random, cache-conscious data placement, and simple. Natural placement sets the bar high for artificial placement measures. Compared to random placement, natural placement was found to have consistently better cache performance. Programmers group logically related variables which results in natural placements with good temporal and spatial locality between variables.

Cache-conscious data placement effectively improved data cache performance, eliminating more than 10% of all misses for many of the programs tested. Cache performance degraded for only one of the tested program, the result of poor summary profile information. Overall, summary profile performance was excellent, with only slight improvements in placement performance when programs were optimized and analyzed with the same input.

A simplified version of the placement algorithms was also evaluated. The simpler algorithm used only reference densities (instead of profiles) and performed only active/inactive variable partitioning. This placement strategy, while performing well for some programs, lost the consistent performance improvements found with the more complex placement algorithm. The degradation for some programs was due to lack of consideration during placement for conflict misses, suggesting that detailed profile information and conflict-miss sensitive placement algorithms will be required for consistent performance improvements, especially for direct-mapped caches.

Cache geometries were also varied to see what effect this had on a program that was optimized for another cache geometry. With the addition of associativity, much of the performance improvements found with cache-conscious data placement were lost because conflict miss reductions were no longer eliminated. With smaller cache sizes, cache performance often degraded due to conflict misses induced but not considered by the placement algorithms. Larger cache sizes still saw consistent performance improvements, although impacts were smaller due to the decreased miss rates for the larger caches.

Finally, the impact cache-conscious data placement had on exposed load latency was explored through detailed timing simulation. Programs were simulated running on a 4-way superscalar processor with one level of instruction and data cache memory. The latency tolerating capability of the processor was varied by examining performance on simulators supporting in-order and out-of-order instruction issue. Cache-conscious data placement was able to remove a large fraction of exposed miss latency for many of the programs tested. However, program speedups were generally small, especially for the out-of-order issue processor, since the processors were already tolerating a large fraction of data cache miss latency.

Chapter 7

Conclusion

As processor demands quickly outpace memory, the performance of load instructions becomes an increasingly critical component to good system performance. This thesis contributes four novel load latency reduction techniques, each targeting a different component of load latency: *address calculation*, *data cache access*, *address translation*, and *data cache misses*.

7.1 Thesis Summary

For many codes, especially integer codes with good cache performance, exposed address calculation latencies account for a significant fraction of total execution time. This thesis introduces a pipeline optimization, called *fast address calculation*, that permits effective address calculation to proceed in parallel with data cache access, thereby eliminating the extra cycle required for address calculation. The technique employs a simple circuit to quickly predict the portion of the effective address needed to speculatively access the data cache. If the address is predicted correctly, the cache access completes without an extra cycle for address calculation. If the address is mispredicted, the cache is accessed again using the correct effective address. The predictor has minimal impact on cache access latency, adding only a single OR operation before data cache access can commence. Prediction verification is also very fast and decoupled from the cache access critical path, ensuring that pipeline control logic impacts are minimal.

Detailed timing simulations of a 4-way in-order issue superscalar processor extended to support fast address calculation found good speedups for all the programs tested. The experiments showed an average speedup of 14% for the integer codes and 6% for the floating point codes. Simple software support was also developed to improve prediction accuracy. Simulated performance with software support improved the average speedup to 19% for the integer codes and 7.5% for the floating point codes.

Address calculation latency, however, is at most one half of the latency of loads, leaving one or more cycles of cache access latency exposed to extend execution critical paths and stall instruction issue. This thesis further extends the latency reduction capability of fast address calculation by combining it with an early-issue mechanism. The resulting pipeline designs are capable of reducing the latency of load instructions by up to two cycles. For a pipeline with one cycle data cache access, loads can complete before reaching the execute stage of the pipeline, creating what is termed a *zero-cycle load*. A zero-cycle load allows subsequent dependent instructions to issue unencumbered by load instruction hazards, resulting in fewer pipeline stalls and increased overall performance.

Detailed evaluations using a 4-way in-order issue processor simulator extended to support zero-cycle loads found an average speedup of 45% for the integer codes and 26% for the floating point codes. Performance for the integer codes was nearly on par with programs running on an out-of-order issue processor (without support for zero-cycle loads). Speedups on an out-of-order issue processor simulator extended to support zero-cycle loads were less due to the latency tolerating capability of the execution model. On architectures with few registers, the frequency of loads and their impact of program performance increases significantly. Providing an 8 register architecture with limited zero-cycle load support resulted in performance comparable to a 32 register architecture, suggesting that the approach may be able to negate the impacts of too few architected registers.

In support of the proposed cache hit optimizations, this thesis presents four new high-bandwidth address translation mechanisms. These new mechanisms feature better latency and area characteristics than current TLB designs, providing architects with effective alternatives for keeping address translation off the critical path of loads. Two designs are borrowed from traditional high-bandwidth memory design techniques, creating *interleaved* and *multi-level* TLB designs. In addition, two more designs crafted specifically for high-bandwidth address translation are introduced. *Piggyback ports* are proposed as a technique to exploit spatial locality in simultaneous translation requests, allowing accesses to the same virtual memory page to combine their requests at the TLB access port. *Pretranslation* is proposed as a technique for attaching translations to base register values, making it possible to reuse a single translation many times.

Extensive simulation-based studies were performed to evaluate address translation designs using the proposed high-bandwidth mechanisms. A number of designs show particular promise. Multi-level TLBs with as few as eight entries in the upper-level TLB nearly achieve the performance of a TLB with unlimited bandwidth. Piggyback ports combined with a lesser-ported TLB structure, *e.g.*, an interleaved or multi-ported TLB, also perform well. Pretranslation over a single-ported TLB performs almost as well as a same-sized multi-level TLB with the added benefit of decreased access latency for physically indexed caches.

Finally, this thesis examines a software-based variable placement optimization for reducing the frequency of data cache misses. The approach, called *cache-conscious data placement*, uses profile-guided heuristics to find variable placement solutions that decrease predicted inter-variable conflict, and increase predicted cache line utilization and block prefetch. The generated placement solutions are implemented partly at compile-time using a modified linker and partly at run-time with modified system libraries.

Various placement strategies were developed and compared to the performance of natural placement (*i.e.*, the naturally occurring layout of variables). Random placement performed consistently worse than natural placement, revealing natural placement as an effective placement strategy that sets the bar high for artificial placement measures. Cache-conscious data placement effectively improved data cache performance, eliminating more than 10% of all misses for many of the programs tested. A simplified and less expensive version of the placement algorithm (*i.e.*, with reduced computation and storage requirements) lost the stability of the more complex algorithm, suggesting that for consistent performance improvements the more capable and expensive algorithm is required. Run-time performance impacts were also examined for in-order and out-of-order issue processor models. Cache-conscious data placement was able to remove a large fraction of exposed miss latency for many of the programs tested. However, program speedups were generally small, especially for the out-of-order issue processor, since the processors were already tolerating a large fraction of cache miss latency.

7.2 Future Directions

The work in this thesis could be extended in many ways. The following sections suggest a number of promising directions to pursue.

7.2.1 Less-Speculative Fast Address Calculation

Fast address calculation uses carry-free addition to compute the set index portion of the effective address computation. In some designs, it may be possible to account for some carries in the set index computation after data cache access begins by steering the row decoders after cache access starts or late-selecting the correct word from multiple cache rows.

Consider a simple example that highlights the first possibility: If a carry is generated into the set index portion of the effective address computation, the resulting data cache row selection will be off by at most one row. If the carry output of the block offset computation were used to steer the last level of row decode, row selection and block offset carry generation could proceed in parallel. The design would no longer incorrectly predict the wrong cache row if a carry were generated out of the block offset computation. It may be possible to compute other carries in the set index computation and use these carries to steer row decoding as well. The degree of which this may be done depends on the specifics of the data cache design and implementation.

The second approach works by increasing the opportunities to late-select cache data. If a carry cannot be computed prior to data cache row access, it may be possible to compute both possible outcomes of the address computation, read both data cache rows, and late-select the data in the column select logic using the outcome of the carry computation. A similar approach is employed in carry-select adders [HP90].

7.2.2 Combining Stateful and Stateless Address Predictors

Fast address calculation has a very predictable and easy to identify failure mode, *i.e.*, loads that use `register+register` mode addressing constitute nearly all of the prediction failures. These accesses are the result of array accesses that the compiler cannot strength-reduce. Stateful address predictors, like the load delta table [EV93], have been shown to work well on array accesses. It seems likely that combining the two approaches could produce more accurate, albeit more expensive, predictor designs. Similar hybrid approaches have worked well for branch prediction [ECP96].

7.2.3 Leveraging Address Prediction Information

A common mechanism used repeatedly in this thesis is address prediction. Fast address calculation employs a stateless set index predictor, as do zero-cycle loads. Pretranslation predicts the next virtual page address a base register will access. Reliable address prediction information could be used as mechanism to improve the performance of a number of other pipeline processes, including cache block prefetch, access/execute computation decomposition, and load/store disambiguation.

7.2.4 Additional Compiler Support for Zero-Cycle Loads

The two most common failure modes for zero-cycle loads are failed fast address calculations and register conflicts (*i.e.*, data hazards between unfinished instructions and the inputs of the zero-cycle load). While software support was employed to improve the performance of fast address calculation, no software support was available to reduce register conflicts. By increasing the apparent latency of instructions that produce pointer values, the instruction scheduler would work to insert more instructions between generation of a pointer value and its first use by a load, thereby reducing the load's exposure to register conflicts. Similar approaches have been shown to work well for "AGI" (or late execute) style pipelines [GM94].

7.2.5 Exposing Address Translation to the Compiler

The pretranslation technique described in Chapter 5 performed well, but overall worse than a same-sized L1 TLB. The primary reason for this difference lies in the mechanism by which each design reuses translations. The pretranslation design is only able to reuse a translation whenever a register pointer is reused, whereas the multi-level TLB design is able to reuse a translation whenever an address is reused. Since the latter case is more frequent, the L1 TLBs generally perform better than same-sized pretranslation designs.

By exposing the inefficiencies of the pretranslation mechanism to the compiler, it should be able to improve the performance of the approach. The compiler could mark instructions which create new register pointers, allowing the processor to initiate pointer value translation when a new pointer value is created, rather than when it is first used by a load or store. In addition, simple modifications to the compiler's register allocation priorities (*e.g.*, make pointer value spills more expensive than non-pointer spills) would work to increase the lifetime of register pointers.

7.2.6 Improving Cache-Conscious Data Placement

While cache-conscious data placement gave fairly good performance improvements, it was shown that a large portion of the misses caused by inter-variable interactions were unaffected. The following list details modifications to the current approach that may improve the impact of cache-conscious data placement:

- The current optimization framework favors conflict miss reduction, even at the expense of introducing capacity misses. This effect appears to be the result of the ordering of the placement algorithm phases. While the current phase ordering provided the best performance for this framework, modifications to the algorithms to control phase interactions would likely result in better performance. Phase interactions between register allocation and instruction scheduling algorithms have been effectively controlled with heuristic methods [BEH91].

- The current optimization framework places all variables at compile time. While inexpensive and simple to implement, the approach lacks the ability to adapt to a particular run of the program. A challenging area of future exploration is run-time variable placement. The approach will require a reliable and inexpensive mechanism to gauge whether a variable placement is a good one. Any implementation would likely benefit from novel hardware and/or software support for profiling and placing variables. Run-time placement techniques have already been shown to work well for virtual memory page placement [BLRC94].

- The current placement algorithms targeted performance improvements for a user-specified fixed-size target cache geometry. Performance improvements on other cache geometries were not as impressive and in many cases stability was lost. Methods have been developed for virtual page placement that optimize for multiple target cache geometries, *e.g.*, Kessler's hierarchical page placement algorithm. Incorporating these approaches into the variable placement algorithm would likely improve program performance over a range of cache geometries.

- The partitioning and grouping phases of variable placement could not be applied to heap variables, resulting in many missed opportunities to improve the cache performance of heap variables. If the placement optimizations were instead directed at the virtual memory system, the larger size of virtual memory pages would permit heap variables to be candidates for all phases of variable placement optimization.

- The current profiling strategy uses profiles with fixed array and bucket sizes. While this approach is simple to implement, memory limitations usually require profiles to be small, resulting in low resolution profiles and poor quality profile information. Profile precision and memory demands could both be improved through the use of P-quantile profiles. A P-quantile profile is a profile with variable-size buckets which track the start and end points of where each $1/P$ of the samples lie. One disadvantage of P-quantile profiles is that they require two passes over the sample stream, however, P-quantile estimators [BZ93, JC85] have been shown to work well with only a single pass. Alternatively, profiles could support progressively increasing buckets sizes like those employed in the Paradyn parallel performance measurement tools [MCC$^+$95].

Bibliography

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[AP93] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):179–190, May 1993.

[APS95] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data cache access with fast address calculation. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 23(2):369–380, June 1995.

[AS92] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351. Association for Computing Machinery, May 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[ASW$^+$93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, December 1993.

[AVS93] T. M. Austin, T.N. Vijaykumar, and G. S. Sohi. Knapsack: A zero-cycle memory hierarchy component. Technical Report TR 1189, Computer Sciences Department, UW–Madison, Madison, WI, November 1993.

[BC91] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. *Supercomputing '91*, pages 176–186, 1991.

[BCT92] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 27(7):311–321, July 1992.

[BEH91] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 19(2):122–131, April 1991.

[BF92] B. K. Bray and M. J. Flynn. Translation hint buffers to reduce access time of physically-addressed instruction caches. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):206–209, December 1992.

[BHIL94] J. Borkenhagen, G. Handlogten, J. Irish, and S. Levenstein. AS/400 64-bit PowerPC-compatible processor implementation. *ICCD*, 1994.

[BKW90] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 18(2):270–279, May 1990.

[BLRC94] B. N Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. *Conference Proceedings of the Sixth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1994.

[BRG$^+$89] D. Black, R. Rashid, D. Golub, C. Hill, and R. Baron. Translation lookaside buffer consistency: A software approach. *Proceedings of the Third International Conference on Architectural Support for Programming Languages Operating Systems*, pages 113–122, 1989.

[BZ93] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):187–196, June 1993.

[CB92] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):51–61, October 1992.

[CBJ92] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 19(2):114–123, May 1992.

[CBM$^+$92] W. Y. Chen, R. A. Bringmann, S. A. Mahlke, R. E. Hank, and J. E. Sicolo. An efficient architecture for loop based data preloading. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):92–101, December 1992.

[CCH$^+$87] F. Chow, S. Correll, M. Himelstein, E. Killian, and L. Weber. How many addressing modes are enough. *Conference Proceedings of the Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–121, October 1987.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):53–65, June 1990.

[CD89] C.-H. Chi and H. Dietz. Unified management of registers and cache using liveness and cache bypass. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 24(7):344–355, July 1989.

[Che87] R. Cheng. Virtual address caches in UNIX. *Proceedings of the Summer 1987 USENIX Technical Conference*, pages 217–224, 1987.

[CK92] T. Chiueh and R. H. Katz. Eliminating the address translation bottleneck for physical address cache. *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):137–148, October 1992.

[CK93] R. F. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. Technical Report UWCSE 93-06-06, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1993.

[CL89] C.-L. Chen and C.-K. Liao. Analysis of vector access performance on skewed interleaved memory. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):387–394, May 1989.

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 23(2):333–344, June 1995.

[CMT94] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 28(5):252–262, October 1994.

[Con92] T. M. Conte. Tradeoffs in processor/memory interfaces for superscalar processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):202–205, December 1992.

[Dav95] B. Davidson. The Microsoft LEGO system. UW-Madison Programming Languages Seminar presentation, September 1995.

[DM82] D. R. Ditzel and H. R. McLellan. Register allocation for free: the C machine stack cache. In *1st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Palo Alto, CA, March 1982.

[DS91] F. Dahlgren and P. Stenstrom. On reconfigurable on-chip data caches. *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 22(1):189–198, November 1991.

[ECP96] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. *Proceedings of the 23nd Annual International Symposium on Computer Architecture*, May 1996.

[EV93] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM J. Res. Develop.*, 37(4):547–564, July 1993.

[Fer74] D. Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[Fer76] D. Ferrari. The improvement of program behavior. *Computer*, 9(11):39–47, November 1976.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transaction on Computers*, C-30(7):478–490, July 1981.

[FJ94] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 22(2):211–222, April 1994.

[FP91] M. Farrens and A. Park. Dynamic base register caching: A technique for reducing address bus width. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 19(3):128–137, May 1991.

[Fra93] M. Franklin. *A New Paradigm for Superscalar Processing.* PhD thesis, UW–Madison, Madison, WI, To appear 1993.

[Gao93] Q. S. Gao. The chinese remainder theorem and the prime memory system. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):337–340, May 1993.

[GGK$^+$83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer – designing a mimd, shared memory parallel machine. *IEEE Transactions on Computers*, 32(2):175–189, February 1983.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for pipelined processors. *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, 21(7), July 1986.

[GM93] M. Golden and T. Mudge. Hardware support for hiding cache latency. CSE-TR-152-93, University of Michigan, Dept. of Electrical Engineering and Computer Science, February 1993.

[GM94] M. Golden and T. Mudge. A comparison of two pipeline organizations. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 25(1):153–161, November 1994.

[Gwe94a] L. Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12):1–10, September 1994.

[Gwe94b] L. Gwennap. MIPS R10000 uses decoupled architecture. *Microprocessor Report*, 8(14):18–22, October 1994.

[Gwe95] L. Gwennap. Hal reveals multichip SPARC processor. *Microprocessor Report*, 9(3):1–11, March 1995.

[Hea86] M. Hill and et al. Design decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[HG71] D. J. Hatfield and J Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[HHL$^+$90] K. Hua, A. Hunt, L. Liu, J-K. Peir, D. Pruett, and J. Temple. Early resolution of address translation in cache design. *Proceedings of the 1990 IEEE International Conference on Computer Design*, pages 408–412, September 1990.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[HS89] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[Hsu94] P. Y.–T. Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[IL89] D. T. Harper III and D. A. Linebarger. A dynamic storage scheme for conflict-free vector access. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):72–77, May 1989.

[JC85] R. Jain and I. Chlamtac. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, October 1985.

[JE95] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile-time data transformations. *Proceedings of the Symposium on Principals and Practice of Parallel Programming*, July 1995.

[Jol91] R. Jolly. A 9-ns 1.4 gigabyte/s, 17-ported CMOS register file. *IEEE J. of Solid-State Circuits*, 25:1407–1412, October 1991.

[Jou89] N. P. Jouppi. Architecture and organizational tradeoffs in the design of the MultiTitan CPU. *Proceedings of the 16st Annual International Symposium on Computer Architecture*, 17(3):281–289, May 1989.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 18(2):364–373, May 1990.

[Jou93] N. P. Jouppi. Cache write policies and performance. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):191–201, May 1993.

[JW94] N. P. Jouppi and S. J.E. Wilton. Tradeoffs in two-level on-chip caching. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 22(2):34–45, April 1994.

[KCE92] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):175–186, October 1992.

[KE93] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is unknown. *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):278–289, June 1993.

[Kes91] R. E. Kessler. *Analysis of Multi-Megabyte Secondary CPU Cache Memories.* Tr 1032, Computer Sciences Department, UW–Madison, Madison, WI, July 1991.

[KH92a] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[KH92b] P. Kolte and M. J. Harrold. Load/store range analysis for global register allocation. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):268–277, June 1992.

[KJLH89] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131–139, 1989.

[KT91] M. Katevenis and N. Tzartzanis. Reducing the branch penalty by rearranging instructions in a double-width memory. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 19(2):15–27, April 1991.

[LBF92] W. L. Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):222–225, December 1992.

[LE89] H. Levy and R. Eckhouse. *Computer Programming and Architecture, The VAX.* Digital Press, 1989.

[LGN92] P. Lenir, R. Govindarajan, and S.S. Nemawarkar. Exploiting instruction-level parallelism: The multithreaded approach. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):189–192, December 1992.

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LS94] A. Lebeck and G. Sohi. Request combining in multiprocessors with arbitrary interconnection networks. *IEEE TPDS*, November 1994.

[LS95] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 30(6):291–300, June 1995.

[LW94] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[MCC$^+$95] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995.

[McF89] S. McFarling. Program optimization for instruction caches. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, 24:183–191, April 1989.

[McF91] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[McF92] S. McFarling. Cache replacement with dynamic exclusion. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 20(2):191–200, May 1992.

[ME92] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, December 1992.

[MLC$^+$92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, December 1992. Association for Computing Machinery.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):62–73, October 1992.

[MMO$^+$94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report TR 94–20, Department of Computer Science, University of Arizona, Tucson, AZ, June 1994.

[Mue95] F. Mueller. Compiler support for software-based cache partitioning. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 30(11):125–133, June 1995.

[PH90] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[PS90] K. V. Palem and B. B. Simmons. Scheduling time-critical instructions on RISC machines. *SIGPLAN Notices*, 25(6):270–279, June 1990.

[Rau91] B. R. Rau. Pseudo-randomly interleaved memory. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 19(3):74–83, May 1991.

[RL92] A. Rogers and K. Li. Software support for speculative loads. *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[Sez93] A. Seznec. A case for two-way skewed-associative caches. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):169–178, May 1993.

[SF91] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[SL93] A. Seznec and J. Lenfant. Odd memory systems may be quite interesting. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):341–350, May 1993.

[Sla94] M. Slater. AMD's K5 designed to outrun Pentium. *Microprocessor Report*, 8(14):1–11, October 1994.

[Smi81] B. J. Smith. Architecture and applications of the HEP multiprocessor. *SPIE*, pages 241–248, 1981.

[Smi82a] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smi82b] J. E. Smith. Decoupled access/execute architectures. *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[SPE91] SPEC newsletter. Fairfax, Virginia, December 1991.

[SPE95] SPEC newsletter. Fairfax, Virginia, September 1995.

[Ste88] G. Steven. A novel effective address calculation mechanism for RISC microprocessors. *Computer Architecture News*, 16(4):150–156, September 1988.

[TH94] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 29(11):171–182, November 1994.

[Vee86] A. H. Veen. Dataflow machine architecture. *Computing Surveys*, 18(4):365–396, December 1986.

[WBL89] W.-H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):140–148, May 1989.

[WE88] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison–Wesley Publishing Company, Inc., 1988.

[WEG$^+$86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 14(2):358–365, June 1986.

[WJ94] S. J.E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Tech report 93/5, DEC Western Research Lab, 1994.

[WRP92] T. Wada, S. Rajan, and S. A. Pyzybylski. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

[Wu92] Y. Wu. Ordering functions for improving memory reference locality in a shared memory multiprocessor system. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):218–221, December 1992.

[YP93] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993. *Computer Architecture News,* 21(2), May 1993.

Appendix A

The SimpleScalar Architecture

Architecture Overview

The SimpleScalar architecture is derived from the MIPS-I ISA [KH92a]. The semantics are a superset with the following notable differences:

- There are no architected delay slots, *i.e.*, loads, stores, and control transfers do not execute the following instruction.
- Loads and stores support two more addressing modes: indexed and auto-increment and auto-decrement, for all data types.
- **SQRT** implements single- and double-precision floating point square roots.
- An extended 64-bit instruction encoding.

Table A.1 lists the architected registers in the SimpleScalar architecture, their hardware and software (recognized by the assembler) names, and their description. The quantity and semantics of the registers are identical to the MIPS-IV ISA.

Figure A.1 details the three instruction encodings. All instructions are 64-bits in length. The register format is used for computational instructions. The immediate format supports the inclusion of a 16-bit constant. The jump format supports specification of 24-bit jump targets. The register fields are all 8 bits, supporting future extension of the register set to 256 integer and floating point registers. Each instruction format has a fixed-location, 16-bit opcode field designed for fast instruction decoding.

The *annote* field is a 16-bit field that is written using annotations to instructions in the assembly files. The annotation interface is useful for synthesizing new instructions without having to change and recompile the assembler. Annotations are attached to the opcode, and come in two flavors: bit and field annotations. A bit annotation, written as follows:

```
lw/a       $4,4($5)
```

The annotation `/a` specifies that the first bit of the annotation field should be set. Bit annotations `/a` through `/p` are supported to set bits 0 through 15. Field annotations are written in the form:

```
lw/6:4(7)  $4,4($5)
```

Hardware Name	Software Name	Description
\$0	\$zero	zero-valued source/sink
\$1	\$at	reserved by assembler
\$2-\$3	\$v0-\$v1	fn return result regs
\$4-\$7	\$a0-\$a3	fn argument value regs
\$8-\$15	\$t0-\$t7	temp regs, caller saved
\$16-\$23	\$s0-\$s7	saved regs, callee saved
\$25-\$25	\$t8-\$t9	temp regs, caller saved
\$26-\$27	\$k0-\$k1	reserved by OS
\$28	\$gp	global pointer
\$29	\$sp	stack pointer
\$30	\$s8	saved regs, callee saved
\$31	\$ra	return address reg
\$hi	\$hi	high result register
\$lo	\$lo	low result register
\$f0-\$f31	\$f0-\$f31	floating point registers
\$fcc	\$fcc	floating point condition code

Table A.1: SimpleScalar Architecture Register Definitions

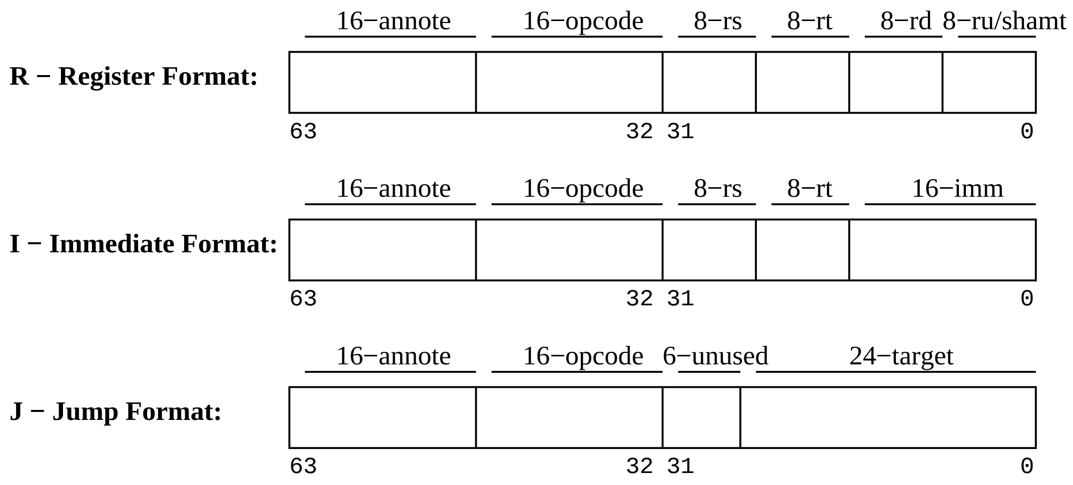


Figure A.1: SimpleScalar Architecture Instruction Formats

This annotation sets the 3-bit field starting at bit 4 to bit 6 *within* the 16-bit annotation field to the value 7.

Instruction Set Definition

This section lists all SimpleScalar instructions with their opcode, instruction and assembler formats, and semantics. The semantics are expressed as a C-style expression utilizing the extended operators and operands described in Table A.2. Operands not listed in Table A.2 refer to actual instruction fields described in Figure A.1. For each instruction, the next PC value (NPC) defaults to the current PC value plus 8 (CPC + 8) unless otherwise specified.

Operator/Operand	Semantics
FS	same as field RS
FT	same as field RT
FD	same as field RD
UIMM	IMM field unsigned-extended to word value
IMM	IMM field sign-extended to word value
OFFSET	IMM field sign-extended to word value
CPC	PC value of executing instruction
NPC	next PC value
SET_NPC(V)	Set next PC to value V
GPR(N)	General purpose register N
SET_GPR(N,V)	Set general purpose register N to value V
FPR_F(N)	Floating point register N single-precision value
SET_FPR_F(N,V)	Set floating point register N to single-precision value V
FPR_D(N)	Floating point register N double-precision value
SET_FPR_D(N,V)	Set floating point register N to double-precision value V
FPR_L(N)	Floating point register N literal word value
SET_FPR_L(N,V)	Set floating point register N to literal word value V
HI	High result register value
SET_HI(V)	Set high result register to value V
LO	Low result register value
SET_LO(V)	Set low result register to value V
READ_SIGNED_BYTE(A)	Read signed byte from address A
READ_UNSIGNED_BYTE(A)	Read unsigned byte from address A
WRITE_BYTE(V,A)	Write byte value V at address A
READ_SIGNED_HALF(A)	Read signed half from address A
READ_UNSIGNED_HALF(A)	Read unsigned half from address A
WRITE_HALF(V,A)	Write half value V at address A
READ_WORD(A)	Read word from address A
WRITE_WORD(V,A)	Write word value V at address A
TALIGN(T)	Check target T is aligned to 8 byte boundary
FPALIGN(N)	Check register N is wholly divisible by 2
OVER(X,Y)	Check for overflow when adding X to Y
UNDER(X,Y)	Check for overflow when subtraction Y from X
DIV0(V)	Check for division by zero error with divisor V

Table A.2: Operator/Operand Semantics

Control Instructions

J:	Jump to absolute address.
Opcode:	0x01
Instruction Format:	J
Assembler Format:	J target
Semantics:	`SET_NPC((CPC&0xf0000000) | (TARGET<<2)))`

JAL:	Jump to absolute address and link.
Opcode:	0x02
Instruction Format:	J
Assembler Format:	JAL target
Semantics:	`SET_NPC((CPC&0xf0000000) | (TARGET<<2))` `SET_GPR(31, CPC + 8))`

JR:	Jump to register address.
Opcode:	0x03
Instruction Format:	R
Assembler Format:	JR rs
Semantics:	`TALIGN(GPR(RS))` `SET_NPC(GPR(RS))`

JALR:	Jump to register address and link.
Opcode:	0x04
Instruction Format:	R
Assembler Format:	JALR rs
Semantics:	`TALIGN(GPR(RS))` `SET_GPR(RD, CPC + 8)` `SET_NPC(GPR(RS))`

BEQ: Branch if equal.
Opcode: 0x05
Instruction Format: I
Assembler Format: BEQ rs,rt,offset
Semantics:
```
if (GPR(RS) == GPR(RT))
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BNE: Branch if not equal.
Opcode: 0x06
Instruction Format: I
Assembler Format: BEQ rs,rt,offset
Semantics:
```
if (GPR(RS) != GPR(RT))
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BLEZ: Branch if less than or equal to zero.
Opcode: 0x07
Instruction Format: I
Assembler Format: BLEZ rs,offset
Semantics:
```
if (GPR(RS) <= 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BGTZ: Branch if greater than zero.
Opcode: 0x08
Instruction Format: I
Assembler Format: BGTZ rs,offset
Semantics:
```
if (GPR(RS) > 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BLTZ: Branch if less than zero.
Opcode: 0x09
Instruction Format: I
Assembler Format: BLTZ rs,offset
Semantics:
```
if (GPR(RS) < 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BGEZ: Branch if greater than or equal to zero.
Opcode: 0x0a
Instruction Format: I
Assembler Format: BGEZ rs,offset
Semantics:
```
if (GPR(RS) >= 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BC1F: Branch on floating point compare false.
Opcode: 0x0b
Instruction Format: I
Assembler Format: BC1F offset
Semantics:
```
if (!FCC)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BC1T: Branch on floating point compare true.
Opcode: 0x0c
Instruction Format: I
Assembler Format: BC1T offset
Semantics:
```
if (FCC)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

Load/Store Instructions

LB:	Load byte signed, displaced addressing.
Opcode:	0x20
Instruction Format:	I
Assembler Format:	LB rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_SIGNED_BYTE(GPR(RS)+OFFSET))`
LB:	Load byte signed, indexed addressing.
Opcode:	0xc0
Instruction Format:	R
Assembler Format:	LB rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_SIGNED_BYTE(GPR(RS)+GPR(RD)))`
LBU:	Load byte unsigned, displaced addressing.
Opcode:	0x22
Instruction Format:	I
Assembler Format:	LBU rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_UNSIGNED_BYTE(GPR(RS)+OFFSET))`
LBU:	Load byte unsigned, indexed addressing.
Opcode:	0xc1
Instruction Format:	R
Assembler Format:	LBU rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_UNSIGNED_BYTE(GPR(RS)+GPR(RD)))`
LH:	Load half signed, displaced addressing.
Opcode:	0x24
Instruction Format:	I
Assembler Format:	LH rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_SIGNED_HALF(GPR(RS)+OFFSET))`
LH:	Load half signed, indexed addressing.
Opcode:	0xc2
Instruction Format:	R
Assembler Format:	LH rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_SIGNED_HALF(GPR(RS)+GPR(RD)))`
LHU:	Load half unsigned, displaced addressing.
Opcode:	0x26
Instruction Format:	I
Assembler Format:	LHU rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_UNSIGNED_HALF(GPR(RS)+OFFSET))`
LHU:	Load half unsigned, indexed addressing.
Opcode:	0xc3
Instruction Format:	R
Assembler Format:	LHU rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_UNSIGNED_HALF(GPR(RS)+GPR(RD)))`
LW:	Load word, displaced addressing.
Opcode:	0x28
Instruction Format:	I
Assembler Format:	LW rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_WORD(GPR(RS)+OFFSET))`
LW:	Load word, indexed addressing.
Opcode:	0xc4
Instruction Format:	R
Assembler Format:	LW rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_WORD(GPR(RS)+GPR(RD)))`
DLW:	Double load word, displaced addressing.
Opcode:	0x29
Instruction Format:	I
Assembler Format:	DLW rt,offset(rs) inc_dec
Semantics:	`SET_GPR(RT, READ_WORD(GPR(RS)+OFFSET))`
	`SET_GPR(RT+1, READ_WORD(GPR(RS)+OFFSET+4))`
DLW:	Double load word, indexed addressing.
Opcode:	0xce
Instruction Format:	R
Assembler Format:	DLW rt,(rs+rd) inc_dec
Semantics:	`SET_GPR(RT, READ_WORD(GPR(RS)+GPR(RD)))`
	`SET_GPR(RT+1, READ_WORD(GPR(RS)+GPR(RD)+4))`
L.S:	Load word into floating point register file, displaced addressing.
Opcode:	0x2a
Instruction Format:	I

Assembler Format:	L.S ft,offset(rs) inc_dec
Semantics:	`SET_FPR_L(FT, READ_WORD(GPR(RS)+OFFSET))`

L.S:	Load word into floating point register file, indexed addressing.
Opcode:	0xc5
Instruction Format:	R
Assembler Format:	L.S ft,(rs+rd) inc_dec
Semantics:	`SET_FPR_L(RT, READ_WORD(GPR(RS)+GPR(RD)))`

L.D:	Load double word into floating point register file, displaced addressing.
Opcode:	0x2b
Instruction Format:	I
Assembler Format:	L.D ft,offset(rs) inc_dec
Semantics:	`SET_FPR_L(FT, READ_WORD(GPR(RS)+OFFSET))` `SET_FPR_L(FT+1, READ_WORD(GPR(RS)+OFFSET+4))`

L.D:	Load double word into floating point register file, indexed addressing.
Opcode:	0xcf
Instruction Format:	R
Assembler Format:	L.D ft,(rs+rd) inc_dec
Semantics:	`SET_FPR_L(RT, READ_WORD(GPR(RS)+GPR(RD)))` `SET_FPR_L(RT+1, READ_WORD(GPR(RS)+GPR(RD)+4))`

LWL:	Load word left, displaced addressing.
Opcode:	0x2c
Instruction Format:	I
Assembler Format:	LWL offset(rs)
Semantics:	See `ss.def` or [KH92a] for a detailed description of this instruction's semantics. NOTE: LWL does not support pre-/post- inc/dec.

LWR:	Load word right, displaced addressing.
Opcode:	0x2d
Instruction Format:	I
Assembler Format:	LWR offset(rs)
Semantics:	See `ss.def` or [KH92a] for a detailed description of this instruction's semantics. NOTE: LWR does not support pre-/post- inc/dec.

SB:	Store byte, displaced addressing.
Opcode:	0x30
Instruction Format:	I
Assembler Format:	SB rt,offset(rs) inc_dec
Semantics:	`WRITE_BYTE(GPR(RT), GPR(RS)+OFFSET)`

SB:	Store byte, indexed addressing.
Opcode:	0xc6
Instruction Format:	R
Assembler Format:	SB rt,(rs+rd) inc_dec
Semantics:	`WRITE_BYTE(GPR(RT), GPR(RS)+GPR(RD))`

SH:	Store half, displaced addressing.
Opcode:	0x32
Instruction Format:	I
Assembler Format:	SH rt,offset(rs) inc_dec
Semantics:	`WRITE_HALF(GPR(RT), GPR(RS)+OFFSET)`

SH:	Store half, indexed addressing.
Opcode:	0xc7
Instruction Format:	R
Assembler Format:	SH rt,(rs+rd) inc_dec
Semantics:	`WRITE_HALF(GPR(RT), GPR(RS)+GPR(RD))`

SW:	Store word, displaced addressing.
Opcode:	0x34
Instruction Format:	I
Assembler Format:	SW rt,offset(rs) inc_dec
Semantics:	`WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)`

SW:	Store word, indexed addressing.
Opcode:	0xc8
Instruction Format:	R
Assembler Format:	SW rt,(rs+rd) inc_dec
Semantics:	`WRITE_WORD(GPR(RT), GPR(RS)+GPR(RD))`

DSW:	Double store word, displaced addressing.
Opcode:	0x35
Instruction Format:	I
Assembler Format:	DSW rt,offset(rs) inc_dec

Semantics: `WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)`
`WRITE_WORD(GPR(RT+1), GPR(RS)+OFFSET+4)`

DSW: Double store word, indexed addressing.
Opcode: 0xd0
Instruction Format: R
Assembler Format: DSW rt,(rs+rd) inc_dec
Semantics: `WRITE_WORD(GPR(RT), GPR(RS)+GPR(RD))`
`WRITE_WORD(GPR(RT+1), GPR(RS)+GPR(RD)+4)`

DSZ: Double store zero, displaced addressing.
Opcode: 0x38
Instruction Format: I
Assembler Format: DSW rt,offset(rs) inc_dec
Semantics: `WRITE_WORD(0, GPR(RS)+OFFSET)`
`WRITE_WORD(0, GPR(RS)+OFFSET+4)`

DSZ: Double store zero, indexed addressing.
Opcode: 0xd1
Instruction Format: R
Assembler Format: DSW rt,(rs+rd) inc_dec
Semantics: `WRITE_WORD(0, GPR(RS)+GPR(RD))`
`WRITE_WORD(0, GPR(RS)+GPR(RD)+4)`

S.S: Store word from floating point register file, displaced addressing.
Opcode: 0x36
Instruction Format: I
Assembler Format: S.S ft,offset(rs) inc_dec
Semantics: `WRITE_WORD(FPR_L(FT), GPR(RS)+OFFSET)`

S.S: Store word from floating point register file, indexed addressing.
Opcode: 0xc9
Instruction Format: R
Assembler Format: S.S ft,(rs+rd) inc_dec
Semantics: `WRITE_WORD(FPR_L(FT), GPR(RS)+GPR(RD))`

S.D: Store double word from floating point register file, displaced addressing.
Opcode: 0x37
Instruction Format: I
Assembler Format: S.D ft,offset(rs) inc_dec
Semantics: `WRITE_WORD(FPR_L(FT), GPR(RS)+OFFSET)`
`WRITE_WORD(FPR_L(FT+1), GPR(RS)+OFFSET+4)`

S.D: Store double word from floating point register file, indexed addressing.
Opcode: 0xd2
Instruction Format: R
Assembler Format: S.D ft,(rs+rd) inc_dec
Semantics: `WRITE_WORD(FPR_L(FT), GPR(RS)+GPR(RD))`
`WRITE_WORD(FPR_L(FT+1), GPR(RS)+GPR(RD)+4)`

SWL: Store word left, displaced addressing.
Opcode: 0x39
Instruction Format: I
Assembler Format: SWL rt,offset(rs)
Semantics: See `ss.def` or [KH92a] for a detailed description of this instruction's semantics. NOTE: SWL does not support pre-/post- inc/dec.

SWR: Store word right, displaced addressing.
Opcode: 0x3a
Instruction Format: I
Assembler Format: SWR rt,offset(rs)
Semantics: See `ss.def` or [KH92a] for a detailed description of this instruction's semantics. NOTE: SWR does not support pre-/post- inc/dec.

Integer Instructions

ADD: Add signed (with overflow check).
Opcode: 0x40
Instruction Format: R
Assembler Format: ADD rd,rs,rt
Semantics: `OVER(GPR(RT),GPR(RT))`
`SET_GPR(RD, GPR(RS) + GPR(RT))`

ADDI: Add immediate signed (with overflow check).
Opcode: 0x41
Instruction Format: I

Assembler Format:	ADDI rd,rs,rt
Semantics:	`OVER(GPR(RS),IMM)`
	`SET_GPR(RT, GPR(RS) + IMM)`
ADDU:	Add unsigned (no overflow check).
Opcode:	0x42
Instruction Format:	R
Assembler Format:	ADDU rd,rs,rt
Semantics:	`SET_GPR(RD, GPR(RS) + GPR(RT))`
ADDIU:	Add immediate unsigned (no overflow check).
Opcode:	0x43
Instruction Format:	I
Assembler Format:	ADDIU rd,rs,rt
Semantics:	`SET_GPR(RT, GPR(RS) + IMM)`
SUB:	Subtract signed (with underflow check).
Opcode:	0x44
Instruction Format:	R
Assembler Format:	SUB rd,rs,rt
Semantics:	`UNDER(GPR(RS),GPR(RT))`
	`SET_GPR(RD, GPR(RS) - GPR(RT))`
SUBU:	Subtract unsigned (without underflow check).
Opcode:	0x45
Instruction Format:	R
Assembler Format:	SUBU rd,rs,rt
Semantics:	`SET_GPR(RD, GPR(RS) - GPR(RT))`
MULT:	Multiply signed.
Opcode:	0x46
Instruction Format:	R
Assembler Format:	MULT rs,rt
Semantics:	`SET_HI((RS * RT) / (1<<32))`
	`SET_LO((RS * RT) % (1<<32))`
MULTU:	Multiply unsigned.
Opcode:	0x47
Instruction Format:	R
Assembler Format:	MULTU rs,rt
Semantics:	`SET_HI(((unsigned)RS*(unsigned)RT)/(1<<32))`
	`SET_LO(((unsigned)RS*(unsigned)RT)%(1<<32))`
DIV:	Divide signed.
Opcode:	0x48
Instruction Format:	R
Assembler Format:	DIV rs,rt
Semantics:	`DIV0(GPR(RT))`
	`SET_LO(GPR(RS) / GPR(RT))`
	`SET_HI(GPR(RS) % GPR(RT))`
DIVU:	Divide unsigned.
Opcode:	0x49
Instruction Format:	R
Assembler Format:	DIVU rs,rt
Semantics:	`DIV0(GPR(RT))`
	`SET_LO((unsigned)GPR(RS)/(unsigned)GPR(RT))`
	`SET_HI((unsigned)GPR(RS)%(unsigned)GPR(RT))`
MFHI:	Move from HI register.
Opcode:	0x4a
Instruction Format:	R
Assembler Format:	MFHI rd
Semantics:	`SET_GPR(RD, HI)`
MTHI:	Move to HI register.
Opcode:	0x4b
Instruction Format:	R
Assembler Format:	MTHI rs
Semantics:	`SET_HI(GPR(RS))`
MFLO:	Move from LO register.
Opcode:	0x4c
Instruction Format:	R
Assembler Format:	MFLO rd
Semantics:	`SET_GPR(RD, LO)`
MTLO:	Move to LO register.

Opcode:	0x4d
Instruction Format:	R
Assembler Format:	MTLO rs
Semantics:	`SET_LO(GPR(RS))`
AND:	Logical AND.
Opcode:	0x4e
Instruction Format:	R
Assembler Format:	AND rd,rs,rt
Semantics:	`SET_GPR(RD, GPR(RS) & GPR(RT))`
ANDI:	Logical AND immediate.
Opcode:	0x4f
Instruction Format:	I
Assembler Format:	ANDI rd,rt,imm
Semantics:	`SET_GPR(RT, GPR(RS) & UIMM)`
OR:	Logical OR.
Opcode:	0x50
Instruction Format:	R
Assembler Format:	OR rd,rs,rt
Semantics:	`SET_GPR(RD, GPR(RS) | GPR(RT))`
ORI:	Logical OR immediate.
Opcode:	0x51
Instruction Format:	I
Assembler Format:	ORI rd,rt,imm
Semantics:	`SET_GPR(RT, GPR(RS) | UIMM)`
XOR:	Logical XOR.
Opcode:	0x52
Instruction Format:	R
Assembler Format:	XOR rd,rs,rt
Semantics:	`SET_GPR(RD, GPR(RS) ^ GPR(RT))`
XORI:	Logical XOR immediate.
Opcode:	0x53
Instruction Format:	I
Assembler Format:	ORI rd,rt,uimm
Semantics:	`SET_GPR(RT, GPR(RS) ^ UIMM)`
NOR:	Logical NOR.
Opcode:	0x54
Instruction Format:	R
Assembler Format:	NOR rd,rs,rt
Semantics:	`SET_GPR(RD,  (GPR(RS) | GPR(RT)))`
SLL:	Shift left logical.
Opcode:	0x55
Instruction Format:	R
Assembler Format:	SLL rd,rt,shamt
Semantics:	`SET_GPR(RD, GPR(RT) << SHAMT)`
SLLV:	Shift left logical variable.
Opcode:	0x56
Instruction Format:	R
Assembler Format:	SLLV rd,rt,rs
Semantics:	`SET_GPR(RD, GPR(RT) << (GPR(RS) & 0x1f))`
SRL:	Shift right logical.
Opcode:	0x57
Instruction Format:	R
Assembler Format:	SRL rd,rt,shamt
Semantics:	`SET_GPR(RD, GPR(RT) >> SHAMT)`
SRLV:	Shift right logical variable.
Opcode:	0x58
Instruction Format:	R
Assembler Format:	SRLV rd,rt,rs
Semantics:	`SET_GPR(RD, GPR(RT) << (GPR(RS) & 0x1f))`
SRA:	Shift right arithmetic.
Opcode:	0x59
Instruction Format:	R
Assembler Format:	SRA rd,rt,shamt
Semantics:	`SET_GPR(RD, SEX(GPR(RT) >> SHAMT, 31 - SHAMT))`
SRAV:	Shift right arithmetic variable.
Opcode:	0x59

Instruction Format:	R
Assembler Format:	SRAV rd,rt,rs
Semantics:	`SET_GPR(RD, SEX(GPR(RT) >> SHAMT, 31 - (GPR(RD) & 0x1f)))`
SLT:	Set register if less than.
Opcode:	0x5b
Instruction Format:	R
Assembler Format:	SLT rd,rs,rt
Semantics:	`SET_GPR(RD, (GPR(RS) < GPR(RT)) ? 1 :  0)`
SLTI:	Set register if less than immediate.
Opcode:	0x5c
Instruction Format:	I
Assembler Format:	SLTI rd,rs,imm
Semantics:	`SET_GPR(RD, (GPR(RS) < IMM) ? 1 :  0)`
SLTU:	Set register if less than unsigned.
Opcode:	0x5d
Instruction Format:	R
Assembler Format:	SLTU rd,rs,rt
Semantics:	`SET_GPR(RD, ((unsigned)GPR(RS)<(unsigned)GPR(RT)) ? 1 :  0)`
SLTIU:	Set register if less than unsigned immediate.
Opcode:	0x5d
Instruction Format:	I
Assembler Format:	SLTIU rd,rs,imm
Semantics:	`SET_GPR(RD, ((unsigned)GPR(RS)<(unsigned)GPR(RT)) ? 1 :  0)`

Floating Point Instructions

ADD.S:	Add floating point, single precision.
Opcode:	0x70
Instruction Format:	R
Assembler Format:	ADD.S fd,fs,ft

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FPR_F(FD, FPR_F(FS) + FPR_F(FT)))`

ADD.D:	Add floating point, double-precision.
Opcode:	0x71
Instruction Format:	R
Assembler Format:	ADD.D fd,fs,ft

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FPR_D(FD, FPR_D(FS) + FPR_D(FT)))`

SUB.S:	Subtract floating point, single precision.
Opcode:	0x72
Instruction Format:	R
Assembler Format:	SUB.S fd,fs,ft

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FPR_F(FD, FPR_F(FS) - FPR_F(FT)))`

SUB.D:	Subtract floating point, double precision.
Opcode:	0x73
Instruction Format:	R
Assembler Format:	SUB.D fd,fs,ft

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FPR_D(FD, FPR_D(FS) - FPR_D(FT)))`

MUL.S:	Multiply floating point, single precision.
Opcode:	0x74
Instruction Format:	R
Assembler Format:	MUL.S fd,fs,ft

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FPR_F(FD, FPR_F(FS) * FPR_F(FT)))`

MUL.D: Multiply floating point, double precision.
Opcode: 0x75
Instruction Format: R
Assembler Format: MUL.D fd,fs,ft
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_D(FD, FPR_D(FS) * FPR_D(FT)))
```

DIV.S: Divide floating point, single precision.
Opcode: 0x76
Instruction Format: R
Assembler Format: DIV.S fd,fs,ft
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
DIV0(FPR_F(FT))
SET_FPR_F(FD, FPR_F(FS) / FPR_F(FT)))
```

DIV.D: Divide floating point, double precision.
Opcode: 0x77
Instruction Format: R
Assembler Format: DIV.D fd,fs,ft
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
DIV0(FPR_D(FT))
SET_FPR_D(FD, FPR_D(FS) / FPR_D(FT)))
```

ABS.S: Absolute value, single precision.
Opcode: 0x78
Instruction Format: R
Assembler Format: ABS.S fd,fs
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, fabs((double)FPR_F(FS))))
```

ABS.D: Absolute value, double precision.
Opcode: 0x79
Instruction Format: R
Assembler Format: ABS.D fd,fs
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, fabs(FPR_D(FS))))
```

MOV.S: Move floating point value, single precision.
Opcode: 0x7a
Instruction Format: R
Assembler Format: MOV.S fd,fs
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, FPR_F(FS))
```

MOV.D: Move floating point value, double precision.
Opcode: 0x7b
Instruction Format: R
Assembler Format: MOV.D fd,fs
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, FPR_D(FS))
```

NEG.S: Negate floating point value, single precistion.
Opcode: 0x7c
Instruction Format: R
Assembler Format: NEG.S fd,fs
Semantics:
```
FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, -FPR_F(FS))
```

NEG.D: Negate floating point value, double precistion.
Opcode: 0x7d
Instruction Format: R
Assembler Format: NEG.D fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_D(FD, -FPR_D(FS))`

CVT.S.D:	Convert double precision to single precision.
Opcode:	0x80
Instruction Format:	R
Assembler Format:	CVT.S.D fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_D(FD, -FPR_D(FS))`

CVT.S.W:	Convert integer to single precision.
Opcode:	0x81
Instruction Format:	R
Assembler Format:	CVT.S.W fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_F(FD, (float)FPR_L(FS))`

CVT.D.S:	Convert single precision to double precision.
Opcode:	0x82
Instruction Format:	R
Assembler Format:	CVT.D.S fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_D(FD, (double)FPR_F(FS))`

CVT.D.W:	Convert integer to double precision.
Opcode:	0x83
Instruction Format:	R
Assembler Format:	CVT.D.W fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_D(FD, (double)FPR_L(FS))`

CVT.W.S:	Convert single precision to integer.
Opcode:	0x84
Instruction Format:	R
Assembler Format:	CVT.W.S fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_L(FD, (long)FPR_F(FS))`

CVT.W.D:	Convert double precision to integer.
Opcode:	0x85
Instruction Format:	R
Assembler Format:	CVT.W.D fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_L(FD, (long)FPR_D(FS))`

C.EQ.S:	Test if equal, single precision.
Opcode:	0x90
Instruction Format:	R
Assembler Format:	C.EQ.S fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_F(FS) == FPR_F(FT))`

C.EQ.D:	Test if equal, double precision.
Opcode:	0x91
Instruction Format:	R
Assembler Format:	C.EQ.D fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_D(FS) == FPR_D(FT))`

C.LT.S:	Test if less than, single precision.
Opcode:	0x92
Instruction Format:	R
Assembler Format:	C.LT.S fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_F(FS) < FPR_F(FT))`

C.LT.D:	Test if less than, double precision.
Opcode:	0x93
Instruction Format:	R
Assembler Format:	C.LT.D fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_D(FS) < FPR_D(FT))`

C.LE.S:	Test if less than or equal, single precision.
Opcode:	0x94
Instruction Format:	R
Assembler Format:	C.LE.S fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_F(FS) <= FPR_F(FT))`

C.LE.D:	Test if less than or equal, double precision.
Opcode:	0x95
Instruction Format:	R
Assembler Format:	C.LE.D fs,ft

Semantics:	`FPALIGN(FS)`
	`FPALIGN(FT)`
	`SET_FCC(FPR_D(FS) <= FPR_D(FT))`

SQRT.S:	Square root, single precision.
Opcode:	0x96
Instruction Format:	R
Assembler Format:	SQRT.S fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_F(FD, sqrt((double)FPR_F(FS)))`

SQRT.D:	Square root, double precision.
Opcode:	0x97
Instruction Format:	R
Assembler Format:	SQRT.D fd,fs

Semantics:	`FPALIGN(FD)`
	`FPALIGN(FS)`
	`SET_FPR_D(FD, sqrt(FPR_D(FS)))`

Miscellaneous Instructions

NOP:	No operation.
Opcode:	0x00
Instruction Format:	R
Assembler Format:	NOP
Semantics:	

SYSCALL:	System call.
Opcode:	0xa0
Instruction Format:	R
Assembler Format:	SYSCALL
Semantics:	See Section A.

BREAK:	Declare a program error.
Opcode:	0xa1
Instruction Format:	I
Assembler Format:	BREAK uimm
Semantics:	Actions are simulator-dependent. Typically, an error message is printed and `abort()` is called.

LUI:	Load upper immediate.
Opcode:	0xa2
Instruction Format:	I
Assembler Format:	LUI uimm
Semantics:	`SET_GPR(RT, UIMM << 16)`

MFC1:	Move from floating point to integer register file.
Opcode:	0xa3
Instruction Format:	R
Assembler Format:	MFC1 rt,fs
Semantics:	`SET_GPR(RT, FPR_L(FS))`

MTC1:	Move from integer to floating point register file.
Opcode:	0xa5
Instruction Format:	R
Assembler Format:	MTC1 rt,fs
Semantics:	`SET_FPR_L(FS, GPR(RT))`

System Call Definitions

This section lists all system calls supported by the simulators with their system call code (syscode), interface specification, and appropriate POSIX Unix reference. Systems calls are initiated with the SYSCALL instruction. Prior to execution of a SYSCALL instruction, register $\$v0$ should be loaded with the system call code. The arguments of the system call interface should be loaded into registers $\$a0 - \$a3$ in the order specified by the system call interface prototype, *e.g.*, for `read`(int fd, char *buf, int nbyte), 0x03 is loaded into $\$v0$, `fd` is loaded into $\$a0$, `buf` into $\$a1$, and `nbyte` into $\$a2$.

EXIT:	Exit process.
Syscode:	0x01
Interface:	`void exit(int status);`
Semantics:	See exit(2).
READ:	Read from file to buffer.
Syscode:	0x03
Interface:	`int read(int fd, char *buf, int nbyte);`
Semantics:	See read(2).
WRITE:	Write from a buffer to a file.
Syscode:	0x04
Interface:	`int write(int fd, char *buf, int nbyte);`
Semantics:	See write(2).
OPEN:	Open a file.
Syscode:	0x05
Interface:	`int open(char *fname, int flags, int mode);`
Semantics:	See open(2).
CLOSE:	Close a file.
Syscode:	0x06
Interface:	`int close(int fd);`
Semantics:	See close(2).
CREAT:	Create a file.
Syscode:	0x08
Interface:	`int creat(char *fname, int mode);`
Semantics:	See creat(2).
UNLINK:	Delete a file.
Syscode:	0x0a
Interface:	`int unlink(char *fname);`
Semantics:	See unlink(2).
CHDIR:	Change process directory.
Syscode:	0x0c
Interface:	`int chdir(char *path);`
Semantics:	See chdir(2).
CHMOD:	Change file permissions.
Syscode:	0x0f
Interface:	`int chmod(int *fname, int mode);`
Semantics:	See chmod(2).
CHOWN:	Change file owner and group.
Syscode:	0x10
Interface:	`int chown(char *fname, int owner, int group);`
Semantics:	See chown(2).
BRK:	Change process break address.
Syscode:	0x11
Interface:	`int brk(long addr);`
Semantics:	See brk(2).
LSEEK:	Move file pointer.
Syscode:	0x13
Interface:	`long lseek(int fd, long offset, int whence);`
Semantics:	See lseek(2).
GETPID:	Get process identifier.
Syscode:	0x14
Interface:	`int getpid(void);`

Semantics: See getpid(2).

GETUID: Get user identifier.
Syscode: 0x18
Interface: `int getuid(void);`
Semantics: See getuid(2).

ACCESS: Determine accessibility of a file.
Syscode: 0x21
Interface: `int access(char *fname, int mode);`
Semantics: See access(2).

STAT: Get file status.
Syscode: 0x26
Interface:

```
struct stat
{
    short          st_dev;
    long           st_ino;
    unsigned short st_mode;
    short          st_nlink;
    short          st_uid;
    short          st_gid;
    short          st_rdev;
    int            st_size;
    int            st_atime;
    int            st_spare1;
    int            st_mtime;
    int            st_spare2;
    int            st_ctime;
    int            st_spare3;
    long           st_blksize;
    long           st_blocks;
    long           st_gennum;
    long           st_spare4;
};
int stat(char *fname, struct stat *buf);
```

Semantics: See stat(2).

LSTAT: Get file status (and don't dereference links).
Syscode: 0x28
Interface: `int lstat(char *fname, struct stat *buf);`
Semantics: See lstat(2).

DUP: Duplicate a file descriptor.
Syscode: 0x29
Interface: `int dup(int fd);`
Semantics: See dup(2).

PIPE: Create an interprocess communication channel.
Syscode: 0x2a
Interface: `int pipe(int fd[2]);`
Semantics: See pipe(2).

GETGID: Get group identifier.
Syscode: 0x2f
Interface: `int getgid(void);`
Semantics: See getgid(2).

IOCTL: Device control interface.
Syscode: 0x36
Interface: `int ioctl(int fd, int request, char *arg);`
Semantics: See ioctl(2).

FSTAT: Get file descriptor status.
Syscode: 0x3e
Interface: `int fstat(int fd, struct stat *buf);`
Semantics: See fstat(2).

GETPAGESIZE: Get page size.
Syscode: 0x40
Interface: `int getpagesize(void);`
Semantics: See getpagesize(2).

GETDTABLESIZE: Get file descriptor table size.
Syscode: 0x59
Interface: `int getdtablesize(void);`
Semantics: See getdtablesize(2).

DUP2: Duplicate a file descriptor.
Syscode: 0x5a

Interface: `int dup2(int fd1, int fd2);`
Semantics: See dup2(2).

FCNTL: File control.
Syscode: 0x5c
Interface: `int fcntl(int fd, int cmd, int arg);`
Semantics: See fcntl(2).

SELECT: Synchronous I/O multiplexing.
Syscode: 0x5d
Interface:
```
int select (int width, fd_set *readfds,
fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```
Semantics: See select(2).

GETTIMEOFDAY: Get the date and time.
Syscode: 0x74
Interface:
```
struct timeval {
    long     tv_sec;
    long     tv_usec;
};
struct int {
    timezone tz_minuteswest;
    int      tz_dsttime;
};
int gettimeofday(struct timeval *tp,
struct timezone *tzp);
```
Semantics: See gettimeofday(2).

WRITEV: Write output, vectored.
Syscode: 0x79
Interface: `int writev(int fd, struct iovec *iov, int cnt);`
Semantics: See writev(2).

UTIMES: Set file times.
Syscode: 0x8a
Interface: `int utimes(char *file, struct timeval *tvp);`
Semantics: See utimes(2).

GETRLIMIT: Get maximum resource consumption.
Syscode: 0x90
Interface: `int getrlimit(int res, struct rlimit *rlp);`
Semantics: See getrlimit(2).

SETRLIMIT: Set maximum resource consumption.
Syscode: 0x91
Interface: `int setrlimit(int res, struct rlimit *rlp);`
Semantics: See setrlimit(2).

Appendix B

Detailed Results

Continued on next page.

	Multi-ported				Multi-level/Pretranslated				Interleaved			Piggybacked		
Program	T4-IPC	T4	T2	T1	M16	M8	M4	P8	I8	I4	X4	PB2	PB1	I4/PB
Compress	1.962	1.000	0.941	0.683	0.908	0.891	0.871	0.953	0.879	0.861	0.862	0.998	0.943	0.994
Doduc	1.757	1.000	0.962	0.848	1.000	0.994	0.985	0.984	0.935	0.929	0.934	0.997	0.974	0.996
Espresso	2.900	1.000	0.934	0.699	1.000	1.000	0.981	0.983	0.893	0.868	0.874	0.994	0.916	0.986
GCC	1.869	1.000	0.910	0.660	1.000	0.977	0.925	0.990	0.883	0.857	0.851	0.990	0.888	0.984
Ghostscript	2.178	1.000	0.959	0.819	0.999	0.990	0.972	0.943	0.931	0.921	0.954	0.999	0.974	0.989
MPEG_play	2.822	1.000	0.908	0.692	0.958	0.937	0.850	0.880	0.876	0.846	0.843	0.972	0.865	0.980
Perl	1.434	1.000	0.937	0.734	0.997	0.972	0.925	0.976	0.939	0.905	0.906	0.992	0.904	0.983
TFFT	1.790	1.000	0.957	0.847	0.991	0.989	0.986	1.000	0.954	0.950	0.946	0.990	0.969	0.989
Tomcatv	2.721	1.000	0.960	0.787	1.000	0.994	0.955	0.908	0.935	0.934	0.974	0.998	0.965	0.996
Xlisp	2.523	1.000	0.876	0.551	0.999	1.000	0.972	0.987	0.834	0.806	0.815	0.992	0.861	0.986
RTW Avg	2.094	1.000	0.940	0.767	0.994	0.988	0.965	0.972	0.916	0.902	0.909	0.993	0.939	0.990

Table B.1: Relative Performance on Baseline Simulator. Results shown are run-time weighted average IPCs normalized to the performance of design T4. All experiments were run on an 8-way out-of-order issue processor simulator with 32 registers and 4k pages.

	Fully-Associative						Set-Associative (128 entries)			
Program	4 entries	8 entries	16 entries	32 entries	64 entries	128 entries	8-way	16-way	32-way	64-way
Compress	18.8	16.0	13.4	9.8	4.0	0.0	0.0	0.0	0.0	0.0
Doduc	7.7	3.9	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Espresso	3.5	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
GCC	11.5	3.7	0.8	0.1	0.0	0.0	0.0	0.0	0.0	0.0
Ghostscript	8.0	3.3	0.3	0.1	0.1	0.0	0.0	0.0	0.0	0.0
MPEG_play	37.7	14.4	12.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
Perl	12.8	5.4	1.3	0.3	0.1	0.0	0.0	0.0	0.0	0.0
TFFT	23.3	5.3	0.1	0.1	0.1	0.1	2.7	0.1	0.1	0.1
Tomcatv	15.8	3.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Xlisp	4.4	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RTW Avg	13.8	4.6	1.4	0.2	0.1	0.0	0.6	0.0	0.0	0.0

Table B.2: TLB miss rates. All values shown indicate percent of all references that miss. The row labeled RTW Avg is the run-time weighted average miss rate over all the benchmarks (weighted by the run-time of configuration T4). The 4, 8, and 16 entry TLBs are managed with LRU replacement, and the 32, 64, and 128 entry TLBs are managed with random replacement.

				8-Way In-order Issue, 4k page					8-Way Out-of-Order Issue, 8k page				
	Insts	Loads	Stores	Inst/Cycle		(Ld+St)/Cycle		Br Pred	Inst/Cycle		(Ld+St)/Cycle		Br Pred
Program	(Mil.)	(Mil.)	(Mil.)	Issue	C'mit	Issue	C'mit	Rate (%)	Issue	C'mit	Issue	C'mit	Rate (%)
Compress	62.0	15.8	6.1	1.31	1.23	0.45	0.43	89.4	3.65	1.96	1.30	0.69	89.7
Doduc	1,375.1	330.4	130.2	1.01	0.97	0.33	0.32	86.4	2.16	1.76	0.71	0.59	86.7
Espresso	517.5	116.5	32.7	1.50	1.41	0.43	0.41	88.8	4.48	2.90	1.32	0.84	90.2
GCC	110.6	26.4	16.5	1.55	1.27	0.56	0.49	78.8	3.56	1.87	1.32	0.72	80.2
Ghostscript	625.2	109.1	53.3	1.36	1.30	0.35	0.33	94.3	2.76	2.18	0.73	0.55	93.3
MPEG_play	529.6	114.9	47.9	1.36	1.29	0.42	0.40	84.6	4.11	2.83	1.17	0.86	86.0
Perl	231.5	57.7	37.2	1.32	1.10	0.52	0.44	79.4	2.85	1.43	1.10	0.57	81.2
TFFT	959.8	136.6	89.4	1.17	1.04	0.27	0.24	84.5	2.70	1.79	0.62	0.42	79.0
Tomcatv	359.7	90.9	18.3	1.12	1.09	0.34	0.33	86.0	3.67	2.73	1.00	0.83	86.6
Xlisp	962.7	289.2	171.6	1.55	1.42	0.73	0.68	87.2	4.17	2.52	1.86	1.21	88.0

Table B.3: Program Execution Performance. Instruction, load, and store counts include only non-speculative operations. The columns labeled *Issue* and *C'mit* indicate the average number of operations issued and committed per cycle, respectively.

	Multi-ported				Multi-level/Pretranslated				Interleaved			Piggybacked		
Program	T4-IPC	T4	T2	T1	M16	M8	M4	P8	I8	I4	X4	PB2	PB1	I4/PB
Compress	1.230	1.000	0.985	0.924	0.981	0.975	0.959	0.967	0.930	0.930	0.931	1.000	1.000	1.000
Doduc	0.966	1.000	0.992	0.945	0.997	0.981	0.937	0.960	0.964	0.963	0.974	0.996	0.994	0.995
Espresso	1.412	1.000	0.975	0.897	1.000	0.998	0.981	0.922	0.952	0.942	0.948	0.999	0.946	0.988
GCC	1.273	1.000	0.980	0.902	0.995	0.975	0.928	0.946	0.936	0.931	0.934	0.999	0.985	0.996
Ghostscript	1.303	1.000	0.992	0.952	0.998	0.984	0.961	0.964	0.961	0.959	0.961	1.000	0.993	0.995
MPEG_play	1.286	1.000	0.984	0.885	0.951	0.937	0.859	0.941	0.958	0.949	0.957	0.992	0.919	0.998
Perl	1.098	1.000	0.986	0.929	0.992	0.972	0.936	0.958	0.959	0.949	0.966	0.999	0.987	0.994
TFFT	1.040	1.000	0.991	0.983	0.999	0.996	0.968	0.984	0.984	0.977	0.982	1.000	0.992	1.000
Tomcatv	1.094	1.000	1.000	0.972	1.000	1.000	0.931	0.951	0.995	0.994	0.904	1.000	0.976	0.999
Xlisp	1.419	1.000	0.975	0.893	1.000	0.992	0.936	0.988	0.937	0.937	0.941	0.999	0.994	1.000
RTW Avg	1.156	1.000	0.988	0.937	0.994	0.984	0.935	0.959	0.963	0.959	0.960	0.998	0.982	0.997

Table B.4: Relative Performance with In-order Issue.

	Multi-ported				Multi-level/Pretranslated				Interleaved			Piggybacked		
Program	T4-IPC	T4	T2	T1	M16	M8	M4	P8	I8	I4	X4	PB2	PB1	I4/PB
Compress	1.962	1.000	0.941	0.682	0.928	0.900	0.874	0.955	0.879	0.833	0.805	0.998	0.944	0.992
Doduc	1.756	1.000	0.991	0.867	1.000	1.000	1.000	0.997	0.959	0.944	0.944	1.000	0.996	0.997
Espresso	2.899	1.000	0.934	0.699	0.997	1.000	1.000	0.984	0.885	0.863	0.868	1.000	0.952	0.996
GCC	1.868	1.000	0.911	0.660	1.000	0.990	0.939	0.992	0.873	0.847	0.844	0.992	0.891	0.982
Ghostscript	2.179	1.000	0.959	0.819	0.999	0.994	0.978	0.970	0.941	0.925	0.935	0.999	0.987	0.997
MPEG_play	2.827	1.000	0.907	0.691	0.978	0.974	0.872	0.898	0.875	0.852	0.849	0.994	0.886	1.000
Perl	1.435	1.000	0.936	0.733	1.000	0.980	0.933	0.990	0.939	0.914	0.921	0.993	0.907	0.981
TFFT	1.792	1.000	0.957	0.846	0.991	0.988	0.987	1.000	0.953	0.934	0.933	0.990	0.968	0.996
Tomcatv	2.734	1.000	0.958	0.783	1.000	0.992	0.959	0.943	0.939	0.937	0.972	0.994	0.987	0.997
Xlisp	2.523	1.000	0.875	0.551	0.999	0.999	0.977	0.989	0.839	0.707	0.707	0.992	0.862	0.969
RTW Avg	2.095	1.000	0.948	0.772	0.996	0.992	0.975	0.982	0.924	0.890	0.893	0.996	0.952	0.992

Table B.5: Relative Performance with 8k Pages.

				8-Way Out-of-order Issue, 8 int/8 fp registers				
	Insts	Loads	Stores	Inst/Cycle		(Ld+St)/Cycle		Br Pred
Program	(Mil.)	(Mil.)	(Mil.)	Issue	C'mit	Issue	C'mit	Rate (%)
Compress	72.9	18.6	6.1	3.64	2.12	1.22	0.72	90.2
Doduc	1,616.4	457.1	227.0	2.35	1.93	0.97	0.82	86.4
Espresso	621.1	182.2	65.1	4.46	3.38	1.78	1.33	92.5
GCC	119.5	33.1	19.8	3.64	1.96	1.57	0.87	80.8
Ghostscript	651.5	126.9	59.1	3.05	1.96	0.96	0.56	89.5
MPEG_play	704.3	225.1	101.6	4.31	2.80	1.83	1.30	81.1
Perl	241.0	61.6	42.4	2.86	1.46	1.23	0.63	81.4
TFFT	1,350.0	402.4	181.6	3.04	2.24	1.15	0.97	79.4
Tomcatv	1,969.8	355.3	131.7	5.47	5.44	1.35	1.35	89.3
Xlisp	946.6	280.0	165.3	4.22	2.59	1.87	1.20	89.3

Table B.6: Program Execution Performance.

	Multi-ported				Multi-level/Pretranslated				Interleaved			Piggybacked		
Program	T4-IPC	T4	T2	T1	M16	M8	M4	P8	I8	I4	X4	PB2	PB1	I4/PB
Compress	2.120	1.000	0.947	0.698	0.932	0.913	0.887	0.967	0.900	0.883	0.888	0.997	0.929	0.994
Doduc	1.930	1.000	0.948	0.759	0.999	0.994	0.986	0.980	0.893	0.878	0.883	0.997	0.949	0.993
Espresso	3.383	1.000	0.841	0.540	1.000	0.997	0.949	0.815	0.814	0.728	0.731	0.964	0.799	0.946
GCC	1.956	1.000	0.885	0.610	1.000	0.979	0.924	0.890	0.855	0.827	0.819	0.985	0.857	0.980
Ghostscript	1.959	1.000	0.938	0.800	1.000	0.988	0.967	0.959	0.904	0.893	0.910	0.996	0.938	0.983
MPEG_play	2.796	1.000	0.854	0.566	0.984	0.971	0.922	0.910	0.811	0.749	0.742	0.962	0.811	0.961
Perl	1.455	1.000	0.927	0.708	1.000	0.977	0.936	0.941	0.938	0.896	0.893	0.991	0.897	0.983
TFFT	2.240	1.000	0.901	0.652	1.000	0.998	0.977	0.996	0.773	0.754	0.759	1.000	0.874	0.958
Tomcatv	5.445	1.000	0.940	0.631	1.000	0.998	0.984	0.677	0.723	0.710	0.713	0.998	0.879	0.974
Xlisp	2.590	1.000	0.869	0.543	1.000	0.999	0.964	0.890	0.842	0.815	0.822	0.981	0.862	0.974
RTW Avg	2.594	1.000	0.912	0.670	0.998	0.992	0.967	0.917	0.837	0.811	0.816	0.992	0.892	0.975

Table B.7: Relative Performance with Fewer Registers (8 int/8 fp).