HARDWARE TECHNIQUES TO IMPROVE THE PERFORMANCE OF THE PROCESSOR/MEMORY INTERFACE

by

DOUGLAS CHRISTOPHER BURGER

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

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN–MADISON
1998

Abstract

Technology trends are making communication, both on and off the microprocessor chip, more expensive relative to computation. In this dissertation, it is shown how a current-generation microprocessor spends over two-thirds of its time performing no useful work, stalled for memory. For the aggressive, modern processors that were measured, over half of the stalls due to memory result from insufficient memory bandwidth, as opposed to bank access or data transmission latency.

While bandwidth limitations can be obviated by paying a sufficiently high price, in this dissertation hardware techniques to mitigate bandwidth-related performance losses are explored. The efficiency of caches is measured, showing that the fraction of useful data in the cache over time is generally under 20%. A theoretical lower bound is placed on the amount of bus traffic that a cache may produce, and it is shown that current caches generally produce one to two orders of magnitude more traffic than is necessary.

A number of solutions are proposed for reducing traffic to improve performance. Two techniques are measured that dynamically adapt what is fetched upon a block miss, filtering unneeded data. The first policy is *dual-size fetching*, which alternates between fetching large and small blocks depending on how much spatial locality exists. The second is *subblock prefetching*, which fetches discontiguous sets of small blocks when stable usage patterns exist. A technique called *bus prioritization* schedules speculative fetches on the bus, to reduce queueing delays for data that are needed by the processor.

Cache and physical memory hybrids are explored, to better manage large on-processor memories. A memory hierarchy taxonomy is proposed, and a hybrid called the *Indirect Cache* (ICE)—which manages an on-chip cache much like a physical memory, with its own page table and translation buffer—is evaluated. It is shown that the performance of ICE is both superior to and more stable than conventional alternatives.

Finally, the distribution of processing power into physical memory, to reduce both memory latency and traffic, is explored. One such architecture is evaluated in detail (the DataScalar architecture), and it is shown that—for memory-limited applications—this scheme can offer significant speedups (9% to 100%).

Acknowledgments

This dissertation is the culmination of not only years of effort, but also of the training, teaching, and support of many people. I would first like to acknowledge gratefully the Wisconsin architecture faculty, who have educated me and trained me in innumerable aspects of a research and academic career: Jim Goodman, my advisor, David Wood, Mark Hill, Guri Sohi, and Jim Smith. I cannot thank them enough for their support, advice, training, squash, and friendship. I hope that my subsequent career makes them proud.

Equally important in my development (and enjoyment of the process) were the Wisconsin architecture students, who helped to create an environment and excitement that will be difficult to replicate ever again. Babak Falsafi, Scott Breach, Alain Kägi, and T.N. Vijaykumar were my closest friends through graduate school; wonderful people with whom I studied, worked, lived, and played. I also owe Stefanos Kaxiras, Andreas Moshovos, Subbarao Palacharla, Todd Austin, Steve Reinhardt, and Alvy Lebeck a debt of thanks, for both their intellectual support, collaborations, and friendship.

I would also like to acknowledge the institutions that provided our group with funds and equipment that supported my research: the Intel Research Council, for funds, workstations, and my fellowship; Sun Microsystems, for their workstation donations, and the National Science Foundation, whose grants funded the majority of my graduate career.

Last and most important, I thank my family: my parents, Ann and Bob Burger, and my brother Bob, who gave me the upbringing and education that allowed me to reach this point. I never could have done it without them.

Given the large number of people who have provided me with advice, support, and technical interaction, I have chosen to write the dissertation in the first person plural. By doing so, I am acknowledging the daily intangible contributions of many of my advisors and peers. I will mention explicitly significant and concrete contributions that others have made to this work, so that others' contributions are not hidden or buried in my choice to use "we" instead of "I".

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of a computer is to perform useful processing of information. In modern, general-purpose computers, this purpose is achieved with an electronic engine that performs arithmetic computations on data. These data must be stored in such a way that the arithmetic engine, or *processor*, can access them quickly and simply. Modern computer systems store data as bits of information in the *memory system*. We believe that there are two fundamental issues in computer system design. One is the orchestration of the communication between the arithmetic units and the stored data (the *processor/memory interface*). (The other is the method of expressing an algorithm to the computational hardware). Effective communication between the processor and memory is crucial in preventing overall computing performance.

The ideal processor/memory interface (to which we shall henceforth refer as the PMI, for brevity) would allow any computational unit to receive any needed operand instantaneously. An ideal memory system has three desirable properties: it is fast (the processor may access any operand quickly), it is large (the memory system holds all the operands that the processor needs), and it is cheap. Unfortunately, technology permits only two of these properties to be improved at the expense of the third [17]. It is therefore possible to build large, cheap memories that are slow (disks and tapes), or fast, cheap memories that are small (registers and level-one caches), and so on. Since the ideal memory system (and consequently the PMI) is not implementable, the PMI must be carefully designed so as not to be the bottleneck for good overall system performance.

The ubiquitous approach for building cost-effective, high performance interfaces between the processor and memory is the use of a memory *hierarchy*. In a memory hierarchy, a centralized processing core is connected to multiple memories, each of which is larger, slower, and

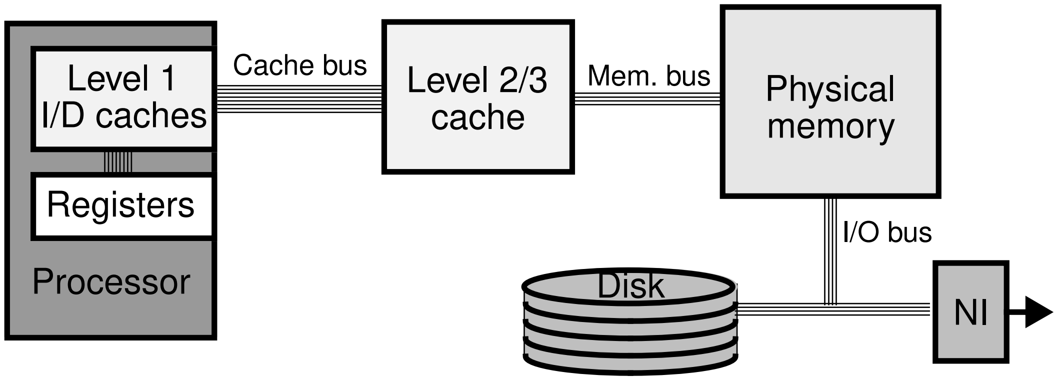


Figure 1-1: Typical modern memory hierarchy

cheaper (per bit) than the memories closer to the processing core. In Figure 1-1, we depict a memory hierarchy that is typical for 1998, in which a small, fast memory (the register file) contains the most important subset of data, a slightly larger, slower memory (the level-one cache) contains a larger subset of data, and so on. At the bottom of this particular hierarchy is the disk (or network), which is extremely slow but holds all of the operands. By varying size and speed, a memory hierarchy may provide the illusion of a single large, fast, cost-effective memory, which can match the rate at which the processor consumes instructions and data (the *processor bandwidth* [80]).

If microprocessor cores become sufficiently powerful, streaming data into a centralized core at a sufficiently high rate may not be possible to do cost-effectively. A potential solution is to distribute the PMI among multiple processing cores [49, 57, 76, 133], each with its own memory hierarchy. A distributed PMI is more difficult to program, and its relative effectiveness may be highly dependent on application behavior. The burden of distributing the communication between processing cores and memory must be placed on the programmer, the compiler, the run-time system software, the hardware, or a combination of the four. Most distributed PMI architectures also use memory hierarchies (SIMD and processor-in-memory approaches can be an exception [7, 49, 50, 57, 67, 75, 76, 130]), both above the level of distribution and below (for instance, SMPs have registers, L1, and L2 caches above the distribution point, physical memory and disk below). The level at which distribution occurs is now often chosen to widen the PMI cost-effectively (*e.g.* higher bandwidth out of the register banks in clustered architectures, such as in the Alpha 21264 [55] and proposed MultiCluster architecture [37], and higher instruction fetch bandwidth in Multiscalar processors [114]). Choosing other levels in the memory hierarchy at which to distribute the PMI can result in interesting architectures, as we shall see in Chapter 6.

1.1 Dissertation roadmap and contributions

In this dissertation, we demonstrate experimentally that careful consideration of the PMI is becoming increasingly important to system designers. Although much previous research has focused on average memory latency (or the *depth* of the PMI), we discuss in this introduction how it is memory *bandwidth* (the *width* of the PMI) that is coming to limit microprocessor performance. Consequently, the focus of the rest of the dissertation is on techniques to improve system performance by reducing cache and memory bus traffic, thus increasing the system's effective bandwidth. One of our previous papers [13] pointed out both that many of the traditional latency tolerance techniques have little effect on bandwidth-bound programs, and that programs are becoming more bandwidth bound. To our knowledge, it was the first to make this case comprehensively.

We show in Chapter 3 that traditional memory hierarchies (caches in particular) make rather poor use of both of their capacity and available memory bandwidth. We show that on average, caches generally use less than 20% of their capacity effectively. We also place and measure a formal upper bound on the effectiveness of caches at reducing communication, and show that the potential exists for up to two orders of magnitude in traffic reduction. This was the first formal bound on cache traffic that we have seen, and it has been extended recently by others [122]. We extend this bound analysis by dissecting the gap between optimal and actual traffic into a breakdown of cache mechanisms, which measures the usefulness of each cache mechanism at reducing memory traffic.

Using the results of the bounded traffic analysis, in Chapter 4 we propose a number of techniques to improve the bandwidth performance of traditional, cache-based memory hierarchies that assume a centralized PMI. The techniques we propose in this chapter are designed to make cache traffic more *efficient* (reducing unneeded communication) for caches of a fixed size. These traffic optimization techniques are: dual-size fetching, subblock prefetching, and bus prioritization. Taken together, they are an aggressive attempt to improve performance by reducing memory traffic, thus increasing effective bandwidth and mitigating bandwidth limitations.

In Chapter 5, we examine how the cache hierarchy may change with the emergence of large (multi-megabyte) on-chip memories. We describe a new memory hierarchy taxonomy, which compares cache mechanisms to those of physical memory, the goals being to rethink on-chip memory management mechanisms and to propose new, alternative cache organizations. We propose three classes of cache/memory hybrids: *logical*, *physical*, and *unified*. Using the taxonomy, we propose a logical hybrid for large caches called an Indirect Cache, which uses page-table-like structures to manage large on-chip level two caches efficiently. We show that the Indirect Cache works synergistically with the traffic optimization techniques described in Chapter 4 improving overall performance across a wide range of benchmarks. We present some brief functional results for a simple physical hybrid, showing that for extremely large on-chip memories, it is possible to map a fraction of physical memory on-chip and incur the same or fewer number of slow off chip accesses. Finally, we examine the effect that manufacturing technology may have on improving the PMI, by integrating more of the system (DRAM) onto the processor, which includes eventually combining all memory and logic onto a single substrate [13, 92, 100]. If the processes permit, merging the DRAM and logic on one die may allow the memory hierarchy to be "flatter," bringing it closer to the ideal and thus reducing the need for distributing it. We present some simulation results that indicate that, with current processors and workloads, full processor/memory integration is unlikely to provide the performance boosts necessary to make it cost-effective. This space has been well-traversed by the IRAM group [42, 78], and our results confirm theirs. We make no fundamentally new contributions in this section, but include it for completeness.

In case centralized PMIs prove unsuitable for high-performance processors in the future, we explore a class of distributed PMI architectures in Chapter 6 called *memory-centric architectures*, in which processors are distributed to portions of the physical memory. The architecture described in this chapter is the DataScalar architecture, which relies on the hardware to perform the distribution of work across the multiple PMIs. We show that the DataScalar architecture can reduce the global traffic significantly—thus improving performance—without placing any complexity burdens on the programmer or compiler. While the base execution model of DataScalar is not new (it was first proposed by the Massive Memory Machine work [45]), we recognized that this execution model could improve performance for modern, asynchronous processors. We also proposed new techniques that solved the problems associated with running this execution model on an implementation that actually improved performance (these problems included asynchronous communication, speculation, and caching). In our last chapter (Chapter 7), we summarize our results and draw conclusions about the long-term implications of this work.

Both technology trends (the oft-cited fact that processor clocks are outstripping DRAM access speeds) and our experimental results indicate that the processor/memory interface will play a more critical role in determining sustained system performance than it has in the past. A number of publications [71, 132] have referred to the unequal scaling of processor and memory performance as a “wall.” Implicit in that term (and explicit in some papers [132]) is the assumption that the memory system will act as an eventual hard limit on the growth of system performance. This belief is mistaken; system designers will redesign the PMI as needed to keep it balanced and cost-effective. The divergent trends may result in less conventional solutions to keeping the system in balance, ranging from more sophisticated and complex memory hierarchies to distributed processor/memory interfaces. In later chapters, this dissertation proposes and explores a number of such solutions. For the rest of this chapter, however, we explore the subtle relationship between memory latency and memory bandwidth, and make the case that memory bandwidth will be a significantly more important resource in driving future designs.

1.2 Increasing importance of memory bandwidth

The memory system must provide operands to the processor with both low latency and high bandwidth. If the memory system provides a high-bandwidth, high-latency path to the processor, data dependences on the critical path will limit the rate at which the processor may request data, resulting in a low effective use of the bandwidth. If the memory system provides a low-latency, low-bandwidth path to the processor, the saturated connection will cause contention delays on the critical path, effectively lengthening the critical path with non-critical work. It is therefore important that the memory system support both a sufficiently low average latency per request and a sufficiently high rate of request completions. While much work in the past has focused on reducing memory latency, the focus has not generally been on the additional latency incurred as a result of insufficient memory bandwidth. In the following subsection, we make the case that the latter will soon be a more important component of memory system performance than row access latency alone.

1.2.1 Increasing bandwidth needs

Memory bandwidth issues will come to dominate performance considerations in microprocessor-based systems for three reasons: (1) exponential performance growth, (2) unequal scaling of bandwidth costs for different components in the system, and (3) the nascent capability to place as many functional units on a die as needed to consume the available memory bandwidth.

As performance increases exponentially, the rate at which instructions and operands are consumed increases correspondingly. Furthermore, as data sets and binaries grow, the microprocessor must consume larger data sets in a shorter period of time. This requirement increases the rate at which large quantities of data must be moved from disk or main memory all the way up the memory hierarchy into the processor's registers.

We predict that the primary bandwidth bottleneck—for processors that are sensitive to packaging costs—will be at the processor pin interface, not the on-chip buses, system buses, or DRAM interfaces. The on-chip buses will not be a problem because in the foreseeable future, the primary problem with moving data from the pin interface to the registers will be latency, not bandwidth. Increased device counts will allow replication of key structures and wide paths on-chip, so bandwidth will be less of an issue than will the delays associated with long on-chip wires [86]. In terms of sustaining sufficient bandwidth, the pin interface will be a considerably more serious problem, simply because it cannot be widened nearly as much as the on-chip paths. Furthermore, the processor pins cannot be distributed, replicated, or interleaved cheaply as can other communication resources in the system (such as replication of buses, or interleaving of DRAM banks). While carefully designed transmission lines, such as the various Rambus interfaces [96] may bring data across the pins at a rate keeping pace with processor clock improvements, increased exploitation of both instruction-level parallelism (ILP) and speculation will continue to increase pin counts.

If pin counts could scale indefinitely with performance, processor bandwidth would not be an issue. However, we believe that packages are unlikely to scale cost-effectively (in the absence of bandwidth-specific solutions) with on-chip device counts. In Figure 1-2, we show the growth in microprocessor package pin counts over the past 20 years. We compiled this data by hand, from both the processors' original manuals and back issues of *Microprocessor Report*. The y-axis uses log scales, and the x-axis use a linear scale. Plotting a line with a least-mean-squares analysis, we find that, for the microprocessors surveyed, pin counts have been growing an average of 16% per year for this period. For the next decade, the 1997 SIA National Technology Roadmap [102] forecasts a lower (~11%/year) increase, predicting packages of 7300 pins for the high-performance microprocessor of 2012. Should these dramatically large packages prove too costly, other techniques must compensate by providing higher effective bandwidth across a narrower channel.

This disparity in pin counts versus performance is by no means limited to future projections. The rate of increase of processor pins has traditionally been much slower than that of transistor density *and* performance. In Figure 1-4, we plot (again on a semi-log scale) processor performance[1] per pin versus time over the past 20 years. The raw performance per pin is increasing exponentially, despite the increase rate in pin count shown in Figure 1-2. This graph, however, does not consider pin frequency (the rate at which signals are clocked across

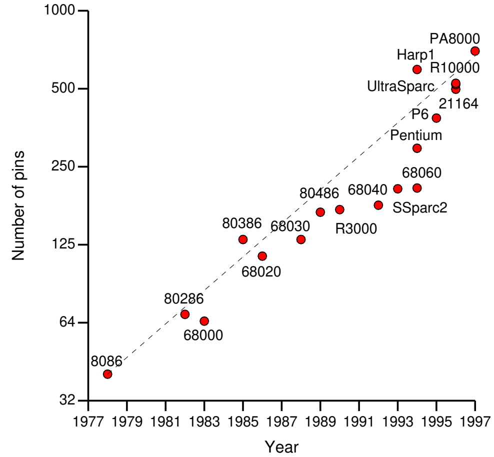


Figure 1-2: Processor pin counts

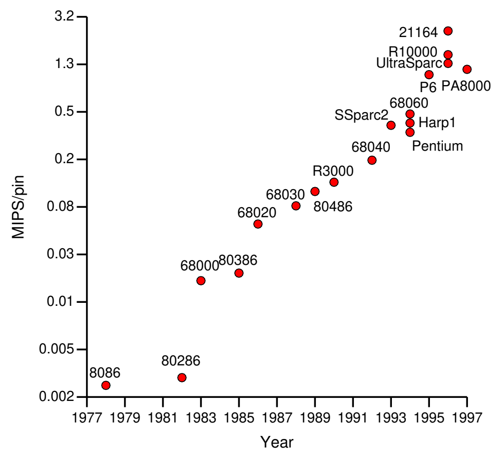


Figure 1-3: Raw performance per pin

the pins)—packages and buses are designed to provide sufficient off-chip bandwidth to each generation of processors. In Figure 1-4, we therefore incorporate increased pin signalling speeds, and plot the raw performance to total package bandwidth ratio versus time. The graph shows that performance increases are also exponentially outstripping the growth in raw peak package bandwidth.

In terms of future projections, the projected package pin count of 2012 is about a factor of ten greater than is typical today, but performance is projected to increase 700-fold. Since the processor bandwidth will likely increase by a proportional factor (or even more, if aggressive

1. Performance here is measured in VAX MIPS for the 680x0 and early 80x86 processors, and issue width times clock rate for the others. These two measures cannot be compared directly, but are sufficient to view 20-year trends.

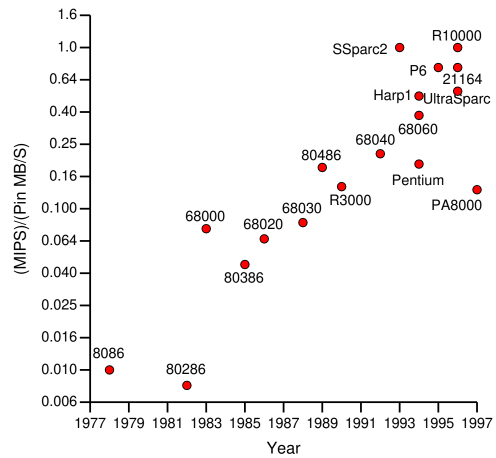


Figure 1-4: Performance per processor pin bandwidth

speculation increases processor bandwidth requirements more quickly than performance), the effective off-chip bandwidth will need to be increased by a factor of 70 without adding pins. Assuming a ten-fold increase in pin frequencies, the off-chip traffic must still be reduced by a factor of seven to balance the PMI. Techniques to reduce off-chip traffic could play an important role in rebalancing the system.

While these numbers are debatable—as applications and cache access patterns are likely to affect off-chip bandwidth requirements significantly more than raw processor performance—it is clear that reducing off-chip traffic would ease the difficulty of scaling the processor chip interface along with processor performance growth. Even if this scaling is technologically feasible, adding bandwidth adds cost. Reducing the need for extra bandwidth will make future systems cheaper while achieving the same level of performance (since, as the supercomputer domain has shown, more bandwidth is always available if the customer is willing to pay.) We discuss techniques and structures to reduce off-chip traffic in Chapter 4, Chapter 5, and Chapter 6.

1.2.2 The interactions of latency and bandwidth

The relationship between latency and bandwidth in the memory system is intricate and subtle. Some techniques (such as increasing the bus clock) that reduce the latency of a single request will improve memory bandwidth, while others (such as hardware prefetching) actually reduce effective memory bandwidth. When the available memory bandwidth is insufficient, other requests may be stalled or queued in the memory system due to contention for shared resources (such as buses, cache ports, or memory ports). This queueing is manifested as additional latency, which may reduce processor performance. In the end, limited bandwidth is measured as additional latency for memory requests. When we describe "trading latency off for bandwidth," we mean that some latencies are reduced while other latencies are increased as a result of more memory traffic.

There has been a historical focus on memory latency because it has been growing relative to processor cycles. The number of cycles required to service a main memory access has steadily increased over the past 20 years. This trend is the result of two factors. First, DRAM access times are being outstripped by processor clock speeds, since DRAM chips are generally optimized for capacity (through high density), while microprocessors are optimized for speed. (This is true even though DRAM access times have dropped considerably over the years, at the rate of approximately 7% per year [97].) Second, the path lengths to main memory have increased, as both the depth and complexity of the memory hierarchy (non-blocking caches [79], multiple levels or cache, and sophisticated memory scheduling and data transmission [30]) have increased.

Consequently, researchers have proposed numerous techniques to reduce (and/or tolerate) the average effective memory access latency. Some of these techniques include multithreading, dynamic scheduling, decoupling, hardware prefetching, software prefetching, and more aggressive hardware in the memory system. In modern processors, however, some of these optimizations that were intended to reduce average memory latency actually worsen it. While they may improve the latency of a single operation, they may also slow down other operations by generating extra traffic and thus causing contention that results in a higher average memory latency, and worse overall performance. As processors exploit more instruction-level parallelism, and memory systems come to resemble queueing systems more than single-transaction systems, sustained memory bandwidth will become a more important quantity than the mean latency of individual requests.

Traditional metrics—such as cache miss ratio or average memory access time—may provide a first-order approximation to memory system performance, but they neither translate directly into system performance, nor do they provide insight as to the sources of performance loss in the memory system. For example, four simultaneous cache misses in a lockup-free cache will appear as one cache miss latency to the processor, but would count as four distinct misses when calculating average memory access time.

In this subsection, we address this problem by dissecting execution time into three discrete components: *processor time*, *latency time*, and *bandwidth time.* These categories are not discrete time periods of execution. They are more similar to "assignments of responsibility" for underutilized resources. Thus, at any given cycle in a program's execution, various underutilized resources in the microprocessor may be contributing to all three categories simultaneously.

Processor time is the time in which the processor is either fully utilized, or is underutilized due to insufficient fine-grained parallelism (as opposed to the memory system). In an ideal system with a perfectly balanced PMI, processor time would equal the program execution time (*i.e.*, the processor would never suffer lower utilization due to the memory system). Such a situation does not represent an upper bound on processor performance; execution time could still be decreased by improving the processor core (better branch prediction, wider issue, etc.)

Latency time is the increase in execution time caused by untolerated, *contentionless* memory latencies. These latencies include the time required to resolve cache misses, access cache or memory banks, and the minimum time required to transmit the data back to the processor. By contentionless we mean that the latency measured is never increased by interference of multiple requests. Thus, adding more bandwidth anywhere in the memory system should never reduce latency time.

Bandwidth time is the increase in execution time caused by contention in the memory system, resultant from insufficient bandwidth between levels of the memory hierarchy. Queueing delays can occur at either the memory banks or at the buses. When memory requests experience queueing delays in the memory system, their latencies to completion are increased. That increase may inflate total program execution time. Bandwidth time measures the inflation caused by memory queueing delays.

We now define this execution time dissection formally. Let T be a program's execution time. T_P, T_L, and T_B are a partitioning of T, the time spent in each of these three categories (processing, latency, and bandwidth, respectively). Let f_P, f_L, and f_B be these times normalized to T (thus representing the fractions of time spent in processor, latency, and bandwidth time). We define T_P as the execution time of the program assuming a perfect memory hierarchy (i.e., every memory access completes in one cycle). T_I is measured as the execution time of the program assuming an infinitely wide path (*i.e.*, infinite bandwidth) in between adjacent levels of the memory hierarchy. f_P, f_L, and f_B are computed as follows:

$$f_P = T_P / T \tag{1-1}$$

$$f_L = T_L / T = (T_I - T_P) / T \tag{1-2}$$

$$f_B = T_B / T = (T - T_I) / T \tag{1-3}$$

These metrics enable us to estimate more accurately the performance impact of an imperfect PMI in complex modern processors, which cannot be calculated directly from average memory latency or miss ratio. They also enable us to view the performance impact of latency tolerance and reduction techniques directly, which we discuss in the next subsection. We note that a similar dissection was independently proposed by Kontothanassis *et al.* [77].

There are two major classes of techniques for reducing the impact of long memory latencies: *latency reduction* and *latency tolerance*. Latency reduction decreases the time between the issue of a memory request and the return of the needed operand. Some latency reduction techniques include hardware prefetching [21, 43, 47] (which speculatively bring in data before they are requested), increased cache block size, larger caches (improved hit ratio), and more aggressive memory hierarchies (*e.g.*, faster buses, sub-banked caches, and lower-latency DRAM cores). Latency tolerance involves performing other computation while a memory request is being serviced, so that the memory latency for that request is partially or completely hidden. Some common latency tolerance techniques include software prefetching [18, 22, 31, 54, 124], dynamic scheduling [123] (allowing instructions ahead of a load in the dynamic instruction stream to execute), decoupling [108, 109] (allowing the memory unit to run (or *slip*) ahead of the execute unit), and multithreading [1, 107, 125] (switching to other threads during long-latency operations).

In Table 1-1, we list the effects that various latency reduction techniques (Table 1-1A), latency tolerance techniques (Table 1-1B), and processor enhancements (Table 1-1C), may have upon overall system performance. The arrows in the table represent the relative change to each fractional component of execution time when the optimization in question is applied. For example, an up arrow indicates that an optimization will cause that fraction of execution time to increase. A question mark indicates that a given fraction is not directly affected by the optimization, but may either increase or decrease depending on the relative contributions of the other two fractions.

The latency reduction techniques listed in Table 1-1A increase f_B in two ways: (1) by increasing the amount of traffic that must be moved across the PMI, and (2) by successfully reducing f_L, which reduces the execution time, and therefore increases the rate at which the same amount of data must be moved across the pins. Hardware prefetching will increase bandwidth stalls (f_B) by fetching unnecessary data (when the prefetch is unneeded), but it will generally reduce latency stalls (f_L) when it does successfully issue a needed request in advance. If the latency stall reduction outweighs the bandwidth stall increase, the fraction of time spent doing useful computation (f_P) will increase, if f_B outweighs f_L, then f_P will decrease. Larger cache blocks have an effect similar to hardware prefetching, reducing f_L and increasing f_B. Both techniques, however, could increase f_L if pushed too far, due to interference/cache pollution effects.

As do the latency reduction techniques, *all* of the latency tolerance techniques that we list reduce memory latency stalls at the expense of increasing bandwidth stalls. The first four latency tolerance techniques listed in Table 1-1B increase bandwidth stalls only by reducing execution time, thus increasing the rate at which the same quantity of data must be moved across the PMI. Lockup-free caches allow the processor to overlap memory requests, thus

A. Latency reduction	f_P	f_L	f_B
Hardware prefetching	?	↓	↑
Larger cache blocks	?	↓	↑
B. Latency tolerance	f_P	f_L	f_B
Lockup-free caches	↑	↓	↑
Software prefetching	↑	↓	↑
Intelligent load scheduling	↑	↓	↑
Data value speculation	↑	↓	↑
Speculative loads	?	↓	↑
Multithreading	?	↓	↑
C. Processor enhancements	f_P	f_L	f_B
Faster clock	↓	?	↑
Wider issue	↓	↑	↑
Dynamic scheduling	?	↓	↑
CMPs	↓	?	↑
Speculative threads	?	?	↑

Table 1-1: Effect of memory latency optimizations on execution time breakdown

reducing execution time but increasing the rate at which data must be brought in (and therefore increasing contention). Software prefetching, aggressive load scheduling, and data value speculation all reduce latency stalls by early acquisition (or speculation) of the result of loads. Such techniques do not reduce bandwidth stalls, since memory traffic is not reduced (with data value speculation, the operands must still be fetched from memory to validate the speculation). However, since these four techniques increase f_P as well as f_B, a larger relative fraction of execution time is spent both doing useful work and stalling for contention.

The fifth and sixth optimizations listed in Table 1-1B increase memory traffic, unlike the first four listed in section B of the table. Speculative loads increase total memory traffic with each misspeculation. Multithreading increases total memory traffic when threads interfere in the cache, causing more cache misses and thus more memory traffic. This additional memory traffic will increase f_B in addition to the increases caused by execution time reduction. If the increases in f_B outweigh the reduction in f_L, the result will be a lower processor utilization (a decrease in f_P). Conceptually, a technique such as multithreading can be effective for a latency-bound program, but multithreading will become less effective as a program becomes more bandwidth-bound (*i.e.*, f_B increases), and may even be detrimental. Finally, if the cache interference caused by multithreading grows sufficiently high, f_L will also increase. (This effect corresponds to the pollution effect previously discussed for large cache blocks).

In Table 1-1C we list the effects that some common microprocessor ILP-style enhancements have on our execution time breakdown. All of the enhancements listed in this part of the table reduce the time it takes to perform the computations, whether by executing the computations faster (increased clock speed), and/or by executing more operations in parallel (increased issue width, speculative threads, or chip multiprocessors). None of these techniques reduce memory traffic, and some may actually increase it (speculative threads may generate extra memory traffic due to both cache interference and coarse-grain misspeculations). As a result, these techniques increase f_B uniformly.

The techniques discussed in this subsection focus on reducing execution time by reducing either latency stalls (f_L) or processing time (f_P). As programs become more bandwidth-bound (f_B grows larger), for the reasons discussed previously in this chapter, these techniques will all become less effective. In previous studies [13, 14], we measured the execution time dissection experimentally for current-generation memory systems, and found that f_B is in fact growing substantially as processors become more aggressive. For simple processors, f_B was 14% of execution time. For fast, aggressive out-of-order processors that incorporated prefetching and speculative execution, the time spent stalling for memory was over 50% of execution time, and over a third of execution time was consumed by bandwidth time. We present the experimental results from the previous papers with an expanded analysis in Appendix A.

The remainder of this introduction is dedicated to a survey and classification of the bandwidth-specific techniques that we propose in this dissertation.

1.3 Bandwidth-specific solutions

There are a variety of ways to improve the effective width of the PMI (*i.e.* the effective bandwidth). In this section, we survey four such categories. The first is the improvement of cache memories in a traditional memory hierarchy with optimizations that reduce traffic, but do not incur correspondingly large penalties in latency. The second category is distribution of processors into the memory, splitting the processor/memory interface into multiple points (ideally making a wide PMI more cost-effective). The third category is flattening the memory hierarchy with tighter integration (specifically, placing the processor and physical memory together on one or more chips), using new manufacturing processes. The fourth and final category we describe to improve cost-effective bandwidth is the only method that we do not address in this dissertation, outside of this chapter. This category consists of techniques to reduce the fundamental, intrinsic amount of PMI communication required to solve a particular problem.

1.3.1 Tuning the PMI (reducing memory hierarchy traffic)

Most of the cache research of the past two decades has focused on two issues: reducing miss rates and improving cache access time (throughput), without necessarily considering memory traffic. Since reducing miss rates may also reduce memory traffic [51], the two goals are closely related. However, minimizing the miss ratio at the expense of increased memory traffic can degrade performance, as we shall see in Chapter 4. For our cache studies, we focus on two related goals: (1) how to reduce memory traffic with only minor increases in the miss ratio, and (2) how to reduce the number of misses without paying the price of significantly increased traffic.

1.3.1.1 Traffic-efficient caches

We will show in Chapter 3 that caches have a low *efficiency*; most of the space of a typical cache holds useless bits at any given time. This result led us to hypothesize that improved mappings could reduce hit rates by holding more useful data on-chip. We also hypothesized that much of the wasted space resulted from unnecessary bytes being loaded from memory, thereby also wasting bandwidth. We validated this hypothesis by performing experiments (presented in Chapter 3) that measured a lower bound on the amount of memory traffic that a cache could produce. We found that caches produce significantly more memory traffic (factors of 2 to 100) than is theoretically necessary. We dissected this gap into the factors by which the lower bound differs from a traditional cache (block size, write policy, associativity, and replacement policy), measuring the relative combinations of each. Our results showed that block size is, unsurprisingly, the largest contributor, but that the other three factors can each be equally or more important, depending on the application.

Since all four of the cache factors we measured have the potential to help reduce memory traffic, we propose distinct solutions for each factor, aimed at reducing traffic without incurring penalties that offset the gains from traffic reduction:

- **Block size/read traffic**: we propose three techniques to reduce unnecessary read traffic. The first is *dual-size fetching*, in which cache misses may either bring in an entire block[1] or simply a subblock into a subblocked cache, based on the expected spatial locality in the block. The second technique is *subblock prefetching*, which loads a subset of subblocks within an address upon a miss to that block. Ideally, the hardware will load only the subblocks that will be needed, preventing the useless (non-loaded) subblocks from consuming bus bandwidth. The third technique is *bus prioritization*, in which non-critical subblocks, specified by the former two policies, are speculatively loaded across the bus so long as there are no other requests pending. Upon arrival of a higher-priority request, the hardware finishes loading the current subblock and then allows the higher-priority request to proceed.
- **Write traffic**: we propose one techniques to eliminate write traffic. By using redundant computation at multiple processors, we can completely eliminate write traffic from the inter-processor bus, at the cost of some extra read traffic. We will describe this scheme in more detail in Section 1.3.2.
- **Associativity**: cache conflicts can generally be reduced by increasing set associativity (barring pathological interaction of the application and replacement policy). In this following subsection, we will discuss a cache organization that borrows from virtual memory designs to allow full associativity with less impact upon hit time than conventional con-

1. Multiple terms exist to describe sector caches [84], in which a large *sector* is broken up into multiple *blocks*. Sector caches are sometimes called subblocked caches, and the sectors are referred to as *address blocks*. The blocks are sometimes also called *transfer blocks* or *subblocks*. For consistency, throughout the dissertation we will refer to address blocks (sectors) simply as **blocks**, and transfer blocks as **subblocks**.

tent-addressable memories.

- **Replacement policy**: the ideal replacement policy would use prescience to predict the best victim in a set. Many caches today use either a least-recently-used (LRU) policy, or an LRU approximation. Our study of optimal caches shows that while this policy is generally effective, there are cases where further improvements are possible. We propose *correlated replacement*, in which the address of a block influences the choice of its replacement, as a technique for improving cache efficiency by better identifying dead blocks in the cache.

The goal of these techniques is to make both the use of the cache capacity and the transmission interconnect more efficient, by loading and storing less useless data. One might argue that all this additional complexity is not worth the trouble, as cache sizes are growing relentlessly with each new generation of chips. We believe that cross-chip wiring delays will force chips to be heavily partitioned, and these partitions will have a finite capacity, and will thus benefit from being more efficient since their size may be restricted. We discuss this issue further in Chapter 6. For now, we turn to a discussion of design strategies for large on-chip caches of the near future.

1.3.1.2 Large on-chip caches

Given the performance increases of microprocessors and the growing difficulty of balancing the PMI, processor designers have been building progressively larger caches with each improved process generation. For example, the Hewlett-Packard has announced that their PA-8500 processor will have 1.5 MB of on-chip cache, in a radical departure from their previous design strategy (such as the PA-8000 and the PA-8200, which had no on-chip caches, but high-performance connections to large off-chip caches.) The Compaq Alpha 21364 will also have 1.5MB of on-chip cache.

This trend of increasing cache sizes shows no sign of abatement in the near future. Even if caches consume the same proportion of the processor die that they do today, the exponential growth in device counts presages giant on-chip memories. In Figure 1-5 we show that the proportion of processor chip transistors consumed by caches is growing, now accounting for

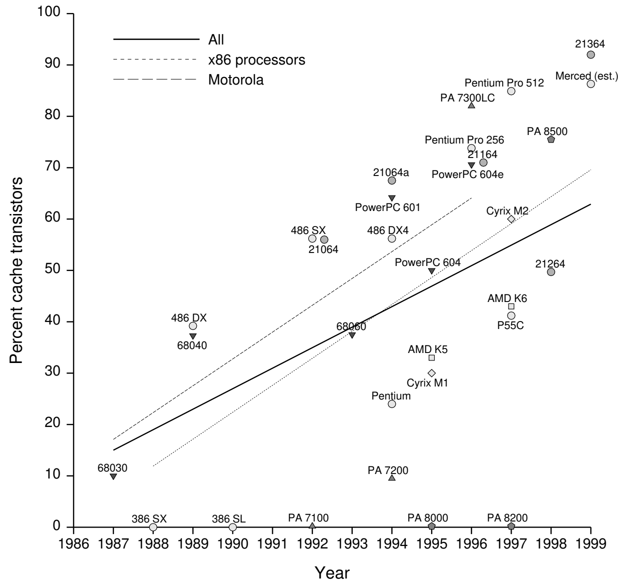


Figure 1-5: Fraction of processor transistors devoted to cache

between 50% and 92% of the on-chip transistor budget. If these trends continue, future processors will be mostly memory.

These large caches will shield the lower levels of the memory system from much of the increased processor bandwidth requirements. Current cache designs, while they will doubtless work well for these large caches, may not be the best operating point for such huge caches. In Chapter 4, we revisit cache design, and propose a taxonomy of mechanisms for individual levels in a memory hierarchy. Using this taxonomy, we propose designs for these huge on-chip caches that may be better suited to traffic-sensitive systems than simply increasing the size of current designs. Specifically, we propose and evaluate an alternative that we call an Indirect Cache (ICE). The ICE is a cache that is managed like a page table, with indirect indexing and a translation cache.

The goal of the optimizations described in this subsection is to improve the performance of conventional systems by improving cache performance. In the next subsection, we describe a more radical approach to improving scalability of the processor-memory interface.

1.3.2 Distributing the PMI (memory-centric architectures)

Traditional uniprocessors have been designed assuming a centralized processing core that is connected to the memory hierarchy using a single logical interface. As the processor grows more powerful and the required physical space for both the processor and the supporting memory system increases, supporting communication through a single logical pipe becomes harder to do. Distributing the PMI among multiple computational units and memories is a more scalable approach, but can introduce significant difficulties in mapping the computation onto the distributed substrate.

Many examples of distributed PMIs have already appeared in both the literature and in practice. Traditional parallel processors are distributed processor/memory interface machines. Symmetric multiprocessors (SMPs), for example, are distributed at the L2 cache level, and distributed shared memory machines (DSMs) are distributed at the physical memory level. To date, however, most of these machines were constructed not because of an unscalable PMI, but because volumes of commodity components (either CPU chips in SMPs or workstation boxes in DSMs) offered significant cost savings over comparably powered alternatives. The proposed RAW architecture [128] is an exception, distributing multiple processors on-chip, each with its own data cache and instruction store, to increase total effective bandwidth out of the PMI. However, the RAW architecture assumes that nearly all distribution of data and assignments of computation to processing nodes is done by the compiler. Henceforth, we will restrict our discussions of distributed PMI organizations to those that assume a logically sequential instruction stream (*i.e.* a uniprocessor programming interface).

Examples of distributed PMIs with sequential programming interfaces exist, with the distribution occurring at different levels in the memory hierarchy. At the register level, clustered architectures, such as the Alpha 21264 (or proposed MultiCluster architecture [37]) distribute the register interface to multiple banks of functional units, thus achieving high, yet cost-effective, bandwidth out of the global register files. Multiscalar processors [41, 114] increase instruction fetch bandwidth by distributing the instruction fetch (at the L1 I-cache interface) as well as the register banks. The Multiscalar work assumed centralized L1 data caches, although more recent proposals distribute the L1 data caches as well [53].

To our knowledge, the only proposal (besides our own) of an architecture with a serial programming interface that distributes the PMI at any level of the memory hierarchy lower than the level-one caches is the Massive Memory Machine [45], from which our work is derived. In Chapter 6, we propose a related class of architectures called *memory-centric architectures*, which distribute the PMI to the physical memory. These architectures execute unmodified serial binaries, and they reduce inter-processor traffic significantly. We propose two such architectures: DataScalar and Dynamic Data Threaded (DDT). The DataScalar architecture [15] uses redundant computation to reduce memory latencies and traffic. DataScalar architectures completely eliminate all request and writeback traffic, at the expense of some extra read traffic. DDT architectures perform a partial dynamic parallelization (in hardware) of the serial program, thus eliminating some of the read traffic, as well as the write and request traffic that DataScalar architectures eliminate.

1.3.3 Flattening the PMI (integrating the processor and physical memory)

In the previous two subsections, we discussed ways of improving systems' effective bandwidth by reducing traffic in conventional hierarchies and by distributing processing power (moving the PMI) into the physical memory. A third alternative is to reduce the number of levels in the memory hierarchy by bringing the large physical memory closer to the processor.

Physical memories have already begun to become more tightly coupled with the processor. The Rambus interface [30] provides close electrical coupling between some processors and physical memory. However, a tight physical coupling is also possible, if the entire system memory and the processing logic were integrated on a single substrate. Such integration is possible only if two factors hold: (1) there is a market for systems with only as much memory capacity (at least for the base models) as can be held on one processor, (2) merged memory and logic processes can be developed that support both fast gates and dense memory cells. Otherwise, the chip will have either insufficient performance or insufficient capacity to be a viable product in the market. Whether the processor support is developed depends on the performance advantages of putting all of the physical memory on-chip. In Chapter 5, we perform a trend and performance analysis, and find that complete integration improves the performance of current systems surprisingly little.

1.3.4 Shrinking the PMI (reducing processor/memory communication)

The previous three subsections dealt with organizing the distribution of processors and various memories to improve the cost-effectiveness of communication between processing logic and storage. An alternative solution to optimizing communication is to actually reduce the need for communication across that interface. We do not evaluate such solutions in this dissertation, but survey three of them here.

- **Algorithmic**: If the PMI becomes a major system bottleneck, different algorithms may be selected that use less cross-PMI communication. In addition to choosing different algorithms, code tuning that improves memory system behavior, such as cache blocking [19], may reduce PMI limitations. Finally, other optimizations exist that reduce PMI communication for a given algorithm (such as common subexpression elimination, which reduces register accesses, or memoization, which reduces both memory and register accesses).
- **Compression**: If the bandwidth of a certain level of the memory hierarchy is difficult to scale, and/or becomes a bottleneck, compression of the transmissions to increase the effective bandwidth may be a viable solution. Particularly as computation becomes less expensive relative to communication, compression is a feasible way of reducing the expense of communication. Researchers have examined numerous techniques for compressing various information being transmitted over the memory bus, such as data [24, 94], addresses [38], and instructions [28].
- **Instruction reuse**: Another way to reduce PMI communication is to avoid doing redundant operations, e.g. avoiding a load if the result of the load is already available in the processor code. *Instruction reuse* does exactly that; an on-chip buffer keeps track of operation results based on their input values, and when an operation is fetched whose inputs match those in the buffer, the result is returned from the buffer rather than computed or brought from memory [112]. In this manner, both register accesses and memory operations may be

reduced. In some sense, instruction reuse is a hardware version of memoization, albeit at a finer grain.

While these techniques may reduce the volume of communication across the PMI, the notion of operating on data is fundamental to computing. These techniques may help to alleviate bottlenecks and balance the processor and memory system; however, the very nature of computing makes it impossible to push these methods sufficiently far to be a comprehensive solution. We also note that these algorithmic techniques for increasing effective memory bandwidth are orthogonal and complimentary to those evaluated in this dissertation.

1.4 A word about cost

One of the things that makes quantitative computer architecture hard is sensitivity of the "best" solutions to cost. Throughout this introduction, we have talked about cost-effectiveness and cost/performance, but have no cost models to back up these assertions. It is possible to produce reasonably accurate cost models for current-generation systems. For example, Microprocessor Report has a complex cost model that estimates manufacturing costs for current-generation microprocessors. Also, Wood and Hill have proposed a cost model for current-generation multiprocessors [131], and showed that costup was a better metric than speedup for scaling parallel simulation systems, and that the dominant costs of these systems was memory. While similar models for future systems would be useful in evaluating tradeoffs among the systems, they are nigh impossible to construct with any confidence in their accuracy.

We attempted to model cost using several different metrics, such as bits of storage, package pins, dollars, and silicon area. Unfortunately, there are too many parameters affecting costs, the constants are frequently closely guarded secrets, and how they scale into the future is determined by market forces that are wholly unpredictable. We will therefore not address the issue of cost quantitatively in this dissertation, but will address performance tradeoffs quantitatively and cost qualitatively, leaving it to the interested industrial reader to determine if the performance gains are worth the price.

Chapter 2

Experimental Methodology

We performed all of the experiments in this dissertation using *software simulation*, in which a microprocessor (called the *target*) is modeled in software at various levels of detail by a software simulator, which executes on the *host*. The simulation environment we used was the SimpleScalar tool suite [9, 8], originally written by Todd Austin and extended for this dissertation research.

In this chapter, we first describe the limitations associated with our experimental methodology. We then describe both our simulation environment and our simulated target in detail. We conclude this chapter with an characterization of our benchmark suite, including validation of our sampling methodology.

2.1 Software simulation

There are risks involved with using software simulation as the sole methodology. Most significant, our tools have never been validated against an actual hardware implementation. Black and Shen [6] showed that microprocessor timing simulators can contain numerous bugs that can affect results significantly (errors on the order of 3% to 5%); specifically, they showed that small bugs can cause significant instability in the reported execution time of a simulated microprocessor, and that the correction of one bug can cause the error in simulated execution time to increase or even change signs.

An advantage to using the SimpleScalar tool suite, however, is that it is now being used extensively throughout the architecture research community. Several bugs have been reported by other people using the tools (and subsequently fixed, of course). In addition, we have made our memory hierarchy extensions (described later in this chapter) public, and they are now being used by several research groups. While the extensive distribution of the tools does not guarantee their accuracy or correctness, our confidence in their accuracy is substantially higher with the extensive external sanity checking.

Another serious concern with software simulation is the size of programs and data sets that can be simulated. Our simulation environment supports two levels of simulation accuracy; cycle-by-cycle microarchitectural simulation, in which the simulated execution time is the output (*timing simulation*) and fast simulation, in which the execution trace for the simulated program is generated, but the only statistics that are maintained are a few counters (*functional simulation*). The former (timing simulation) models the microarchitectural state, but incurs a four order-of-magnitude slowdown over running the target benchmark on real hardware. The functional simulation incurs only a two order-of-magnitude slowdown, but gives little useful data other than number of instructions traced and a few other statistics.

We have characterized the attempt to evaluate future microprocessors with software simulation as “simulating the processors of tomorrow on the machines of today with the benchmarks of yesterday” [15]. Even using yesterday’s benchmarks (such as SPEC95) with small data sets, a four order-of-magnitude slowdown is prohibitively large. For example, simulating the longest-running SPEC95 benchmark with our timing simulator would require approximately 100 days. There are a number of possibilities for reducing the simulation time sufficiently to perform tractable timing simulation of these benchmarks. We list them below in order of least to greatest complexity:

- **Small inputs**: by simulating the benchmarks with small inputs, the number of instructions that the target benchmark takes to execute may be reduced. However, small inputs have two disadvantages: they may demonstrate different memory system behavior (requiring a less aggressive memory system for a balanced PMI), and they may spend a disproportionate amount of time in specialized routines (such as initialization) for which the execution characteristics are atypical of the program when executed with large inputs. When the effects of these two conditions are acceptably small, running benchmarks with data sets that reduce execution time is an acceptable solution.

- **Simulate an initial fraction of the instructions**: it is possible to simulate a benchmark with its full data set, terminating the simulation before the benchmark completes, thus simulating some initial fraction of the benchmark execution. The main drawback with this strategy is that, as with small inputs, the initial fraction may capture an atypical period of execution (the initialization phase is a particular problem with this strategy). The initialization issue may be countered by starting up the timing simulation after some fraction of the program has already been simulated by a faster simulator (*e.g.* performing functional simulation to get through initialization, and then timing a fraction of the execution). This solution eliminates the most visible problem (initialization behavior), but the fraction of the program measured with timing simulation may still be atypical of the execution as a whole.
- **Sampling**: an improvement on the latter scheme is to simulate small fractions of the execution with a detailed (timing) simulator, racing from one *sample* to the next with a faster simulator (such as a functional simulator). The statistics taken from each sample are aggregated upon completion of the simulation, and should ideally approximate the simulated behavior of the entire application. Sampling has two drawbacks: the time required to move from sample to sample (which can be significant, even with a fast functional simulator), and *cold start* effects at the commencement of each sample (the simulator state is stale at the beginning of each sample, thus affecting the sample results until it is brought up to date, or *warm*). The overhead of moving from sample to sample may be eliminated by saving the architectural and I/O state at intervals (saving a *state checkpoint*), and jumping directly to the next checkpoint when a sample period completes, to begin the next sample (the drawback to this strategy is that each checkpoint requires disk space). Cold start effects may be mitigated by either ensuring that the samples are each sufficiently long, or by explicitly warming up the simulator (*i.e.*, branch predictor and cache) state before beginning the measurement of each sample.
- **Parallel simulation**: if completion time of a particular simulation is critical, the period between each state checkpoint may be simulated in full on different machines, in parallel,

effectively reducing the simulation time by a factor of as many machines are available (provided enough disk space is available to hold the checkpoints). The drawbacks to this approach are that maintaining numerous runs (and aggregating the statistics) becomes more complicated, plus this approach does not improve throughput, only latency. If time to completion of a particular run is not critical, each machine could be dedicated to running its own independent simulation, reducing the complexity of statistics aggregation. If throughput is critical, the techniques listed above (or a combination thereof) should be used instead.

The approach we take in this dissertation is twofold. For some benchmarks, which have data sets that lend themselves to reduction, we alter the inputs to reduce the number of instructions that must be simulated to run the benchmark to completion, but maintain the behavior typical of the full reference data sets. For some others, we use sampling. For still others, we use a combination of the two techniques.

2.2 The SimpleScalar tools

The SimpleScalar tools were originally developed by Todd Austin for his thesis work, while working for Guri Sohi in the MultiScalar project. Alain Kägi wrote the first instance of the detailed memory hierarchy simulator, extending the cache module to support callbacks (in which the requests and responses to the memory system are decoupled), non-blocking caches, and MSHRs. We extended his efforts by adding a virtual memory system, with address translation (physical or virtual caches), a generalized cache and bus network, back pressure throughout the memory system, and subblocked caches.

Software simulators may be either *trace-driven* or *execution-driven*. Trace-driven simulators accept a stream of execution events (from the benchmark) and calculate results based on that stream, whereas execution-driven simulators perform the execution of the benchmark as part of the actual simulation engine (*i.e.*, results from the simulation engine can affect the execution itself). We depict the organization of the SimpleScalar simulation environment in Figure 2-1. The tools generate an execution trace in a functional simulator, which is fed on-the-fly to a simulation engine (the timing simulator). The timing simulator is thus a trace-

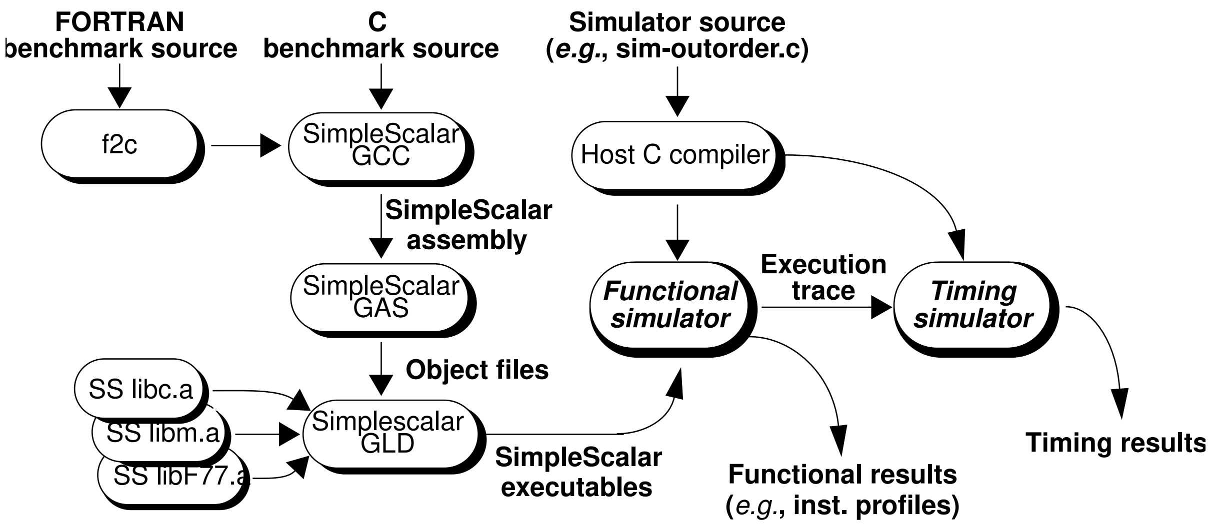


Figure 2-1: Overview of the SimpleScalar tools

driven simulator, albeit one tightly coupled with the execution engine. The separation of timing and execution is a source for some concern, as bugs in the timing simulator will not affect correctness of the benchmark execution (making them harder to detect). Ravi Rajwar merged our timing simulator with the functional core, resulting in a true execution-driven simulator, which was useful as a sanity check for the split simulation model. The simulators take binaries compiled to the SimpleScalar assembly format, decode the program text, and execute the instructions one by one. Correct execution of the benchmarks may be verified by comparing the outputs against outputs from binaries run on native machines.

2.2.1 Machine model

We assume a single machine model for the simulation results presented in this dissertation. In Figure 2-2, we list the SimpleScalar instruction set (ISA). The SimpleScalar ISA is similar to that of MIPS [95], except that there are no architected delay slots, and SimpleScalar supports both some additional instructions (square root) and some additional addressing modes (register+register addressing and auto increment/decrement). In Table 2-1, we show the archi-

Load/Store

lb - load byte
lbu - load byte unsigned
lh - load half (short)
lhu - load half (short) unsigned
lw - load word
dlw - load double word
l.s - load single-precision FP
l.d - load double-precision FP
sb - store byte
sbu - store byte unsigned
sh - store half (short)
shu - store half (short) unsigned
sw - store word
dsw - store double word
s.s - store single-precision FP
s.d - store double-precision FP

Miscellaneous

nop - no operation
syscall - system call
break - declare program error

Addressing modes:

(C)
(reg+C) (with pre/post inc/dec)
(reg+reg) (with pre/post inc/dec)

Integer Arithmetic

add - integer add
addu - int. add unsigned
sub - integer subtract
subu - int.sub.unsigned
mult - integer multiply
multu - int. mult. unsigned
div - integer divide
divu - int. div. unsigned
and - logical AND
or - logical OR
xor - logical XOR
nor - logical NOR
sll - shift left logical
srl - shift right logical
sra - shift right arithmetic
slt - set less than
sltu - set less than unsigned

Floating Point Arithmetic

add.s - single-precision (SP) add
add.d - double-precision (DP) add
sub.s - SP subtract
sub.d - DP subtract
mult.s - SP multiply
mult.d - DP multiply
div.s - SP divide
div.d - DP divide
abs.s - SP absolute value
abs.d - DP absolute value
neg.s - SP negation
neg.d - DP negation
sqrt.s - SP square root
sqrt.d - DP square root
cvt - int., single, double conversion
c.s - SP compare
c.d - DP compare

Control

j - jump
jal - jump and link
jr - jump register
jalr - jump and link register
beq - branch == 0
bne - branch != 0
blez - branch <= 0
bgtz - branch > 0
bltz - branch < 0
bgez - branch >= 0
bct - branch FCC TRUE
bcf - branch FCC FALSE

Figure 2-2: Summary of SimpleScalar instructions

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 2-1: SimpleScalar architecture register definitions

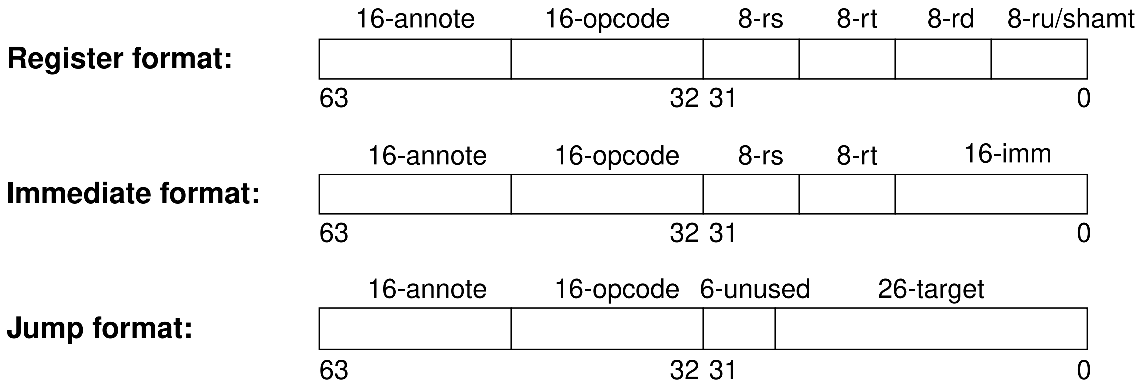


Figure 2-3: SimpleScalar architecture instruction formats

tectural registers supported in our machine model (32 integer registers, distributed as described in the figure, 32 floating point registers, and three special-purpose registers for holding results and condition codes). The SS ISA supports three formats of instructions—register, immediate, and jump—depicted in Figure 2-3. The instructions are 64 bits long, and each include a 16-bit annote field, which can be used for passing extra information to the hardware. Although the instructions are 64 bits long, our simulators have the capability to simulate instruction fetch as if they were 32 bits, since we are simulating a 32-bit machine.

In Figure 2-4, we depict the virtual memory organization that we assume in our system. At some point in the target's memory hierarchy, address translation is needed to provide a physical address, whether to access physical memory or a physically tagged cache. We assume a 32-bit virtual address space, with 4KB pages. Upon a translation, the high-order 20 bits of the virtual address—the virtual tag—is forwarded to the translation lookaside buffer (TLB), if the system has one (refer to **(a)** in Figure 2-4). On a TLB hit **(b)**, the physical tag is combined with the 12 low-order bits of the virtual address (the page offset, **(c)**) to produce the physical address. On a TLB miss **(d)**, or if the system has no TLB **(e)**, the virtual tag is shifted right 10 bits to produce the virtual address **(f)** of the page table entry (PTE). The page table occupies the low 4 MB in the virtual address space. Once the virtual address of the PTE is obtained, it must also be translated to produce the physical address of the PTE. To do this translation, the high-order 20 bits (actually bits 13-22, since the high-order 10 bits are zero) are passed to a table that we call the MMU (for memory management unit), which holds 1024 virtual to phys-

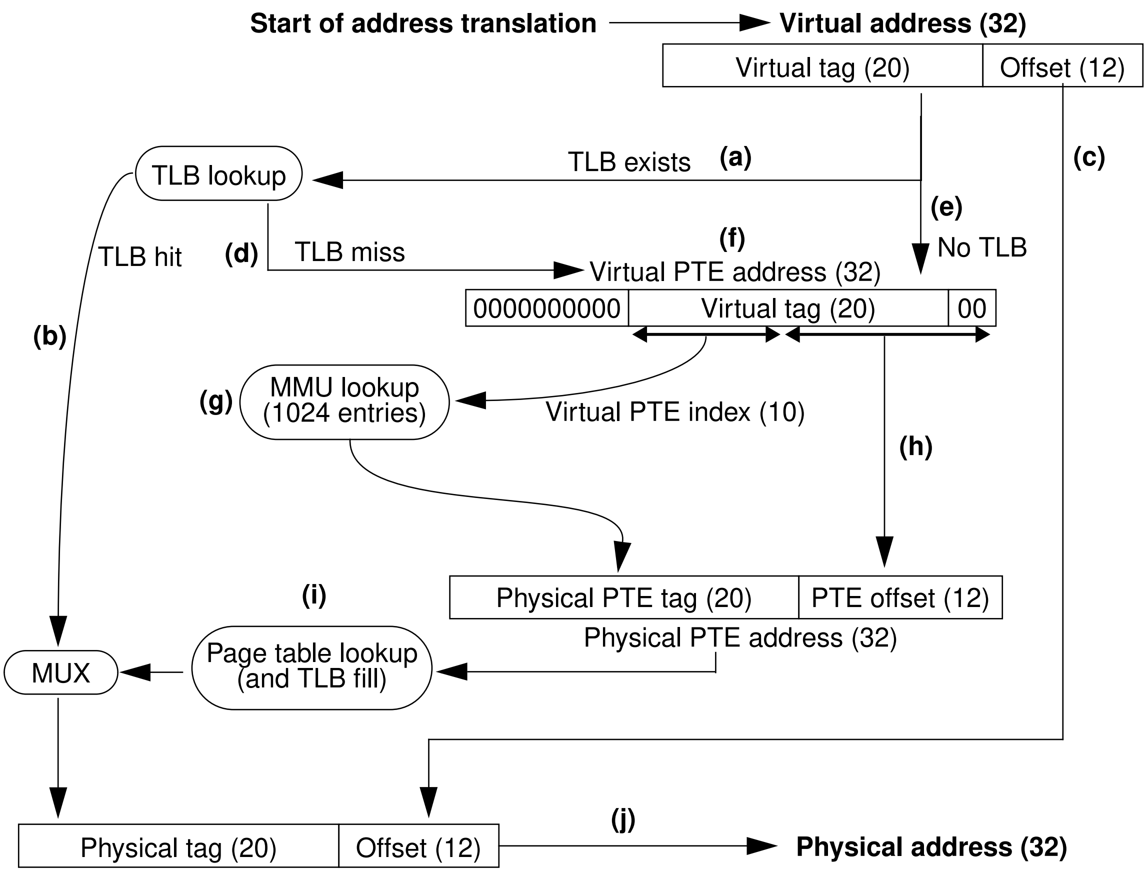


Figure 2-4: Virtual memory organization

ical mappings of PTE pages **(g)**. Since the PTEs are each 4 bytes, each page of PTEs can thus hold 1024 PTEs. Since each PTE maps one page (4 KB), each page of PTEs maps 4MB (4KB * 1024). Since the MMU holds translations for 1K PTE pages, it can cover 4GB of virtual address space, which is complete coverage for a 32-bit address space. Once the physical tag for the PTE page is obtained from the MMU, it is concatenated with the offset from the PTE virtual address **(h)** to obtain the PTE physical address. With that address, the PTE can be obtained, providing the physical tag for the requested translation. If the system has a TLB, the PTE is loaded into the TLB **(i)**. The memory access then continues using the required physical address **(j)**.

2.2.2 Functional simulation

Functional simulation merely executes the benchmark program operations without accounting for time (*i.e.*, how long it takes to execute those instructions). The SimpleScalar tools include two functional simulators that we use in this dissertation, **sim-profile** and **sim-cache**. **sim-profile** maintains statistics based on individual instructions, which we use to characterize our benchmarks later in this chapter. **sim-cache** tracks miss ratios for a functional cache module, which does not account for contention or finite resources (having no notion of time). Our modified version of **sim-cache** supports virtually or physically tagged caches, and as many levels of cache as desired connected in an arbitrary topology. These simulations run roughly an order of magnitude faster than the timing simulator described in the following section.

2.2.3 Timing simulation

The timing simulator (**sim-outorder**) models a dynamically scheduled microprocessor, performing cycle-by-cycle simulation at a high level. The effects of circuit technology are not modeled; all delays are specified whole numbers of cycles. The simulated microprocessor is a five-stage execution pipeline (with a sixth stage for commitment of instructions), depicted in Figure 2-5. Instructions are fetched, and the branch predictor accessed to determine a speculative address from which to fetch on branches. The *fetch engine* is set to run at an integer multiple of the core speed, and can fetch across one fewer taken branches than the ratio of fetch engine speed to core speed. In all our simulations, we assumed that the fetch and core speeds were identical, so a taken branch would terminate fetches within a given cycle.

Once fetched, instructions are decoded and sent to the reservation stations in the *dispatch* stage of the pipeline. The execution core of **sim-outorder** is derived from the Register Update Unit (RUU) [113], depicted in Figure 2-6. The RUU is a centralized structure that effectively acts as a combined register renaming unit, reservation station pool [123], and reorder buffer [110, 115]. The RUU is implemented as a circular queue, with head and tail pointers. The tail pointer is advanced as new instructions are dispatched to the RUU, and the head moves as the

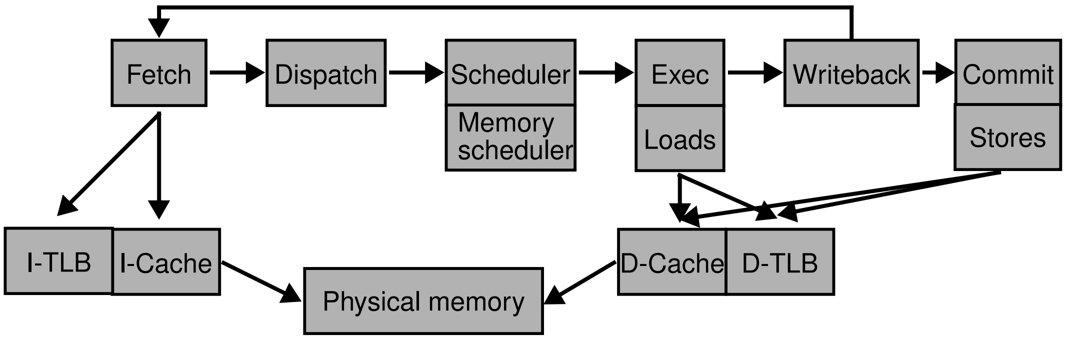


Figure 2-5: Pipeline for sim-outorder

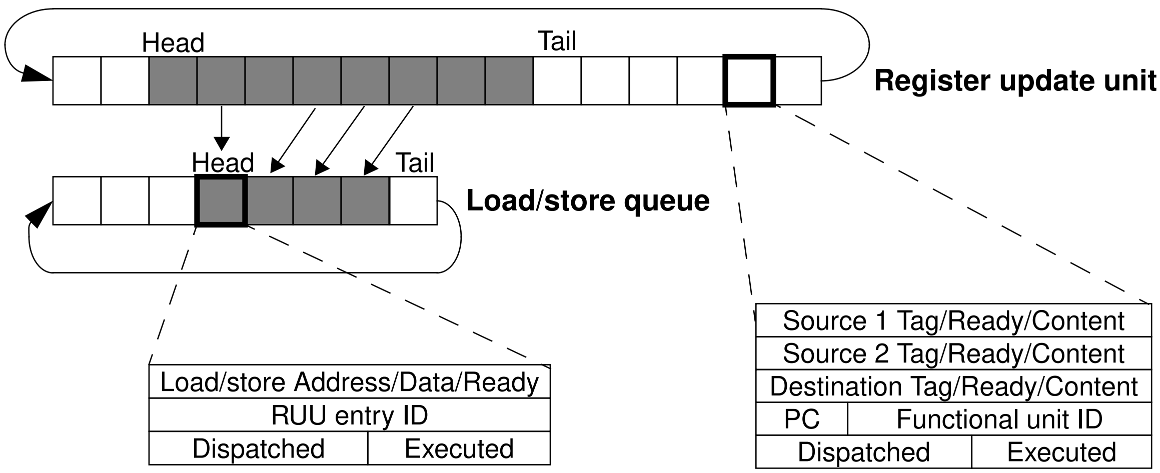


Figure 2-6: Structure of the Register Update Unit core

oldest instructions are committed to the architectural state. Operands are stored in the RUU, and are identified with unique tags to preserve data dependences. Once an operation's input operands are all available, it is marked as ready for issue. Each cycle, a number of ready instructions are issued to the functional units through the scheduler, shown in Figure 2-5. Branches, memory operations, and long latency operations (such as multiplies) are always inserted directly at the head of the ready queue since they are most likely to be on the program's critical path. All other instructions are queued so that they are issued in program order.

When a load is dispatched to the issue units, it is split into two components. A slot is reserved in the RUU for the effective address computation, and a slot is reserved in a companion structure called the *load/store queue* (LSQ), which performs the actual communication to memory. The LSQ is responsible for identifying which loads may be sent to memory—loads are not issued if an earlier store with an unresolved address is in the LSQ. (A more aggressive implementation might perform data dependence speculation [89], allowing the loads to issue speculatively). If an earlier store's address is resolved and it matches an unissued load's address, the value is forwarded to the load directly in the LSQ. Our simulator does not consider whether a value is a partial word (*e.g.*, store byte instructions) when matching addresses on word boundaries, which introduces some inaccuracy. Another potential source of inaccuracy occurs when the program uses a double word (held in two registers) as an input; the input dependence tracking in the simulator only creates a dependence link for one of the two registers holding the double word. If the two halves become available at different times, this assumption may allow the dependent instruction to issue early.

The simulator does not issue ready instructions if there are insufficient resources available (functional units or cache ports) for that class of instruction. Instructions that are blocked due to insufficient resources are returned to the ready list; the issue unit attempts to reissue the ready instructions each successive cycle.

The *writeback* stage of the pipeline is that which returns computed values to the execution core. When an instruction completes, its result is written back on the result bus (to the RUU), the value is copied into the RUU entries of the instructions that depend on that result, and those instructions were waiting solely for the result in question are marked as ready for issue. This stage is the point at which mispredicted branches are resolved, so pipeline flushes occur when identified in the writeback stage.

The final stage of the pipeline, which is generally off the execution's critical path, is the *commit* stage. In this stage, results are written back to the RUU in program order. We assume in our simulations that the number of instructions that can be committed to the architectural register file each cycle is the same as the fetch and issue widths. It is in this stage that stores are issued to the memory system, since they are guaranteed not to be mis-speculative stores. Retirement of instructions can be blocked if a store takes a cache or TLB miss, or if there are insufficient store ports to the memory system. The commit stage can affect program performance when it is blocked for enough time (*e.g.* a long latency cache miss) that the RUU or LSQ fills up, preventing instructions from being dispatched. Streams of stores cause a similar effect even without cache misses, since the number of cache store ports we assume is less than the commit width.

Our simulations model user-level programs down to the physical memory. We do not currently simulate disk accesses (demand paging), nor do we simulate operating system code. System calls in SimpleScalar are handled through *proxy system calls*, in which a system call generated by a SimpleScalar binary is intercepted by the simulator, translated into an equivalent call on the host system, and then called directly on the host. Upon completion of the system call, the results are copied back into the appropriate registers for the target system, and simulation resumes. From the target's perspective, it appears as if system calls occur instantaneously.

2.3 SPEC95 benchmarks

The benchmarks we use throughout this dissertation are those from the SPEC95 suite. These benchmarks are well understood by the architecture research community, and consist of several different application types. There are 18 total benchmarks in the suite; 8 integer benchmarks (SPECINT95) and 10 floating-point benchmarks (SPECFP95). These benchmarks are not without their problems; their data sets (particularly their code sizes) are much smaller than many applications today, as are their corresponding footprints in memory. However, the difficulty of obtaining sources (or traces, for that matter) of current-generation industrial applications restricts us to using these benchmarks for this dissertation.

2.3.1 Choosing the input set

Each of the SPEC95 benchmarks is distributed with three data sets: a test set (**test**), a training set (**train**), and a reference data set (**ref**). The test inputs are intended as small inputs that allow the user to see if the benchmark runs to completion. The training inputs were intended for use in training a compiler with profile-directed feedback. Both data sets are intended to be significantly smaller than the reference data set, which is the data set intended for actually running to measure the performance of various machine configurations. However, since **ref** was intended for long runs on real machines, most of the reference sets take too long to simulate to completion with timing simulation. The simulation times with **test** and **train** tend to be more tractable, but they may not provide an accurate characterization of real programs' execution (either because their data sets are too small, or because they are dominated by initialization code). To choose which inputs to use, we take a two-tiered approach. For the floating point codes, most of which are loop-bound, we use the **ref** data set with reduced numbers of iterations. We ensure that the number of iterations is sufficiently large that initialization is no more than 10% of the total running time (with a few exceptions as discussed in Section 2.3.3). For the integer codes and the floating point codes that are not amenable to reduced numbers of iterations, we profile the three data sets and use the data set that requires the smallest number of instructions while exhibiting behavior similar to that of **ref**.

When trying to determine how long a program we can simulate, we must consider the speeds of the various simulators. In Table 2-2, we list the simulation speed ranges of the five simulators we use (the speed of each one varies depending on the execution characteristics of the benchmark). We also list the low and high times required to simulate one billion instructions with each simulator. We took the measurements on a 266-MHz Pentium II. The slowest, most detailed simulator required approximately one day for each billion instructions simulated.

2.3.2 Benchmark characterizations

In this subsection, we characterize the behavior of each benchmark experimentally. We choose one input set for each experiment (trying to minimize the number of instructions simulated while maintaining behavior similar to simulating the **ref** data set in full). We refer to this set collectively as the *standard* input set (or **std**). We choose the **std** input set for each benchmark based on profiled program characteristics. Our goal is to simulate benchmarks with as large a data set as possible (since the SPEC95 data sets are already generally smaller than typical applications today), but to simulate as few instructions as possible while retaining the behavior typical of the full application. Since the focus of this dissertation is the PMI, we want to choose the workloads that stress the memory system. We therefore choose the data sets that

Simulator	sim-fast	sim-cheetah	sim-cache	sim-profile	sim-outorder
Speed (insts/s)	2M-3M	400K-700K	200K-400K	30K-300K	10K-80K
Time/G inst (low)	8.3 min.	41.7 min.	1.4 hr.	9.3 hr.	27.8 hr.
Time/G inst (high)	5.6 min.	23.8 min.	41.7 min.	55.6 min.	3.5 hr.

Table 2-2: Simulation speeds of the five simulators

sustain high miss rates, but must ensure that we do not reduce execution so that compulsory (cold start) misses inflate the cache miss rates.

We list the number of instructions for each input set of SPECINT95 in Table 2-3, as well as the breakdown of instructions into memory, computation, and control (we obtained all results in the following four tables with **sim-profile**). In Table 2-4, we display similar statistics for SPECFP95. In all subsequent tables in this section, we will represent the **std** input set in boldface. These profiles show that the integer codes are much more control-bound than the floating-point codes; the integer codes' instructions are generally 15%-25% control, with the one exception being ijpeg, for which control instructions account for about 8% of the total. There is more variance among the distribution of computation versus memory for the integer codes; the over half of the vortex instructions are memory operations, whereas the memory instructions for ijpeg account for about a quarter of the total. The rest of the benchmarks fall somewhere in between. The floating-point codes have more consistent distributions; they typically have between 25% and 35% memory operations (the one exception is fpppp, at about 53%), less than 8% control instructions (the sole exception is hydro2d, at 12%), and high percentages of computation (greater than 60%, except for fpppp, at 45%).

The instruction counts listed range from 3.5M (compress with the **test** input) to 175G (fpppp with the **ref** input). The **std** inputs we chose (described in more detail in Section 2.3) have a maximum instruction count of 16G instructions (go), placing an upper bound of 18 days of simulation time for any benchmark with the slowest simulator. On average, the **std** inputs run for approximately 29 hours with **sim-outorder**, and about 4 hours with **sim-cache**.

In Table 2-5, we list memory operation profiles for SPECINT95, showing the breakdown of memory operations into loads and stores, plus the distribution of memory operations to the data, stack, and heap segments. In Table 2-6, we show the same results for SPECFP95. The

benchmark	input	inst	%comp	%mem	%ctrl
099.go	**test**	**16389.6**	**0.563**	**0.290**	**0.148**
	train	548.1	0.567	0.287	0.146
	ref	33119.1	0.564	0.288	0.148
124.m88ksim	test	416.5	0.474	0.311	0.216
	train	**111.9**	**0.483**	**0.333**	**0.184**
	ref	63408.5	0.460	0.350	0.190
126.gcc	test	1265.2	0.396	0.405	0.199
	train	1277.6	0.389	0.409	0.201
	ref	**1023.2**	**0.400**	**0.403**	**0.198**
129.compress	test (100)	3.5	0.292	0.611	0.096
	train (10K)	35.7	0.454	0.374	0.172
	std (400K)	**1257.5**	**0.472**	**0.320**	**0.208**
	ref (14M)	43064.8	0.473	0.324	0.204
130.li	test	956.7	0.288	0.476	0.236
	train	**183.3**	**0.347**	**0.425**	**0.228**
	ref	76570.0	0.332	0.430	0.238
132.ijpeg	test	553.1	0.652	0.255	0.093
	train	**1462.5**	**0.664**	**0.255**	**0.081**
	ref1 (vigo)	30819.9	0.668	0.258	0.074
	ref2 (specmun)	27011.4	0.670	0.258	0.072
	ref3 (penguin)	29810.1	0.671	0.256	0.073
134.perl	test	10.5	0.349	0.447	0.204
	train	**2391.5**	**0.370**	**0.436**	**0.193**
	ref1 (primes)	14282.3	0.330	0.480	0.190
	ref2 (scrabble)	24240.3	0.345	0.462	0.193
147.vortex	test	9051.6	0.309	0.526	0.165
	train	2520.2	0.308	0.528	0.164
	ref.1it	**7712.7**	**0.309**	**0.526**	**0.165**
	ref (14 it)	74014.3	0.312	0.514	0.174

Table 2-3: Instruction profile for SPECINT95

integer benchmarks tend to have a higher percentage of stores than the floating-point benchmarks; the floating benchmarks tend to use about 20%-25% stores (75%-80% loads), with two notable exceptions: mgrid (96%/4% loads/stores) and turb3d (60%/40% loads/stores). The integer codes are roughly 63% loads, except for ijpeg and go, which are 70% and 75% loads, respectively.

The distribution of the memory operations among the data segment, heap, and stack vary widely across the benchmarks, particularly the integer codes. The floating point codes tend to make a much higher use of the data segment (a notable exception is tomcatv, which issues over 90% of its memory operations to the stack), and they almost never access the heap.

benchmark	input	inst	%comp	%mem	%ctrl
101.tomcatv	test	2798.9	0.532	0.310	0.157
	train	17660.7	0.715	0.259	0.026
	ref.62it	**10651.7**	**0.673**	**0.270**	**0.057**
	ref (750 it)	105323.2	0.718	0.258	0.025
102.swim	test	849.9	0.630	0.310	0.060
	train	849.9	0.630	0.310	0.060
	ref.45it	**2846.2**	**0.650**	**0.322**	**0.028**
	ref (900 it)	51613.0	0.659	0.327	0.015
103.su2cor	test	1054.1	0.583	0.329	0.088
	train	19851.1	0.614	0.324	0.062
	ref.5it	**11548.7**	**0.589**	**0.327**	**0.084**
	ref (40 it)	62616.3	0.614	0.324	0.062
104.hydro2d	test	974.5	0.600	0.264	0.136
	train	7583.0	0.635	0.247	0.118
	ref.6it	**2443.1**	**0.624**	**0.252**	**0.124**
	ref (200 it)	73666.6	0.639	0.244	0.116
107.mgrid	test	4422.3	0.619	0.367	0.013
	test.4it	**480.3**	**0.621**	**0.363**	**0.017**
	train	14292.1	0.622	0.363	0.015
	ref (40 it)	110556.9	0.619	0.367	0.013
110.applu	test	19408.1	0.712	0.255	0.034
	train	531.9	0.711	0.255	0.034
	ref.5it	**1748.1**	**0.713**	**0.254**	**0.033**
	ref (300 it)	93423.3	0.712	0.255	0.034
125.turb3d	test, train	17120.6	0.720	0.227	0.052
	ref.2it	**2836.8**	**0.717**	**0.230**	**0.052**
	ref (111 it)	169598.6	0.717	0.231	0.053
141.apsi	test	9191.7	0.639	0.316	0.046
	train	2350.0	0.623	0.323	0.054
	ref.6it	**318.2**	**0.643**	**0.310**	**0.047**
	ref (960 it)	47883.2	0.648	0.311	0.041
145.fpppp	test	1872.3	0.456	0.531	0.013
	train	**331.1**	**0.454**	**0.532**	**0.014**
	ref	175465.0	0.464	0.520	0.016
146.wave5	test	4627.1	0.605	0.324	0.071
	train	3132.8	0.603	0.318	0.079
	ref.10it	**13072.9**	**0.608**	**0.332**	**0.060**
	ref (40 it)	44888.9	0.610	0.337	0.053

Table 2-4: Instruction profile for SPECFP95

In Table 2-7, we show the sizes of each segment for SPECINT95 (text, data, heap, and stack segments). We show the sum of these four segments, and compare that with the total data set that was statically allocated for each benchmark. Simply examining the size of the statically allocated segments is insufficient because most of the FORTRAN benchmarks (and some of

benchmark	input	%loads	%stores	%data	%heap	%stack
099.go	**test**	**0.737**	**0.263**	**0.679**	**0.000**	**0.321**
	train	0.737	0.263	0.668	0.000	0.332
	ref	0.741	0.259	0.687	0.000	0.313
124.m88ksim	test	0.669	0.331	0.656	0.075	0.269
	train	**0.615**	**0.385**	**0.355**	**0.112**	**0.533**
	ref	0.638	0.362	0.501	0.052	0.447
126.gcc	test	0.637	0.363	0.160	0.215	0.625
	train	0.649	0.351	0.162	0.222	0.616
	ref	**0.638**	**0.362**	**0.164**	**0.216**	**0.620**
129.compress	test	0.123	0.877	0.973	0.003	0.025
	train	0.552	0.448	0.914	0.000	0.085
	std	**0.649**	**0.351**	**0.925**	**0.000**	**0.075**
	ref	0.644	0.356	0.925	0.000	0.075
130.li	test	0.629	0.371	0.182	0.362	0.456
	train	**0.610**	**0.390**	**0.163**	**0.395**	**0.442**
	ref	0.634	0.366	0.138	0.451	0.411
132.ijpeg	test	0.692	0.308	0.035	0.598	0.366
	train	**0.699**	**0.301**	**0.032**	**0.647**	**0.321**
	ref1 (vigo)	0.703	0.297	0.030	0.657	0.312
	ref2 (specmun)	0.705	0.295	0.030	0.670	0.300
	ref3 (penguin)	0.704	0.296	0.030	0.662	0.308
134.perl	test	0.613	0.387	0.130	0.337	0.533
	train	**0.591**	**0.409**	**0.112**	**0.392**	**0.495**
	ref1 (primes)	0.633	0.367	0.130	0.297	0.573
	ref2 (scrabble)	0.607	0.393	0.140	0.361	0.499
147.vortex	test	0.586	0.414	0.120	0.150	0.730
	train	0.581	0.419	0.116	0.152	0.732
	ref.1it	**0.584**	**0.416**	**0.120**	**0.149**	**0.731**
	ref	0.619	0.381	0.138	0.167	0.695

Table 2-5: Memory operation profile for SPECINT95

the integer codes) statically allocate some maximum data set, but access only an input-dependent fraction. We measured accessed regions of memory at a 4KB (page) granularity (*i.e.*, if a single word in a single page is touched, that 4KB page is counted toward the total). This metric thus quantifies the application's footprint in physical memory. In Table 2-8, we show the same statistics for SPECFP95. We obtained these numbers using a modified version of the **sim-cache** simulator.

The **std** data set sizes vary widely across the benchmarks as well. li and fpppp have data sets of less than 1MB. Most of the integer codes have data sets between 1MB and 10MB: go, m88ksim, gcc, compress, and ijpeg. apsi, hydro2d, and mgrid are the floating-point bench-

benchmark	input	%loads	%stores	%data	%heap	%stack
101.tomcatv	test	0.674	0.326	0.287	0.050	0.663
	train	0.794	0.206	0.028	0.002	0.970
	ref.62it	**0.762**	**0.238**	**0.097**	**0.015**	**0.888**
	ref (750 it)	0.796	0.204	0.024	0.002	0.975
102.swim	test	0.778	0.222	0.818	0.000	0.182
	train	0.778	0.222	0.818	0.000	0.182
	ref.45it	**0.806**	**0.194**	**0.948**	**0.000**	**0.052**
	ref (900 it)	0.816	0.184	0.997	0.000	0.003
103.su2cor	test	0.756	0.244	0.319	0.026	0.655
	train	0.767	0.233	0.343	0.003	0.653
	ref.5it	**0.756**	**0.244**	**0.334**	**0.019**	**0.647**
	ref (40 it)	0.768	0.232	0.342	0.003	0.655
104.hydro2d	test	0.763	0.237	0.802	0.022	0.177
	train	0.807	0.193	0.936	0.003	0.062
	ref.6it	**0.792**	**0.208**	**0.892**	**0.009**	**0.099**
	ref (200 it)	0.813	0.187	0.955	0.000	0.045
107.mgrid	test	0.962	0.038	0.784	0.000	0.216
	test.4it	**0.954**	**0.046**	**0.777**	**0.000**	**0.223**
	train	0.960	0.040	0.783	0.000	0.217
	ref (40 it)	0.962	0.038	0.784	0.000	0.216
110.applu	test	0.815	0.185	0.667	0.000	0.333
	train	0.814	0.186	0.667	0.000	0.333
	ref.5it	**0.817**	**0.183**	**0.669**	**0.000**	**0.331**
	ref (300 it)	0.815	0.185	0.667	0.000	0.333
125.turb3d	test	0.610	0.390	0.218	0.000	0.782
	train	0.610	0.390	0.218	0.000	0.782
	ref.2it	**0.607**	**0.393**	**0.217**	**0.000**	**0.783**
	ref	0.606	0.394	0.211	0.000	0.789
141.apsi	test	0.724	0.276	0.641	0.000	0.359
	train	0.712	0.288	0.584	0.000	0.416
	ref.6it	**0.725**	**0.275**	**0.633**	**0.002**	**0.365**
	ref	0.731	0.269	0.660	0.000	0.340
145.fpppp	test	0.725	0.275	0.420	0.000	0.580
	train	**0.722**	**0.278**	**0.420**	**0.000**	**0.580**
	ref	0.733	0.267	0.418	0.000	0.582
146.wave5	test	0.722	0.278	0.889	0.000	0.111
	train	0.717	0.283	0.848	0.000	0.152
	ref.10it	**0.732**	**0.268**	**0.930**	**0.000**	**0.070**
	ref	0.736	0.264	0.960	0.000	0.040

Table 2-6: Memory operation profile for SPECFP95

benchmark	input	text	data	heap	stack	total	allocated
099.go	**test**	**580 K**	**524 K**	**20 K**	**8 K**	**1.1 M**	**1.1 M**
	train	560 K	496 K	24 K	8 K	1.0 M	1.1 M
	ref	584 K	528 K	24 K	8 K	1.1 M	1.1 M
124.m88ksim	test	248 K	128 K	472 K	12 K	860 K	918 K
	train	**252 K**	**128 K**	**3.8 M**	**12 K**	**4.1 M**	**4.2 M**
	ref	268 K	128 K	18.5 M	12 K	18.9 M	18.9 M
126.gcc	test	1.9 M	252 K	1.6 M	308 K	4.1 M	3.9 M
	train	1.9 M	252 K	1.3 M	200 K	3.7 M	3.6 M
	ref	**1.9 M**	**252 K**	**2.8 M**	**568 K**	**5.5 M**	**5.1 M**
129.compress	test	80 K	536 K	20 K	8 K	644 K	42.2 M
	train	80 K	640 K	20 K	8 K	748 K	42.2 M
	std	**80 K**	**1.5 M**	**20 K**	**8 K**	**1.6 M**	**42.2 M**
	ref	80 K	34.8 M	20 K	8 K	34.9 M	42.2 M
130.li	test	152 K	20 K	84 K	12 K	268 K	304 K
	train	**144 K**	**20 K**	**160 K**	**28 K**	**352 K**	**380 K**
	ref	156 K	20 K	392 K	28 K	596 K	612 K
132.ijpeg	test	268 K	36 K	4.3 M	12 K	4.6 M	21.0 M
	train	**268 K**	**40 K**	**7.8 M**	**12 K**	**8.1 M**	**24.5 M**
	ref (vigo)	268 K	224 K	7.4 M	12 K	7.9 M	25.6 M
	ref (specmun)	268 K	172 K	6.6 M	12 K	7.1 M	24.7 M
	ref (penguin)	268 K	196 K	7.1 M	12 K	7.6 M	25.3 M
134.perl	test	392 K	72 K	56 K	8 K	528 K	685 K
	train	**432 K**	**72 K**	**25.0 M**	**8 K**	**25.5 M**	**25.6 M**
	ref (primes)	392 K	72 K	56 K	8 K	528 K	625 K
	ref (scrabble)	428 K	72 K	18.4 M	12 K	18.9 M	19.0 M
147.vortex	test	896 K	116 K	25.2 M	12 K	26.2 M	26.3 M
	train	896 K	116 K	10.3 M	12 K	11.3 M	11.4 M
	ref.1it	**896 K**	**1116 K**	**29.1 M**	**12 K**	**30.1 M**	**30.2 M**
	ref	896 K	116 K	45.7 M	12 K	46.7 M	46.8 M

Table 2-7: Data set and segment sizes for SPECINT95

marks that fall into that category. Tomcatv and Swim are the two codes whose data sets fall between 10MB and 20MB. The six benchmarks with the largest data sets, all over 20MB, are perl, vortex, su2cor, applu, turb3D, and wave5.

In Table 2-9 and Table 2-10, we list cache miss rates for the **std** inputs of SPECINT95 and SPECFP95, respectively. We show miss rates for direct-mapped, write-allocate caches with 32-bytes blocks, and sizes ranging from 4KB to 1MB. Dotted lines denote cache sizes that are larger than the data set sizes, which we therefore did not simulate. We obtained these results using **sim-cheetah**, which couples the SimpleScalar functional simulator with the Cheetah cache simulation library developed at Michigan [119]. In Appendix B (Section B.3), we vali-

benchmark	input	text	data	heap	stack	total	allocated
101.tomcatv	test	160 K	28 K	36 K	14.0 M	14.2 M	14.3 M
	train	160 K	28 K	36 K	7.0 M	7.2 M	14.3 M
	ref.62it	**160 K**	**28 K**	**36 K**	**14.0 M**	**14.2 M**	**14.3 M**
	ref	160 K	28 K	36 K	14.0 M	14.2 M	14.3 M
102.swim	test	160 K	14.0 M	24 K	12 K	14.2 M	14.2 M
	train	160 K	14.0 M	24 K	12 K	14.2 M	14.2 M
	ref.45it	**160 K**	**14.0 M**	**24 K**	**12 K**	**14.2 M**	**14.2 M**
	ref	160 K	14.0 M	24 K	12 K	14.2 M	14.2 M
103.su2cor	test	256 K	2.2 M	36 K	5.7 M	8.2 M	8.6 M
	train	256 K	3.7 M	36 K	8.3 M	12.4 M	8.6 M
	ref.5it	**256 K**	**8.3 M**	**36 K**	**13.6 M**	**22.2 M**	**8.6 M**
	ref	256 K	8.3 M	36 K	13.6 M	22.2 M	8.6 M
104.hydro2d	test	208 K	8.4 M	40 K	16 K	8.6 M	8.7 M
	train	208 K	8.4 M	40 K	16 K	8.6 M	8.7 M
	ref.6it	**208 K**	**8.4 M**	**48 K**	**16 K**	**8.6 M**	**8.7 M**
	ref	208 K	8.4 M	40 K	16 K	8.6 M	8.7 M
107.mgrid	test	168 K	7.3 M	24 K	12 K	7.5 M	7.5 M
	test.4it	**168 K**	**7.3 M**	**24 K**	**12 K**	**7.5 M**	**7.5 M**
	train	168 K	1.0 M	24 K	12 K	1.2 M	7.5 M
	ref	168 K	7.3 M	24 K	12 K	7.5 M	7.5 M
110.applu	test	228 K	13.5 M	24 K	28 K	13.7 M	31.8 M
	train	228 K	3.0 M	24 K	28 K	3.2 M	31.8 M
	ref.5it	**228 K**	**28.7 M**	**24 K**	**28 K**	**29.0 M**	**31.8 M**
	ref	228 K	28.7 M	24 K	28 K	29.0 M	31.8 M
125.turb3d	test	228 K	24.7 M	36 K	12 K	25.0 M	25.0 M
	train	228 K	24.7 M	36 K	12 K	25.0 M	25.0 M
	ref.2it	**228 K**	**24.7 M**	**36 K**	**12 K**	**25.0 M**	**25.0 M**
	ref	228 K	24.7 M	44 K	12 K	25.0 M	25.0 M
141.apsi	test	340 K	556 K	48 K	16 K	960 K	9.6 M
	train	340 K	184 K	48 K	16 K	588 K	9.6 M
	ref.6it	**340 K**	**1.9 M**	**48 K**	**16 K**	**2.3 M**	**9.6 M**
	ref	340 K	1.9 M	48 K	16 K	2.3 M	9.6 M
145.fpppp	test	284 K	140 K	24 K	24 K	472 K	803 K
	train	**284 K**	**136 K**	**24 K**	**24 K**	**468 K**	**803 K**
	ref	284 K	232 K	24 K	24 K	564 K	803 K
146.wave5	test	312 K	27.2 M	32 K	12 K	27.5 M	41.2 M
	train	312 K	27.2 M	32 K	12 K	27.5 M	41.2 M
	ref.10it	**308 K**	**40.1 M**	**36 K**	**12 K**	**40.5 M**	**41.2 M**
	ref	308 K	40.1 M	32 K	12 K	40.5 M	41.2 M

Table 2-8: Data set and segment sizes for SPECFP95

benchmark	input	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
099.go	**test**	**28.007**	**21.403**	**9.971**	**5.468**	**3.035**	**1.681**	**1.481**	**0.001**	**0.000**
	train	24.305	18.014	6.070	2.935	1.590	0.097	0.065	0.009	0.004
	ref	28.974	22.209	10.644	5.772	3.255	1.846	1.587	0.001	0.000
124.m88ksim	test	4.546	2.564	1.522	0.904	0.426	0.141	0.132	0.007	------
	train	**3.268**	**2.407**	**1.111**	**0.669**	**0.528**	**0.423**	**0.334**	**0.328**	**0.326**
	ref	4.016	2.583	1.173	0.556	0.313	0.052	0.008	0.007	------
126.gcc	test	7.951	5.146	3.265	1.975	1.043	0.619	0.359	0.128	0.064
	train	8.332	5.218	3.197	1.960	1.021	0.553	0.309	0.096	0.060
	ref	**8.136**	**5.385**	**3.428**	**2.143**	**1.126**	**0.735**	**0.465**	**0.215**	**0.109**
129.compress	test	5.617	5.519	5.466	5.427	5.380	5.162	1.113	0.369	------
	train	7.873	6.157	4.912	3.654	2.643	1.539	0.920	0.126	------
	std	**15.722**	**13.458**	**11.758**	**9.745**	**7.858**	**5.407**	**2.561**	**0.228**	**0.168**
	ref	15.137	12.851	11.166	9.215	7.399	5.121	2.642	0.206	0.165
130.li	test	3.829	2.241	1.127	0.476	0.016	0.000	0.000	------	------
	train	**4.929**	**3.231**	**2.178**	**1.464**	**0.810**	**0.136**	**0.004**	**------**	**------**
	ref	4.912	3.085	2.152	1.519	1.035	0.585	0.125	**------**	------
132.ijpeg	test	9.607	3.577	1.843	0.826	0.552	0.360	0.278	0.233	0.217
	train	**10.499**	**3.988**	**1.837**	**1.148**	**0.795**	**0.638**	**0.515**	**0.465**	**0.449**
	ref1	18.107	8.171	4.175	1.171	0.469	0.349	0.255	0.230	0.210
	ref2	17.596	8.371	4.343	1.336	0.676	0.444	0.281	0.235	0.216
	ref3	16.069	8.243	4.223	1.200	0.873	0.340	0.278	0.252	0.215
134.perl	test	6.817	3.014	1.790	1.304	0.869	0.778	0.021	0.021	0.021
	train	**5.688**	**3.145**	**2.150**	**1.679**	**0.801**	**0.495**	**0.257**	**0.205**	**0.165**
	ref1	6.108	2.841	1.038	0.779	0.007	0.006	0.000	0.000	0.000
	ref2	8.934	5.944	3.880	2.443	0.829	0.654	0.019	0.016	0.014
147.vortex	test	6.955	5.103	3.141	1.464	0.922	0.519	0.318	0.215	0.133
	train	7.342	5.537	4.263	2.356	1.738	0.538	0.364	0.229	0.143
	ref.1it	**7.017**	**5.094**	**2.548**	**1.700**	**1.184**	**0.794**	**0.464**	**0.350**	**0.161**
	ref	6.772	3.469	2.365	1.669	1.135	0.720	0.480	0.317	0.217

Table 2-9: Cache miss rates for varied SPECINT95 data sets (data stream)

date the Cheetah simulation by comparing it with miss rates from **sim-cache** (and vice-versa). Also in Appendix B, we provide a comprehensive set of cache miss rates for SPEC95, showing miss rates for varied associativities (Section B.1) and block sizes (Section B.2), using three reference streams (instruction, data, and unified).

2.3.3 SPEC95 benchmark analysis

In this subsection, we describe each of the benchmarks (the eight integer benchmarks followed by the ten floating-point benchmarks). We justify our choice of the **std** input set for each benchmark, and characterize each benchmark's behavior with that input set. Most of the

benchmark	input	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
101.tomcatv	test	8.955	7.561	4.275	1.933	1.175	1.157	1.145	1.137	1.126
	train	24.021	22.557	14.626	6.774	4.134	4.102	4.063	4.020	3.982
	ref.62it	**20.989**	**19.507**	**12.542**	**5.817**	**3.534**	**3.486**	**3.457**	**3.433**	**3.401**
	ref	24.550	23.001	14.960	6.954	4.223	4.167	4.133	4.104	4.066
102.swim	test	49.698	39.780	21.024	6.658	2.015	1.989	1.976	1.968	1.960
	train	49.698	39.780	21.024	6.658	2.015	1.989	1.976	1.968	1.960
	ref.45it	**65.062**	**52.319**	**27.894**	**8.314**	**2.299**	**2.265**	**2.248**	**2.239**	**2.234**
	ref	70.934	57.111	30.520	8.947	2.408	2.371	2.353	2.343	2.338
103.su2cor	test	10.110	8.058	7.279	6.693	2.350	1.883	1.372	0.640	0.286
	train	9.940	8.465	7.794	7.326	2.381	2.005	1.557	1.020	0.442
	ref.5it	**9.775**	**7.843**	**7.136**	**6.623**	**2.229**	**2.051**	**1.740**	**1.201**	**0.675**
	ref	9.952	8.440	7.811	7.311	2.406	2.200	1.862	1.371	0.760
104.hydro2d	test	5.203	4.258	3.539	2.880	2.728	2.660	2.636	2.523	2.289
	train	5.520	4.578	3.990	3.297	3.158	3.076	3.049	2.932	2.653
	ref.6it	**5.425**	**4.482**	**3.855**	**3.173**	**3.029**	**2.952**	**2.925**	**2.810**	**2.544**
	ref	5.561	4.619	4.047	3.350	3.211	3.128	3.100	2.983	2.698
107.mgrid	test	5.934	2.620	1.865	1.457	1.235	0.966	0.901	0.596	0.566
	test.4it	**5.941**	**2.635**	**1.884**	**1.480**	**1.259**	**0.992**	**0.928**	**0.632**	**0.602**
	train	5.409	3.986	3.126	2.447	2.248	2.136	2.033	0.355	0.147
	ref	5.934	2.620	1.865	1.437	1.235	0.966	0.901	0.596	0.566
110.applu	test	5.092	2.630	1.913	1.573	1.380	1.266	1.226	1.184	1.098
	train	4.949	2.549	1.902	1.574	1.387	1.208	1.054	0.761	0.470
	ref.5it	**5.105**	**2.571**	**1.839**	**1.494**	**1.319**	**1.228**	**1.179**	**1.150**	**1.094**
	ref	5.194	2.677	1.934	1.574	1.393	1.299	1.250	1.220	1.175
125.turb3d	test	4.065	3.461	3.255	2.158	1.364	1.271	0.871	0.394	0.386
	train	4.065	3.461	3.255	2.158	1.364	1.271	0.871	0.394	0.386
	ref.2it	**4.010**	**3.408**	**3.202**	**2.111**	**1.426**	**1.345**	**0.909**	**0.409**	**0.398**
	ref	3.839	3.228	3.019	1.908	1.293	1.198	0.801	0.323	0.315
141.apsi	test	6.995	5.911	5.646	4.450	2.943	1.673	0.816	0.056	0.001
	train	6.306	5.369	3.070	1.731	0.838	0.119	0.000	0.000	0.000
	ref.6it	**11.056**	**5.675**	**4.945**	**4.832**	**4.572**	**4.408**	**2.914**	**1.630**	**0.757**
	ref	11.327	5.761	5.019	4.908	4.641	4.486	2.945	1.648	0.779
145.fpppp	test	5.638	4.334	3.726	2.986	2.921	2.823	0.000	------	------
	train	**5.689**	**4.401**	**3.798**	**3.064**	**2.988**	**2.898**	**0.001**	**------**	**------**
	ref	5.631	4.160	3.441	2.652	2.605	2.508	0.003	------	------
146.wave5	test	24.882	21.038	12.873	7.568	1.888	1.057	0.824	0.680	0.610
	train	23.635	19.994	12.247	7.213	1.797	1.001	0.772	0.638	0.575
	ref.10it	**26.548**	**22.492**	**13.769**	**8.138**	**2.004**	**1.155**	**0.922**	**0.773**	**0.673**
	ref	27.820	23.619	14.553	8.763	2.343	1.466	1.195	1.002	0.851

Table 2-10: Cache miss rates for varied SPECFP95 data sets (data stream)

benchmarks showed less than a 2% difference between the **std** and **ref** data sets for instruction distribution, load/store distribution, and segment access distribution. In the following descriptions of individual benchmarks, we note and address only those disparities for which **std** and **ref** differ by more than 2%.

2.3.3.1 SPEC95 integer codes

- 099.go

The go benchmark is a simplified version of a program that plays the game Go. The benchmark plays against itself, and spends much of its execution doing pattern matching, managing data structures, and doing look-ahead computations on the board. For Go, we use the **test** input set as **std**, since all three data sets have approximately the same data set size. The **train** set has the fewest instructions, but has a vastly different profile than does the **ref** data set (about 60% computational instructions as opposed to about 10% for **ref**). The **test** data set has a similar percentage of loads, instruction distributions, and cache miss rates.

- 124.m88ksim

m88ksim is a timing simulator that models the Motorola 88100 microprocessor. Like SimpleScalar, it takes target binaries and simulates them, passing proxy system calls through to the host. Both **test** and **train** have small instruction counts (400M and 100M, respectively), while **ref** has an intractably large instruction count (60G). **test** does little actual simulation other than initializing the simulator, and has a much smaller data set than the other two. **train** performs actual simulation, and has a 4.2 MB data set. Although **ref** has a much larger data set than does **train** (18.9 MB), **train**'s cache miss rates are much higher, due to the inflated effects of compulsory misses (**train** issues about 40 references per byte of its data set, whereas **ref** issues about 3000 references per byte). The instruction distributions between **train** and **ref** differ more than the difference between **std** and **ref** for any other benchmark. **train** issues fewer (15% fewer of all memory operations) to the data segment, but 6% and 8% more memory operations to the heap and stack, respectively. Despite these differences, we use **train** as the **std** set, since **train** does perform a complete simulation of a small benchmark, and the number of instructions in **ref** is too large.

- 126.gcc

gcc is a version of the Free Software Foundation's GNU C compiler version 2.5.3. The benchmark compiles pre-processed C source files into optimized Sparc assembly language files. The **ref** data set is actually a collection of multiple distinct compilations. Since our simulation environment does not currently support multiple distinct initiations from a shell, we chose the largest of the C files in the **ref** data set to use for the simulation. All three data sets have extremely similar profiles and instruction counts. We therefore use **ref** for **std**, since **ref** has the largest data set size and highest cache miss rate of the three.

- 129.compress

compress applies the adaptive Lempel-Ziv compression algorithm to a buffer in memory (the SPEC version implements three statically allocated 14 MB buffers that are used for the input, comparison, and output buffers). The major data structures are a hash table of approximately 400KB, and the memory buffers. The inputs each consist of a number that represents the number of bytes to compress from the memory buffer. The **test** input compresses 1KB, **train** compresses 10KB, and **ref** compresses the full 14MB buffer. Since the number of instructions is roughly linear in the number of bytes from the memory buffer that are compressed, we can effectively choose the simulation length by setting the input. The **ref** input set requires an intractable 43G instructions. We chose **std** to be 400KB, which gives a total data set size of merely a megabyte, but requires a more tractable 1.2G instructions.

- 130.li

li is a Lisp interpreter written in C. We use **train** as the **std** input set, since it has the largest data set (about 200K) of any of the inputs with a tractable number of instructions (111M as opposed to 76G for **ref**). **train** differs from **ref** in the distribution of loads and stores to the memory segments (6% fewer accesses to the heap, and about 3% more stores). Another drawback to using **train** is the fact that only the **ref** input set has significant cache miss rates for caches larger than about 64K. However, as with m88ksim, we are unable to find an intermediate input, and since **ref** is far too long, we use **train** for **std**.

- 132.ijpeg

The ijpeg benchmark reads an image into a memory buffer and processes the image repeatedly with different compression settings. Like gcc, the **ref** data set processes multiple files independently but sequentially. We present the profiles for each of those files (`vigo`, `specmun`, and `penguin`) separately. For **std**, we use the **train** input set, since it has an instruction profile, data set size, and cache miss rate comparable to each of the three input files from the **ref** set, but only produces 1.4G instructions, instead of the 27G-30G produced by the **ref** inputs.

- 134.perl

The perl benchmark interprets code files written in the Perl scripting language. Like gcc and ijpeg, the perl **ref** set contains multiple (two) files: `primes` and `scrabble`. The **test** set is a smaller version of `primes`, and the **train** set uses a file called `jumble`. We use the **train** set (2.4G instructions) for **std**, since the **ref** set executions are prohibitively long (14G and 24G instructions) and **test** is tiny (10M instructions). **train** also has a data set size that is, surprisingly, larger than that of any of the **ref** files (and also generates higher cache miss rates). The execution profile of train is slightly different from either of the two ref data files (3% more heap accesses and 4% more computation instructions). However, the difference between the two **ref** data files is even larger, so the difference is an inevitable consequence of interpreting different scripts.

- 147.vortex

vortex is a object-oriented database benchmark, coded in C, that uses “schema” to map application queries into the database files. The benchmark accesses three different benchmarks through the schema: a mailing list, a parts list, and geometric data. The database distributed with SPEC95 holds about 45MB of data. We use the **ref** input set with one iteration for **std**, since the **ref** data set is significantly larger data set than **train** or **test** (30MB). By running for only one iteration, the data set size is smaller than **ref**, since the amount of data accessed increases with the number of iterations. Unfortunately, the initialization is high with only one iteration, accounting for 34% of the execution time (5.1G instructions per iteration plus 2.6G instructions for initialization). Since the number of instructions per iteration is so high, we pay the price of having to simulate a high fraction of initialization instructions. We view this as justifiable since the instruction profiles in Table 2-3 and the memory operation profiles in Table 2-5 for our **std** input more closely resemble the **ref** set than does the **train** set, which was the alternative candidate for **std** (plus, the differences between **std** and **ref** instruction and access distributions are all less than 4%).

2.3.3.2 SPEC95 floating point codes

All of the floating-point codes were originally written in FORTRAN, and converted to C using AT&T's **f2c** tool. The benchmarks were then compiled with the **peak** optimizations that SPEC defines (which includes -O3), using the version of gcc 2.6.3 retargeted to SimpleScalar assembly.

For the loop-based floating point codes, we can adjust the number of loop iterations in the input files, to reduce the running length of the benchmarks. We can obtain a first-order estimation of the loop-based codes' execution time using the following equation:

$$T = I + El \tag{2-1}$$

T is the running time of the program (number of instructions executed), I is the number of "overhead" instructions (initialization and cleanup/output), E is the number of instructions executed per loop iteration, and l is the number of loop iterations. This is only an approximation, and since I and E depend on the input, the data set must be held constant when adjusting the number of iterations. By measuring T for two values of l, we can solve for E and I. We want to find the minimal l such that I is less than or equal to a certain fraction of the total number of instructions. We adjust l, the number of iterations, such that initialization is no more than 10% of the total execution time. There are a few exceptions where even at 10% the program running time is still too long; in these cases we reduce the number of iterations further so that initialization accounts for no more that 20% of all execution instructions.

- 101.tomcatv

tomcatv is a vectorized mesh generation program that performs finite difference approximation and LU factorization on two two-dimensional arrays. The **test** input does little other than initialization, at 2.7G instructions. **train** uses a smaller data set (7MB instead of 14MB for **test** and **ref**), and runs for 17.6G instructions. **ref** runs for 750 outer loop iterations, for a total of 105G instructions. We ran the **ref** set with 60 iterations, and found that $E = 137.6M$ and $I = 2.12G$ instructions. Since holding initialization to 10% of execution would result in a execution length of over 20G instructions, we set the initialization to be less than 20%, which resulted in 62 loop iterations for the **std** input set (just over 10G instructions). The ref data set uses almost all stack references and little control. The higher fraction of initialization results in the **std** input set issuing 9% of the references to the data segment instead of the stack. Also in std, 3% more of the instructions than **ref** are branches, rather than computation.

- 102.swim

Swim solves a system of shallow water equations (also using finite difference approximations) on a two-dimensional grid. The **ref** data set runs for 900 iterations. **test** and **train** run on the same data set, but for a mere 10 iterations. Solving for E and I, we find that $I = 279.5M$ and $E = 57.0M$ instructions. We set the number of **std** iterations to be 45, at which initialization is under 10%. Even so, the **std** input issues 5% more of the memory accesses to the stack (the **ref** set issues almost no accesses to the stack).

- 103.su2cor

Su2cor is a vectorizable program that computes the masses of elementary particles with a monte carlo method. **test** and **train** use data sets that are about a third and a half of the **ref** data set size, respectively. Our measurements show that, using the **ref** data set, $E = 1.52G$ and $I = 1.77G$ instructions. Given this high number of instructions needed for initialization, limiting initialization to 10% requires too high of an instruction count (18.5G). For the **std** input, we therefore limit initialization to 20%, running the **ref** data set for 5 iterations (9.4G).

- 104.hydro2d

Hydro2d uses double-precision floating point computations for solving the astrophysics problem of computing galactical jets, using hydrodynamic Navier-Stokes equations. All the inputs use the same data set, and simply run for differing numbers of iterations (2, 20, and

200, for **test**, **train**, and **ref**, respectively). The 200 iterations for **ref** require 62G instructions of simulation. Our measurements showed that $E = 367.1M$ and $I = 240.3M$ instructions. We hold initialization to under 10% for **std** by running the **ref** data set for 6 iterations, requiring 2.4G instructions. The residual effects of the initialization cause an extra 5% of memory access to go to the stack (5% in **ref** and 10% in **std**) instead of the data segment.

- 107.mgrid

Mgrid implements a multigrid solver for computing a three-dimensional potential field. The input files specify a grid size, a number of points to solve, and a number of timesteps to calculate solutions for each point. The execution is a two-deep nested loop, with the outer loop incrementing through each spatial point from the input, and the inner loop running through the timesteps for each point. The **test** and **ref** inputs both use a grid that is twice as large in each dimension as the **train** input. **test** computes one point for 40 timesteps, and **ref** computes the effect of 25 points for 40 timesteps each. Since the effects of each point on the grid are independent, we simulate the effects of only one point (*i.e.*, the **test** input set) for the **std** input set. Since our measurements show that, for one point, $E = 109.5$ and $I = 42.3$ instructions, we run for 4 timesteps to keep the initialization under 10%.

- 110.applu

Applu, from the NAS benchmark suite [4], is a solver for five coupled partial differential equations. The code solves a computational fluid dynamics (CFD) problem on a three-dimensional grid. The **ref** data set is larger (29MB) than **test** or **train** (13MB and 3MB, respectively), so we use the **ref** input with a reduced number of iterations for **std**. Our results show that $E = 315.0M$ and $I = 173.0M$ instructions, so we run for 5 iterations to keep initialization under 10%.

- 125.turb3d

Turb3d simulates turbulence in a cube with periodic boundary conditions in all three spatial dimensions. It does so by solving the Navier-Stokes equations using a pseudo-spectral method. All three input files (**test**, **train**, **ref**) use the same data set, but they differ in the number of iterations (**test** and **train** are identical with 11 iterations each, and **ref** runs for 111 iterations). Our results show that $E \gg I$, so we set the **std** input to run only 2 iterations.

- 141.apsi

Apsi is an atmospheric simulator that uses double-precision floating point code to compute the variations of potential temperature, wind components, mesoscale vertical velocity pressure and distribution of pollutants in a three-dimensional environment. The **ref** data set size is 2MB, larger than both that of **test** (1MB) and **train** (512KB). **ref** runs for 960 iterations and 47G instructions, which is prohibitively long. Our measurements show that $E = 29.2M$ and $I = 48.6M$ instructions, so we use the **ref** data set with 6 iterations for **std**.

- 145.fpppp

Fpppp is a quantum chemistry benchmark that simulates an important computational kernel, the two electron integral derivative. The input is a number of atoms, and the execution time is proportional to the fourth power of the number of atoms. The data sets of **test**, **train**, and **ref** are of similar magnitudes (472KB, 468KB, and 564KB, respectively). Since computation time grows so explosively with increases in data set size (number of atoms), we use the **train** input, which has a short (333.1M instructions) running time but has similar execution characteristics to the other inputs. The profiled statistics—including instruction type, load/store ratio, and distribution of memory access to different segments—differ between **train** and **ref** by no more than 1.2%, and generally much less than that.

- 146.wave5

Wave5 solves Maxwell's equations on a two-dimensional mesh with double precision floating point arithmetic. The computation is used to study plasma phenomena. Unlike many of the other SPECFP benchmarks, Wave5 uses heavy indirect addressing. **test** and **train** have the same data set size (27MB). The **ref** data set is much larger, at 40MB. Our measurements show that $E = 1061.2M$ and $I = 2440.5M$ instructions. Given the large number of instructions per loop iteration, we limit the initialization to 20% instead of 10%, and set the **std** input to use the **ref** data set for 10 loop iterations (the **ref** input runs for 40 iterations). The larger fraction of initialization affects the distribution by issuing 3% more of the total memory operations to the stack instead of the data segment.

2.4 Sampling validation

Since sampling may introduce unknown error into the simulation, we validate our sampling methodology against a baseline for a range of sampling parameters. In Table 2-11 and Table 2-12 we present our sampling validation for the SPEC integer and floating point benchmarks, respectively. For each benchmark, we perform two baseline simulations, the performance of which (in IPC) are listed in the third column. The two baseline simulations use the same set of target parameters as the timing experiments described in Chapter 4 (including the Rambus timing model), except that we measure a 4-wide issue superscalar processor here instead of an 8-wide issue machine (with 64KB split level-one caches and a 1MB, 4-way set associative level-two cache). The first baseline for each benchmark, listed in the "cold" row, represents the IPC of the target system. The second baseline, listed in the "perfect" row, represents the IPC of the target CPU core assuming perfect memory and perfect branch prediction.

For each benchmark, we display the IPC of the sampled runs, normalized to their respective baselines. We take samples at intervals of one, ten, and one hundred million instructions committed (listed in the second heading row of each table). For each interval, we perform timing simulation for 1/5, 1/20, and 1/100 of the sample interval (the fractions of timing simulation are listed in the first heading row of each table). There are three modes for each set of sampling parameters: cold, lukewarm, and warm. Cold sampling means that we run in bare-bones functional mode in between timing intervals. Warm sampling means that in the functional (fast) portions in between timing intervals, we send memory references to the cache hierarchy and branch decisions to the branch predictors, keeping both of them updated, eliminating cold start effects at the beginning of each timing interval. In lukewarm sampling, we run in cold mode for most of the non-timing parts of the sampling interval, but then switch to warm mode for a period equal to the length of the timing interval, to warm up the state right before switching into timing mode. The only present cold results for the perfect memory and branch prediction set, since there is no difference between cold and warm mode if the cache or branch predictors aren't used.

Fract. sampled/period			1/5			1/20			1/100		
Benchmark	Method	IPC	1M	10M	100M	1M	10M	100M	1M	10M	100M
099.go	cold	1.321	0.97	0.99	1.00	0.93	0.98	0.99	0.84	0.95	0.96
	lukewarm		0.95	0.98	1.00	0.87	0.97	0.98	0.76	0.91	0.95
	warm		0.93	0.98	1.00	0.80	0.96	0.98	0.56	0.85	0.95
	perfect	2.749	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
124.m88ksim	cold	1.627	1.03	1.07	1.08	1.01	1.06	1.06	0.92	0.97	0.98
	lukewarm		1.02	1.07	1.09	1.01	1.06	1.07	0.95	1.00	1.02
	warm		1.02	1.06	1.09	1.00	1.05	1.07	0.93	0.97	1.00
	perfect	2.748	1.00	1.00	0.96	1.00	1.00	0.96	1.00	1.00	0.96
126.gcc	cold	1.338	0.91	0.97	0.92	0.77	0.90	0.89	0.53	0.75	0.82
	lukewarm		0.88	0.96	0.92	0.72	0.89	0.88	0.47	0.74	0.83
	warm		0.86	0.95	0.92	0.67	0.86	0.88	0.38	0.65	0.80
	perfect	2.619	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	0.99
129.compress	cold	1.347	1.00	0.98	0.98	0.98	0.96	0.94	0.97	1.02	0.93
	lukewarm		0.98	0.99	0.98	0.99	0.97	0.95	1.00	1.05	0.95
	warm		0.99	0.98	0.98	0.96	0.93	0.94	0.98	0.95	0.92
	perfect	2.761	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	0.99
130.li	cold	1.917	1.00	1.00	0.99	0.97	0.98	0.96	0.86	0.89	0.86
	lukewarm		1.00	1.00	1.00	0.98	0.99	0.99	0.92	0.95	0.96
	warm		1.00	1.00	1.00	0.99	0.99	0.99	0.95	0.97	0.97
	perfect	2.650	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.03	1.00
132.ijpeg	cold	2.691	****	0.98	0.99	0.95	0.95	0.96	****	0.89	0.92
	lukewarm		****	0.99	0.99	****	0.97	0.97	****	0.92	0.93
	warm		****	0.98	0.99	0.94	0.95	0.95	****	0.89	0.87
	perfect	2.806	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
134.perl	cold	1.569	1.00	0.98	0.96	0.96	0.88	1.03	0.80	0.83	1.00
	lukewarm		1.00	0.99	0.95	0.94	0.87	1.02	0.82	0.82	1.00
	warm		0.99	0.98	0.95	0.96	0.88	1.02	0.86	0.81	0.99
	perfect	2.594	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
147.vortex	cold	1.639	0.98	1.00	1.00	0.95	0.98	1.00	0.88	0.93	0.97
	lukewarm		0.92	0.97	0.99	0.86	0.94	0.98	0.78	0.87	0.95
	warm		0.88	0.94	0.99	0.73	0.82	0.97	****	0.59	0.85
	perfect	2.453	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2-11: Sampling validation for SPECINT95

Fract. sampled/period			1/5			1/20			1/100		
Benchmark	**Method**	**IPC**	**1M**	**10M**	**100M**	**1M**	**10M**	**100M**	**1M**	**10M**	**100M**
101.tomcatv	cold	1.931	0.92	0.99	0.98	0.84	0.98	1.00	0.70	0.90	0.99
	lukewarm		0.99	1.02	0.98	0.92	1.02	1.01	0.79	0.98	1.00
	warm		1.02	1.02	1.00	0.99	1.00	0.97	0.93	0.99	0.94
	perfect	2.883	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
102.swim	cold	1.772	0.92	0.99	0.95	0.75	0.99	0.96	0.59	0.87	0.97
	lukewarm		1.02	1.01	0.95	0.95	1.04	0.98	0.67	1.02	0.96
	warm		1.00	1.02	0.95	1.00	1.03	0.97	0.96	1.02	0.92
	perfect	2.916	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.01	0.99
103.su2cor	cold	2.068	0.96	0.97	1.00	0.93	0.95	0.98	0.81	0.92	0.98
	lukewarm		0.96	0.99	0.99	0.95	0.96	0.99	0.90	0.92	1.00
	warm		0.97	0.99	0.99	0.96	0.98	0.98	0.93	0.96	0.99
	perfect	2.761	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00
104.hydro2d	cold	1.112	1.03	1.03	0.97	1.01	1.05	1.08	0.91	1.05	1.16
	lukewarm		1.04	1.02	0.97	1.02	1.03	1.07	0.92	1.06	1.15
	warm		0.99	1.02	0.97	0.96	1.02	1.07	0.90	0.99	1.14
	perfect	2.494	1.00	1.01	1.00	1.00	1.01	0.91	1.00	1.01	0.87
107.mgrid	cold	2.037	0.89	0.98	1.02	0.82	1.01	1.02	0.65	0.95	1.08
	lukewarm		0.90	0.99	1.02	0.86	1.13	1.04	0.84	1.03	1.17
	warm		1.00	1.00	1.02	0.99	1.10	1.04	0.95	1.10	1.17
	perfect	2.817	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99
110.applu	cold	1.817	0.98	1.01	1.07	0.90	1.01	1.02	0.67	0.98	1.01
	lukewarm		0.99	1.00	1.07	0.96	1.02	1.03	0.82	1.00	1.01
	warm		0.98	1.01	1.07	0.93	1.02	1.03	0.86	0.96	1.01
	perfect	2.732	1.00	1.00	1.01	1.00	1.00	1.00	0.99	1.00	1.00
125.turb3d	cold	2.294	0.81	0.95	0.99	0.54	0.86	1.00	0.30	0.66	0.90
	lukewarm		0.94	1.01	1.00	0.74	0.98	1.02	0.40	0.76	0.95
	warm		0.98	1.02	0.99	0.94	1.01	1.01	0.86	0.93	0.90
	perfect	2.785	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
141.apsi	cold	1.844	0.95	0.97	1.05	0.83	0.92	0.94	0.58	0.77	0.88
	lukewarm		0.97	0.97	1.05	0.93	0.95	0.93	0.69	0.89	0.92
	warm		0.97	0.95	1.05	0.93	0.94	0.93	0.78	0.87	0.94
	perfect	2.090	1.00	0.97	1.01	0.99	0.97	0.93	0.98	0.96	0.92
145.fpppp	cold	0.539	1.02	0.95	0.86	1.02	0.97	1.01	1.18	0.93	2.21
	lukewarm		1.02	0.95	0.86	1.02	0.97	1.01	1.14	0.94	2.28
	warm		1.02	0.95	0.86	1.01	0.97	1.02	1.07	0.93	2.30
	perfect	2.554	1.00	1.00	1.01	1.00	1.00	1.00	0.99	1.01	0.98
146.wave5	cold	1.968	0.93	0.98	1.00	0.81	0.91	0.92	0.55	0.83	0.91
	lukewarm		0.94	0.99	1.01	0.86	0.97	0.97	0.74	0.87	0.95
	warm		0.97	1.00	1.01	0.90	0.99	0.97	0.84	0.94	0.99
	perfect	2.549	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2-12: Sampling validation for SPECFP95

In both tables, we shade the parameter set for each benchmark that we use for sampling in our experiments. If no table cell is shaded for a particular benchmark, we did not use sampling for that benchmark, as the simulation time with the **std** input set was tractable. For most of the benchmarks, the sampling was inaccurate when the timing period was 1/100 of the sample interval. Some of the benchmarks were sufficiently accurate at 1/20 timing simulation, but most of them required 1/5 timing simulation. All of the simulations needed lukewarm or warm simulation to be maximally accurate. By varying the sampling interval on a per-benchmark bases, we never exceeded a 1% error in IPC for those benchmarks that were sufficiently long-running to require sampling.

Chapter 3

Measuring Cache and Traffic Efficiency

Caches reduce bus traffic by buffering data so that they may service multiple requests with only one transmission of data from the next lower level of the memory hierarchy [51]. However, since cache lines are larger than one word, both cache capacity and bus bandwidth are wasted when spatial locality is poor. Words that will not be referenced are transmitted across the bus, wasting a critical resource for the bandwidth-bound programs defined in the previous chapter. Useless words also reside in the cache, taking up space that could be used to hold needed data. In this chapter, we define and evaluate two metrics: *cache efficiency* and *traffic efficiency*. Cache efficiency measures the fraction of useful data that a cache holds at any given time. Traffic efficiency measures the effectiveness with which bus bandwidth is utilized. By measuring and analyzing these two metrics, we can discover opportunities for improving the effectiveness with which both resources are used.

3.1 Cache efficiency

We define the *cache efficiency* of a given memory to be the average fraction of the cache that holds *live* data [87] over the execution of a program. We define a word in the cache to be *live* if it will be read again before it is overwritten or evicted. A word in the cache is *dead* if its *value* will not be read again before being evicted. If the block is thrown out and subsequently loaded, that space in the cache is dead between its last read and its eviction. A word in the cache is also dead in between a read and a write to that word, since the value is destroyed upon the write and is never reused. Only the period between a write and a read or two reads to a given value is considered to be live. For simplicity, during the following discussion, we will assume a cache that uses one-word blocks.

When a block is first referenced (we will assume by a read), it is loaded into the cache. Once that occurs, there are three possibilities:

- The second reference to the block is a read. In that case, we define the block to be *live* for the period between the two reads.
- The second reference to the block is a write. In that case, we define the block to be *dead* for the period in between the read and the write. Even though the block is referenced again, the data in the block are destroyed (overwritten) with a value produced by the processor; thus the contents of the cache block before the write were not needed.
- The block in question is replaced by a second block that maps to the same location in the cache, before a second reference to the first block occurs. In this case, we define the first block to be dead for the period between the read to the first block and its replacement.

When cache blocks are larger than a single word, the definitions of liveness and efficiency are slightly more complicated. Liveness of a large block can be measured in two ways:

- *coarse grain*, in which we consider the block to be live in between successive reads to that block. This approach is crude, as it lumps operations to separate addresses into one category, and is thus less suitable for evaluating intra-block efficiency.
- *fine grain*, in which we consider a block to be live so long as any word in that block is live (with the definition of liveness for each word being the same as previously defined. This approach is more difficult to measure, since the determination as to whether a block is live cannot be made at the time of each reference to that block. For instance, when a read and then a write are issued to the same cache block, but to different offsets within the block, the block could be live for the period between the read and the write if the write is followed by another read to the same address as the first read (we illustrate this problem with the two reads to address **X1** in Figure 3-1). If the cache block is evicted before another read to a non-overwritten word, then the block should have been dead for the time in between the read and the write. Thus, at the time of the write, future knowledge is required to determine the status of the block.

We depict an example of a cache efficiency calculation in Figure 3-1. In Figure 3-1a, we show an example of how efficiency would be calculated for a one-word block. **X** and **Y** are two cache lines that conflict in the cache. In each box, we show the contents of a cache line after

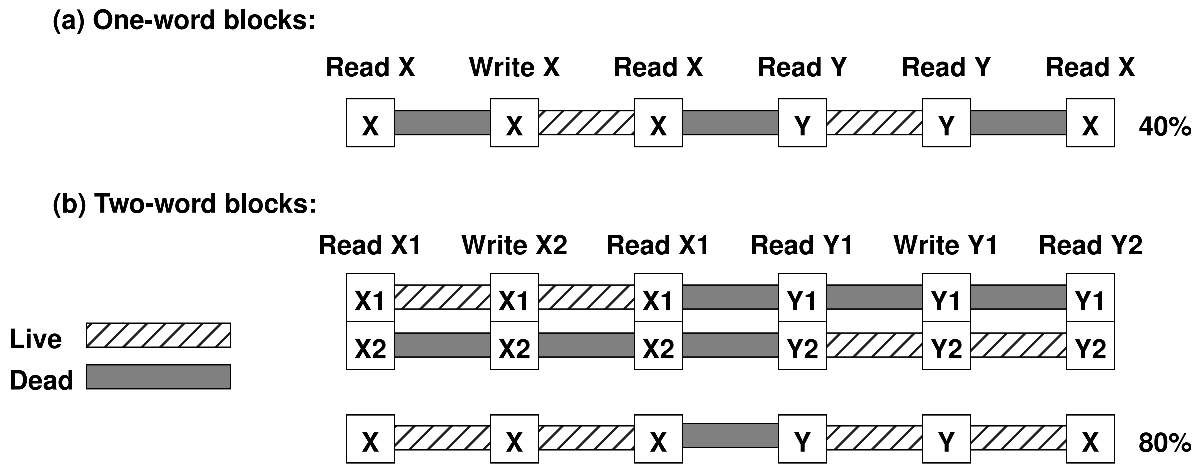


Figure 3-1: Examples of block liveness

the operation above it completes. The bars between the boxes represent live periods for the block (hatched) and dead periods (grey). When **X** is brought into the cache with a read, we mark it as dead between the read and the write, since the read data are subsequently overwritten (and thus did not hold useful data). For the next period, it is live, since it will be read again after the write. The cache line is marked dead, live, and dead over the next three periods, as it is replaced, consumed, and replaced again, in this example. Assuming unit time periods between each operation to this line, the efficiency of this line would be $2/5 = 0.40$.

In Figure 3-1b, we depict an example (measuring efficiency using the fine-grained approach, in which a block is live if any word in the block is live) with two-word cache lines. Each line holds two addresses (cache line **X** holds words **X1** and **X2**, for example). In the upper part of the two-word figure, we show the status (live or dead) for each word. In the lower part of the figure, we show the status for the cache line as a whole (applying a logical OR to the "live bit" of every word in a given cache line). For the first two time periods, word **X2** is never live because it is never read, but **X1** is live for both periods because it is loaded in and then read two operations later. All the words in the block are dead between the last reference to **X** and its replacement by **Y**. There are two methods of measuring efficiency in this case; we can count the entire line as live if any of its constituent words are live (effectively using a logical OR), or we can measure the intra-block efficiency, considering words within a cache line that are dead.

With the former approach, the efficiency of this line for the time period shown (again assuming constant time between operations) would be $4/5 = 0.80$. Using the latter approach, the efficiency would be $4/10 = 0.40$. With the latter approach, the efficiency will always be lower than a cache of equivalent size with one-word blocks, except in pathological cases where the replacement policy punishes finer-grain mapping of blocks into the cache.

3.1.1 Methodology

We measured cache efficiencies in a previous study [12], the results of which we present here. In that study—performed before we had brought up our version of the SimpleScalar tools—we used a modified version of DineroIII [60] (a cache simulator written by Mark Hill) to scan address traces produced by Shade [26] (a tracing tool written by Sun Microsystems).

We measured cache efficiencies for caches with 32-byte blocks and a write-allocate, write-back policy. We simulated all cache sizes that were powers of two between 4KB and 2MB, and with set associativities of 1, 2, and 4. We did not simulate larger caches because most of the benchmarks we used for this study were the SPEC92 benchmarks [116], which had small data sets (all less than 4 MB). We used SPEC92 because SPEC95 was not available at the time.

The SPEC92 benchmarks we used were compress, eqntott, swm256, and su2cor. We ran Compress and Eqntott with the default inputs. We ran su2cor with a short input, and swm256 with the default input for 20 iterations. In addition to SPEC92, we used two other benchmarks: buk and g++. Buk is a NAS [4] kernel that implements bucket sort. g++ is release 2.6.0 of the Gnu C++ compiler, which generates the assembly code of the preprocessed CPU module of a multiprocessor simulator. We produced all Shade traces using Sun Sparcstation 10 workstations, compiled with **`-O3 -mflat`**[1] using GCC version 2.6.0.

1. The "mflat" option compiles code without using the SPARC register windows. Running with register windows would have hidden a fraction of the addresses produced by the benchmark code from our trace, as instructions from traps on window overflows and underflows are not output by Shade. The libraries we used were unavoidably compiled with register windows, and therefore generated some addresses that were not included in our trace.

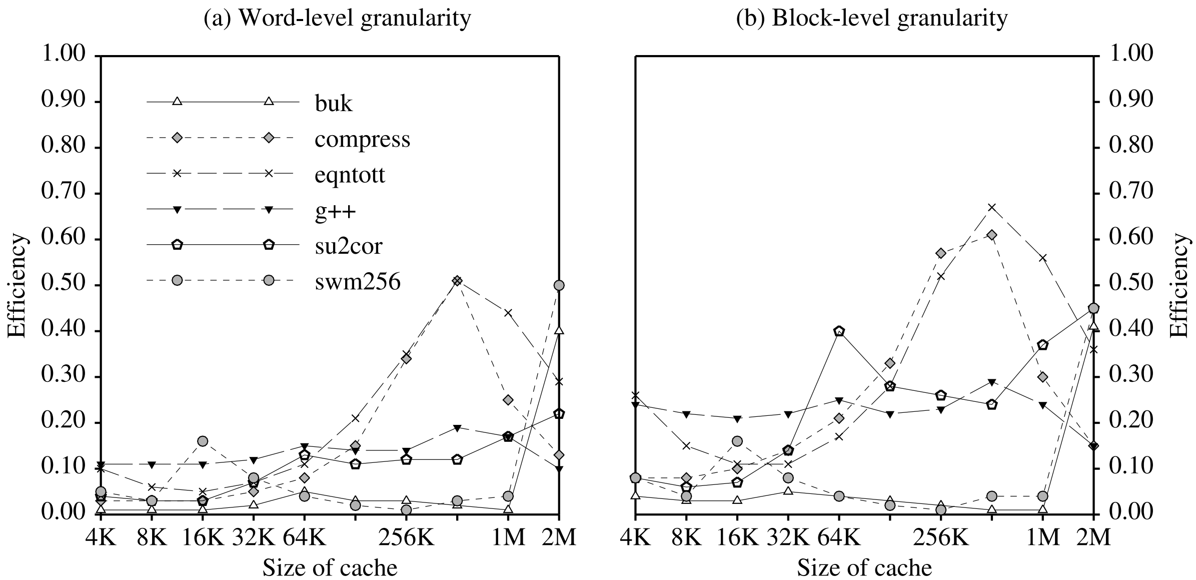


Figure 3-2: Efficiency measurements

To establish that the low efficiencies were caused by poor use of the cache, and not cold-start or dead data (program commencement and termination effects, respectively), we measured the dead time before a frame's first and after a frame's last reference. Those quantities were appropriately negligible, indicating that the programs were sufficiently long-running to prevent endpoint effects from affecting our results.

3.1.2 Measurement of cache efficiencies

In Figure 3-2 we plot the measured efficiency of 4-way set-associative caches. We assume 32-byte blocks for both graphs shown. In Figure 3-2a, we depict efficiencies calculated by differentiating between live and dead words within blocks, and in Figure 3-2b, we show cache efficiencies examining the coarse-grain method of measuring blocks rather than individual words (e.g., the period in between two reads to different words in the same line would be considered live).

Caches substantially smaller than the data set size (and/or the working set size) of the traced application show poor efficiency, as loaded lines are evicted after few uses and the cache thrashes. Efficiency improves with increasing cache size, peaking at the point where the entire data set just fits in the cache. Peaks with a lower value occur when the cache is just sufficiently large to hold the working set; two such peaks are visible for swm with a 16KB cache and su2cor with a 64KB cache. Once the cache is larger than a benchmark's data set, the efficiency decreases inversely proportionally to increased cache size, as the added cache is never touched.

Higher set associativities produce slightly—but not qualitatively—more efficient caches. We present the 4-way set associative results here to show that the efficiencies are generally low even with a high associativity. The direct-mapped efficiencies are even lower.

Although the shapes of the curves match our intuition, we found the height of the curves to be surprisingly low. The word-level granularity efficiencies tend to remain under 20% for cache sizes that are less than a quarter of an application's data set size (for block-level granularity, the efficiencies are under 30%). The ratio between the block- and word-level efficiencies gives a rough idea of what percentage of the words in the block are actually used. Codes that access arrays linearly, with a unit stride, will produce similar efficiencies for the word- and block-level efficiency measurements. The swm benchmark shows this phenomenon. In general, the word-level efficiencies should always be less than block-level efficiencies, since if any word in a block is live, block-level runs count the whole block as live. The one data point where this relation does not hold is Swm with a 2MB cache. The block-level efficiency is lower than the word-level efficiency here because of the method we used in this experiment for calculating block liveness (coarse grained); a store marks everything in the block as dead. In this particular case, blocks that contained multiple live words were declared dead in the block-level calculation, enough that the block-level efficiency was driven under the word-level efficiency.

Although the efficiencies for the larger caches tend to be high compared to those for the smaller caches, these are uninteresting data points because of the close correspondence between the larger cache sizes and the applications' data sets. The two benchmarks with larger data sets (Swm and Buk with 4MB and 6MB, respectively) have efficiencies of under 5% for a *one megabyte* cache. As with the other benchmarks, efficiency rises precipitously when the cache is sufficiently large to hold the working set, which for these two benchmarks is about 2MB. Buk and Swm efficiencies for a 4MB cache are lower than those of a 2MB cache (we did not plot the 4MB results). Even when the cache size is closest to the working-set size, the highest word-level efficiencies were just above 50%, which is a poor *best-case* utilization.

The implications of these low cache efficiencies are twofold: that the cache moves much useless data across the bus (data that are dead on arrival), and that the cache keeps once useful data in the cache longer than necessary. In Chapter 4 and Chapter 5, we propose a number of techniques for addressing both sources of low efficiency. In the remainder of this chapter, we quantify the amount of superfluous traffic moved across the bus, and place a lower bound on bus traffic, thus measuring the highest possible effective bandwidth for a given bus. We note that these efficiencies do not necessarily correlate directly with performance; it may be possible to have a cache with a lower efficiency and still have the system demonstrate superior performance. The low efficiencies that we measured are simply an indicator that it may be possible to improve cache performance; they are evidence that there is potential to make better use of the cache space.

3.2 Traffic efficiency

In this section, we explore three metrics: (1) *traffic ratio*, a well-known metric that measures how much traffic a cache reduces (or increases) from one level of the memory hierarchy to the next, (2) *optimal traffic ratio*, which defines the maximal possible traffic reduction that a cache of a given capacity could perform, and (3) *traffic efficiency*, which quantifies the gap between how much traffic reduction a cache *could* perform and actually *does* perform. With this metric, we are able to quantify how much individual cache features may reduce traffic.

3.2.1 Definition of traffic ratios

In Chapter 1, we discussed how—for a class of programs—stalls caused by insufficient memory bandwidth may become dominant as processors and memory hierarchies attempt to tolerate memory latencies more aggressively. On-chip memory plays a crucial role in reducing off-chip traffic [51]. This reduction increases the effective pin and/or bus bandwidth, as seen by the processor. When bandwidth limits performance, an important metric is the extent to which caches reduce traffic to lower levels of the memory hierarchy, since the traffic reduction increases the effective bandwidth to and from those lower levels.

Therefore, in Section 3.2.3, we measure the *traffic ratio*s of a number of caches, which allows us to calculate effective memory bandwidth for a given processor. Goodman first proposed the concept of a traffic ratio, calling it bus transfer ratio [51]. Hill and Smith proposed the term traffic ratio, which we use herein [61]. We generalize this metric to multiple levels of cache. Let D_i represent the traffic volume—the total amount of transmitted bytes—during the execution of a given program. For a level i in the memory hierarchy, we obtain the data traffic ratio (R_i) by dividing the traffic between levels i and $i + 1$ (D_i) by the traffic between levels $i - 1$ and i (D_{i-1}):

$$R_i = D_i / D_{i-1} \tag{3-1}$$

For example, if a level-one data cache had 1K 4-byte loads issued to it from level 0 (the registers), and the cache produced 32 misses (with 32-byte lines), the traffic ratio would be:

$$R_1 = D_1 / D_0 = (32 \times 32) / (1024 \times 4) = 0.25 \tag{3-2}$$

For simple caches with a write-through policy, we can calculate R_i directly from the cache miss ratio, the number of issued loads and stores, and the cache block size. A write-back cache decouples the direct correlation between miss rate and traffic ratio. Using the miss rate to estimate traffic ratios becomes less accurate for more complicated memory hierarchies: a lockup-free cache may combine two misses with one response from memory, prefetching increases traffic more than it reduces the miss rate, and support for variable transfer sizes makes it difficult to measure cache traffic accurately with miss rate alone.

We use the traffic ratio at each level in the hierarchy to calculate the effective bandwidth to the next lower level of the hierarchy. By dividing the bandwidth from level $i + 1$ of the memory hierarchy by R_i, we obtain the *effective bandwidth* from level $i + 1$. By taking

$$E^{pin} = B^{pin} / \prod_{i=1}^{k} R_i \tag{3-3}$$

where k is the number of levels of on-chip caches, and B^{pin} is the pin bandwidth for the processor in question, we obtain E^{pin}, which is the effective pin bandwidth seen by the processor. Higher traffic ratios for on-chip caches will thus increase the effective pin bandwidth. Note that this metric assumes uniformity in the access patterns. In a real system, working set changes will produce "bursty" periods of traffic, which may be followed by underutilized periods. As memory systems come to look more like queueing systems (as discussed in Chapter 1), and the processor continues to exploit larger instruction windows, the request rate will become more uniform.

3.2.2 Definition of traffic efficiency

While the traffic ratio of a cache shows how effective a cache is at traffic reduction, it gives no indication as to whether the amount of traffic the cache produces close to optimal, or if there is much remaining potential for traffic reduction. The *optimal traffic ratio* of a cache allows us to compute a lower bound on memory traffic, and thus an upper bound on effective memory bandwidth. Assume that D_i^{opt} is the theoretically minimal volume of traffic that may be produced by a memory of a given capacity at level i in the hierarchy. We compute the optimal traffic ratio (R_i^{opt}) as follows:

$$R_i^{opt} = D_i^{opt} / D_{i-1} \tag{3-4}$$

This upper bound is only valid if the processor model remains unchanged; it is possible to change the memory reference stream and therefore further reduce traffic. Also, we note that the traffic volume at a given level i (D_i) is dependent on the organization of the memory hierarchy in the higher levels ($1 \rightarrow i-1$). Measurements for different levels in the hierarchy, whether normally or optimally managed, may not therefore be taken with independent reference streams and then multiplied.

If processor pin bandwidth is the primary bottleneck in a system, we can compute the optimal pin bandwidth (the same could be done for memory bus bandwidth). Let O^{pin} be the upper bound on effective pin bandwidth. Using the optimal traffic ratio, we can compute this upper bound as follows (k and B^{pin} are the same as in Equation 3-6):

$$\mathrm{O}^{pin} = B^{pin} / \prod_{i=1}^{k} R_i^{opt} \tag{3-5}$$

Now that we have an expression for the optimal traffic ratio, we can compute *traffic efficiency*, which we shall denote as E. Traffic efficiency measures how close to optimal the traffic reduction of a given cache is, by expressing the number of times greater the actual cache traffic is than the minimal amount of traffic. Formally, we define E as the ratio of the traffic ratios of a normal cache and a perfectly managed cache of the same size.

The traffic efficiency for level i in the memory hierarchy, E_i, is therefore:

$$E_i = \frac{R_i^{cache}}{R_i^{opt}} = \frac{D_i^{cache}}{D_{i-1}} \times \frac{D_{i-1}}{D_i^{opt}} = \frac{D_i^{cache}}{D_i^{opt}} \geq 1 \tag{3-6}$$

where D_i^{cache} is the traffic generated by the cache at level i, and D_i^{opt} is the minimal volume of traffic that could be generated by a perfectly managed cache at level i.

A level in the memory hierarchy with $E_i = 1$ is therefore perfectly managed, in terms of memory traffic reduction. Large values of E_i indicate a memory organization that generates much more traffic below it than is necessary. Large values of E_i also indicate that there is potential to reduce unnecessary traffic.

In this subsection, we have not discussed how to obtain optimal cache traffic volumes. In Section 3.2.4 we propose a structure that enables us to approximate D^{opt} experimentally, and thus obtain both R^{opt} and E.

3.2.3 Measurement of traffic ratios

We used trace-driven simulation to measure memory traffic for various cache sizes and configurations. We used QPT to generate traces [60]. The traces contained data memory references but no instructions or TLB miss traffic. QPT handles double-word memory accesses by consecutively issuing the two adjacent single-word addresses.

We used the DineroIII cache simulator [60] to perform our cache simulations. We measured cache traffic for the same set of SPEC92 benchmarks and inputs as listed in the **E1** experiment set shown in Table A-1. We list results here for one other SPEC92 benchmark not evaluated in Appendix A: dnasa2, an FFT-based floating-point code, which we ran with a 128x64x64 grid.

We measure the traffic ratio by measuring the total traffic for a given cache with Dinero, and dividing the total traffic by the product of the loads and stores issued and the load/store size. "Total traffic" in these experiments includes write-back traffic but not request traffic (*i.e.*, addresses). We flush the cache upon program completion, writing back all dirty data. We include these flushed write-backs in our traffic measurements.

In Table 3-1, we list traffic ratio measurements for a range of single-level, direct-mapped, 32-byte-block, write-allocate, write-back cache sizes. We saw similar results for caches with higher associativities. Table cells marked "—" are those for which the cache size in question is larger than the benchmark's data set size. This area of the experiment space is uninteresting, since R will always approach 0 when the program runs out of the cache.

When $R_i = 1.0$, a cache generates exactly as much total traffic to memory as there would be with no cache. It is well known [51, 61] that small caches may generate more traffic than a cacheless reference stream. For five of the eight benchmarks, we see this effect in Table 3-1, for caches with sizes of 4KB and less. If a block is replaced quickly after its first use—or if there is little spatial locality associated with the access that caused the miss—the other six or seven words loaded with the 32-byte block are superfluous, and will contribute to a higher traffic ratio.

We see the effect of caches increasing total traffic for some larger cache sizes as well. compress and su2cor generate more traffic with a 64KB cache than would a cacheless system.

Trace	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
Compress	1.76	1.59	1.46	1.29	1.10	0.82	0.43	—	—	—
Dnasa2	1.34	0.94	0.73	0.62	0.29	0.05	—	—	—	—
Eqntott	0.55	0.47	0.43	0.39	0.34	0.27	0.18	0.11	0.06	—
Espresso	0.39	0.20	0.08	0.01	—	—	—	—	—	—
Su2cor	6.88	6.11	4.75	2.99	1.43	0.82	0.61	0.29	0.13	—
Swm	3.94	1.79	0.63	0.60	0.59	0.58	0.58	0.56	—	—

Table 3-1: Traffic ratios for 32-byte block, direct-mapped caches

compress repeatedly accesses a large hash table, so its memory reference stream contains little spatial locality (a larger block size will consequently waste bandwidth). Su2cor iterates over several large arrays, some of which conflict heavily in its main loop for cache sizes less than 64KB. In contrast to su2cor, swm has roughly the same traffic ratio from 16KB to 1MB cache sizes. Swm iterates over large arrays, with a reference pattern that contains little locality and no small working sets [99]. Swm does have high spatial locality, however, allowing one each cache miss to service multiple loads, thus keeping the traffic ratio under 1.0 for all but the smallest cache sizes. In general, R_i ranges between 0.1 and 1.0 for caches that are not overly large or small for a given program.

The generation of machines that these benchmarks were designed to test did not have on-chip caches larger than 64KB. We therefore calculated the arithmetic mean of the R_i for all caches with sizes greater than or equal to 64KB and less than the data set size of each benchmark. The mean across all benchmarks was 0.51. While this estimate cannot be applied to an individual program/cache combination, we can say that for these benchmarks running on systems with cache sizes typical of the benchmarks' generation, the processor traffic is reduced about in half. Since the SPEC92 benchmarks' data sets are not large, however, these results are conservative—many of these programs run out of the caches, whereas larger benchmarks would incur more conflict misses, thus increasing the traffic ratio.

3.2.4 Methodology for measuring traffic efficiency

In this section, we measure an upper bound on effective memory bandwidth. By experimentally measuring a value that approaches D^{opt}, we can calculate traffic efficiency using Equation 3-6, and thus obtain the highest effective memory bandwidth for a given bus.

We approximate D^{opt} by simulating a special cache organization that comes close to minimizing the traffic it generates from misses and write-backs. We call this structure a *minimal-traffic cache*, and will henceforth refer to it as an MTC. An MTC differs from a traditional cache in four respects:

- **Block size**: both the transfer size and the block size are equal to the request size. Thus, only data that are needed by the processor are loaded across the bus or stored in the MTC.
- **Associativity**: the MTC is fully associative. No conflicts can therefore ever evict a needed block, which would cause it to be reloaded and increase traffic.
- **Replacement policy**: the ideal replacement policy would choose to evict the block that will cause the least total memory traffic. Belady's **min** policy [5]—which uses oracular knowledge, and thus can never be implemented in a real machine—replaces the block that the processor will reference the farthest in the future (or any dead block in the cache, which either will never be referenced again or will be overwritten). The **min** policy is an approximation of the optimal policy; we shall discuss the difference between **min** and optimal subsequently. If the next reference to the loaded block is of lower priority (*i.e.*, will be read farther in the future) than any block in the cache, the block bypasses the cache rather than evicting something of higher priority.
- **Write policy**: the traffic-optimal write policy is *write-back*, *write-validate* [70]. A write-back policy will always produce less memory traffic than write-through for caches that have one-word blocks[1]. A write-validate policy overwrites the contents of a block and the block's associated tag, rather than fetching the block from memory and then overwriting the word, as in a *write-allocate* policy. For blocks with multiple words per block, a write-validate policy requires valid bits for individual words, and if a read accesses part of a write-validated block that is not valid (*i.e.*, has not received a store to that particular word), a read miss occurs. Read misses to write-validated blocks do not occur in an MTC, because the blocks contain one word, and thus no part of the block is invalid whenever it is created in the cache by a write. We also incorporate *write bypass* into the MTC, in which a write that has a lower priority than anything else in the cache—according to the **min**

1. For larger cache lines, write-back will have less traffic only when the number of writes to a line (while it is in the cache) is greater than the number of words in the cache line.

replacement policy—is not loaded into the cache, but instead is sent directly to the next level of the memory hierarchy.

The MTC we measured does not place an optimal lower bound on memory traffic for two reasons. First, the **min** policy is sub-optimal for write-back caches, since in our application of **min** there is an additional cost (extra traffic) associated with replacing a dirty block. Horwitz *et al.* proposed an algorithm to manage optimal replacement in the presence of write-backs [63]. We implemented only the **min** algorithm, and not the optimal write-conscious Horwitz algorithm. We believe that the disparity between the two is small for large caches, and therefore not worth the large additional complexity of simulating the Horwitz algorithm.

Second, after we published this study [13], we discovered that, under the constraints imposed by the MTC, Belady's **min** algorithm is sub-optimal for read traffic. For write-validate caches with one-word blocks, we can generate less read traffic by modifying Belady's algorithm. When prioritizing blocks for replacement, we must ignore both all future writes and all reads that follow any future writes and are to the same address as the writes. Since writes effectively create a block (with one-word blocks), the block can be considered to be dead beforehand, as described in Section 3.1. Since the block is dead before the write, it is a good candidate for replacement, and when the write to that block occurs, it can be written anywhere in the cache (overwriting another block that may have recently died). We can reduce misses over Belady's algorithm essentially because, with one-word blocks and a write-validate policy, we are treating blocks as *values* and not *addresses*. If the caching paradigm was thrown away, and only individual values were considered, Belady's scheduling algorithm would still be optimal.

We show an example of this extension in Figure 3-3. The figure shows the effects of the two different priority schemes on an MTC with a fixed reference stream. The reference stream (time) goes from left to right. The initial contents of the MTC are addresses **X**, **Y**, and **Z**. In the original priority scheme, when **W** is loaded (1), **Z** is the cache block that will be referenced farthest in the future, so **Z** is replaced by **W**. The subsequent three accesses (2,3,4) all hit (and **X** dies on reference (2); we assume that it won't be referenced again before being evicted). Finally, **Z** is loaded, and replaces **X**. In the original prioritization, we load a total of two cache

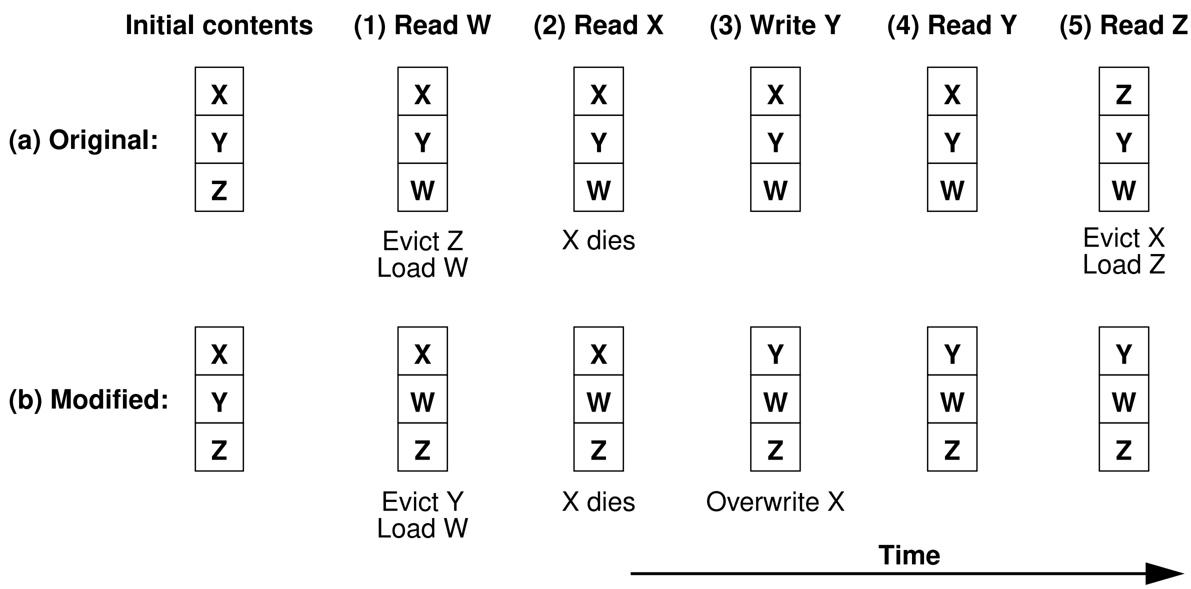


Figure 3-3: Extending Belady's **min** algorithm

blocks from memory. In the modified priority scheme, we load **W** (1), and consider **Y** to be dead, since it will be overwritten before the next read to **Y**. We therefore replace **Y** with **W**. We then read **X** (2), which subsequently dies. When we write **Y** (3), we overwrite **X**, since it is dead. The following reads (4,5) to **Y** and **Z** are hits. With the modified priority scheme, we only loaded one cache block from memory, but loaded two with **min**. Thus, **min** is non-optimal for read traffic with caches that have a write-validate policy and one-word blocks (**min** may be non-optimal for write-validate caches with larger blocks as well, but that study is beyond the scope of this dissertation).

For the above two reasons, are our measurements of D^{opt} are not an optimal bound, but an approximation. We show in the following section, however, that D^{opt} is still substantially smaller than D^{cache} in most cases. Finally, we note that we do not consider tag overhead in the MTC. Since tag overhead increases when smaller blocks are used (because there are more blocks and the tags are slightly larger), the *gross cache size* [61] of an MTC with one-word blocks and a traditional cache with larger blocks will be different. We equate the data capacities, not the gross cache sizes, in this study.

3.2.5 Measuring traffic efficiency

We used QPT-generated traces, coupled with the Dinero cache simulator, to measure D_1^{cache} for the numerator of E_1, the traffic efficiency expression shown in Equation 3-6. We approximate the denominator of E_1 (D^{opt}) by measuring D^{MTC}—for which the definition of MTC is as described in Section 3.2.4—with our own two-pass simulator. Since replacement decisions in an MTC require future knowledge, our simulator scanned each trace once to construct a live range graph (necessary for prioritizing blocks in the cache for replacement), and once to perform the actual cache simulation. Since maintaining live range information for each reference in the compressed QPT trace would have required prohibitively large disk space (at the time we did this study), we broke the program down into *epochs* (constant segments of time). In the first pass, we saved to disk only those live ranges that crossed the epoch boundary. During the second pass, we scanned ahead in the trace and constructed all live ranges within each epoch, using the file on disk to fill in those inter-epoch live ranges. By increasing epoch size, we could increase time (scanning each epoch during MTC simulation) at the expense of space (the inter-epoch disk file).

We used the same benchmarks and inputs as described in Section 3.2.3. The traffic measurements for both simulators also include the same components (*e.g.*, write-back traffic) as did the traffic ratio experiments. For the D^{cache} measurements, we assumed direct-mapped, 32-byte block, write-allocate, write-back caches.

In Table 3-2, we list traffic efficiencies for caches ranging from 4KB to 2MB. Our results show that there is a wide disparity of values for E across the different benchmarks. Four of the benchmarks have E between 20 and 100 (compress, eqntott, espresso, and su2cor)—even for large caches. The other two—dnasa2 and swm—typically have values of E between 2 and 10. These two benchmarks are scientific codes that display little temporal locality, thus the reference stream contains less opportunity for optimization by a better-managed cache. The large jump to a E of 124 for swm with a 1MB cache occurs because the MTC (being fully associative) is able to eliminate the conflicts of the major data arrays, which are significant in swm for

Trace	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
compress	18.7	19.5	21.9	25.5	29.2	30.7	32.5	—	—	—
dnasa2	6.2	4.7	4.1	4.6	7.0	10.0	—	—	—	—
eqntott	34.5	35.8	49.7	94.4	100.5	94.1	72.7	47.7	28.6	—
espresso	26.3	40.4	82.2	28.9	—	—	—	—	—	—
su2cor	15.1	16.4	17.2	21.9	20.1	25.7	40.3	28.7	35.8	—
swm	17.2	7.9	2.8	2.7	2.8	3.0	3.5	5.4	124.1	74.8

Table 3-2: Traffic efficiencies for 32-byte block, direct-mapped caches

traditional caches. Only when a traditional cache is sufficiently large (4MB or greater) are these conflicts in Swm ameliorated.

3.2.6 Factorization of traffic efficiency

These high traffic efficiencies demonstrate that there is a significant opportunity to increase effective pin bandwidth—between one and two orders of magnitude—by making better use of the on-chip memory. We now turn to determining which factors contribute to these large gaps. In Figure 3-4, we show a log-log plot of traffic volumes (in KB) versus cache sizes, for three of the SPEC92 benchmarks. We include only compress, eqntott, and swm, since they are representative of the other benchmarks. The top six lines in each graph represent 4-way, set-associative caches with block sizes from 4B to 128B. The thick dotted line represents a fully associative, write-allocate, write-back cache that uses **min** as its replacement policy. The thick solid line represents the MTC that we used for all traffic efficiency calculations. Large gaps between a line and the MTC line indicate large traffic efficiencies.

There are three factors visible on Figure 3-4 that contribute to large gaps between cache and MTC traffic. The first is increased block size. Compress has little spatial locality, since many of its accesses are to a hash table. Any increase in block size causes a corresponding increase in traffic. The same effect is visible for Eqntott (to a lesser extent), and for Swm when the cache sizes are smaller than 32KB. Swm shows spatial locality for larger caches (between 32KB and 2MB) because the extra words in larger blocks are used when the block is not quickly replaced—when the working set fits into the cache. The second factor contributing substantially to the cache/MTC traffic gap is the combination of associativity and oracular

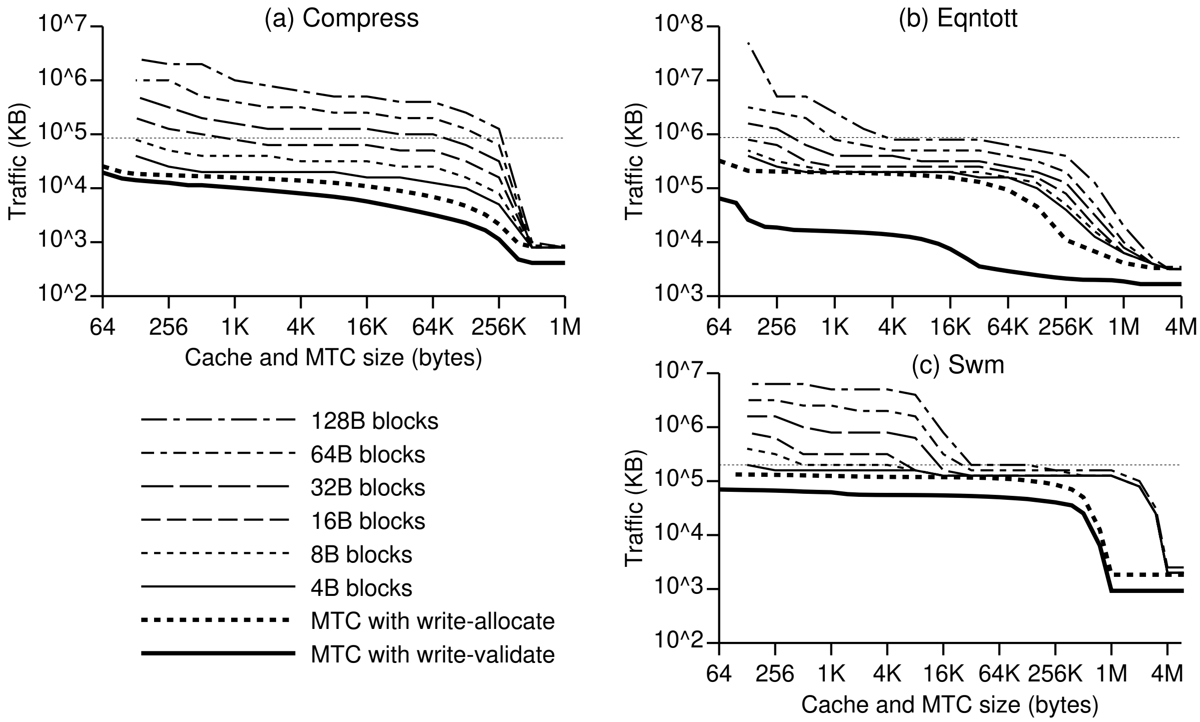


Figure 3-4: Total traffic generated by different cache and MTC sizes

replacement, which causes the large gap for Swm at 1MB. The third factor is the write-validate policy, which causes the majority of the gap for Eqntott.

To better understand which of these factors are significant, we isolate each factor experimentally. Traditional caches and an MTC differ by four factors. To dissect E into its components, we begin with a cache and add one MTC-like factor at a time. We can measure the addition of each factor by simulating and comparing structures that have all but one factor in common. We list the factors isolated with the pairs of structures in Table 3-3. In the first column, we list the factors that we isolated. In the second column, we list the common features of the experiment pairs for each isolation. The third and fourth columns list the two organizations used for each isolated factor. We did not isolate cache bypassing as a factor, since it was implicit in the **min** replacement policy.

The traffic reduction for each factor depends on the underlying structure—for instance, **min** replacement will reduce traffic differently for a write-validate, fully associative, 4-byte block cache than it will for a write-allocate, 32-byte block, direct-mapped cache. We can add the four MTC mechanisms successively in a number of orders. There are two restrictions on the

Factor	Common	Exp1	Exp2
I. Associativity	LRU, 32B, write-allocate	direct mapped	fully associative
II. Replacement	fully assoc., 32B, write-allocate	LRU replacement	min replacement
IIIa. Blk. size (MTC)	min, fully assoc., write-allocate	32B blocks	4B blocks
IIIb. Blk. size (cache)	LRU, fully assoc., write-allocate	32B blocks	4B blocks
IV. Write-validate	**min**, fully assoc., 4B	write-allocate	write-validate

Table 3-3: Experimental parameters for Table 3-4

order; full associativity should be measured before **min** replacement (replacement policy is irrelevant for a direct-mapped cache), and the small blocks should be measured before the write-validate policy. With these restrictions, there are still six possible orders in which the mechanisms may be measured. We show only one such order in Table 3-3. We performed two separate experiments for isolating the effect of block size—one with **min** (Factor IIIa) and one with LRU replacement (Factor IIIb). The block size experiment with LRU replacement is shaded because it is not part of the successive addition of factors that the other four experiments are.

In Table 3-4, we quantify how toggling each factor affects *E* for each benchmark. The values in the table show the change in traffic efficiency as each factor is toggled. We simulated one cache size per benchmark, using 64KB data caches for all benchmarks except espresso, for which we simulated a cache size of 16KB (because of its small data set). In these experiments, we do not include request traffic, which increases with smaller block sizes, and thus our traffic results are biased in favor of smaller blocks.

Below the four rows for individual mechanisms, we compare the sum of the contributions of the individual factors to the traffic efficiency. The rows containing the sum and *E* should be equal. We are not yet able to adequately explain the gap between the two rows (which ranges from 0.2 to 1.0).

In Table 3-5, we show the relative fraction of traffic efficiency that each mechanism contributes to the total. Of these mechanisms, reduced block size is, unsurprisingly, the largest contributor to *E*, constituting the largest component of two of the benchmarks (compress and eqntott, at 0.48 and 0.37) and the second largest for another two (su2cor and swm, at 0.25 and 0.11). The other three mechanisms are the largest component of *E* for at least one benchmark

Benchmark	compress	dnasa7	eqntott	espresso	su2cor	swm
Cache size	64KB	64KB	64KB	16KB	64KB	64KB
I. Associativity	1.8	-3.8	0.5	73.0	8.4	0.1
II. Replacement	12.0	8.4	31.0	3.9	4.6	0.3
IIIa. Block size (MTC)	14.0	0.4	37.0	3.5	5.0	0.3
IV. Write-validate	1.2	1.2	31.0	1.0	1.2	1.3
Sum (I+II+IV+V)	29.0	6.2	99.5	81.4	19.2	2.0
Traffic efficiency	29.2	7.0	100.5	82.2	20.1	2.8
IIIb. Block size (cache)	25.0	2.7	47.0	68.0	14.0	0.3

Table 3-4: Efficiency gap for different optimizations

Benchmark	compress	dnasa7	eqntott	espresso	su2cor	swm
Cache size	64KB	64KB	64KB	16KB	64KB	64KB
I. Associativity	0.062	-0.543	0.005	0.888	0.418	0.036
II. Replacement	0.411	1.200	0.308	0.047	0.229	0.107
IIIa. Block size (MTC)	0.479	0.057	0.368	0.043	0.249	0.107
Write-validate	0.041	0.171	0.308	0.012	0.060	0.464
IIIb. Block size (cache)	0.856	0.386	0.468	0.827	0.697	0.107

Table 3-5: Fraction of traffic efficiency per factor

each: associativity for espresso and su2cor (0.89 and 0.42), write-validate for swm (0.46), and **min** replacement for dnasa7 (1.20). What is surprising about these results is the lack of one factor (or even two) that dominates in traffic reduction. This result indicates that—to reduce traffic substantially—caches must incorporate a range of mechanisms to be effective across different benchmarks.

One aberration in Table 3-5 stands out: the negative value for dnasa7. The sign change is caused by an *increase* in traffic when a fully associative cache is compared to a direct-mapped one. The increase in traffic is caused by a antagonistic interaction between the reference stream and the LRU replacement policy; a well-known case in which LRU replacement works poorly for sequentially accessed data [48]. In this case, less mapping flexibility (a direct-mapped cache) prevents the replacement policy from evicting some useful blocks, causing the direct-mapped cache to produce less traffic. (When the **min** replacement policy is added, it eliminates that problem and reduces traffic much further, which is why the replacement policy component of traffic reduction is larger than the final traffic efficiency for dnasa7).

In the last row of Table 3-5, we show the relative effect of reducing block size in a fully associative LRU replacement cache, instead of a **min** replacement cache. The relative contributions of reduced block size are much larger for a cache with an LRU replacement policy, ranging from a small 0.11 (Swm), to over a third (0.39 and 0.47 for dnasa7 and eqntott), to well over half (0.86, 0.83, and 0.70, for compress, espresso, and su2cor). Since LRU replacement is less efficient at packing data into the cache than **min**, increasing the number of blocks under LRU produces a large reduction in traffic.

We have shown in this chapter that a large gap—as much as two orders of magnitude—exists between the amount of traffic that a cache generates and an approximation of the optimal. Furthermore, each of the design aspects (block size, associativity, direct/indirect accessing, etc.) in the near-optimal structure can contribute significantly to traffic reduction. In the next three chapters, we discuss how each of these mechanisms can be implemented and/or approximated in a cost-effective manner, reducing memory traffic and thus improving performance for bandwidth-bound codes.

Chapter 4

Reducing the Impact of Memory Traffic

In Chapter 1, we discussed how and why memory traffic can cause significant degradations in processor performance. In Chapter 3, we showed that memory traffic was significantly heavier than a theoretical lower bound. However, the bound that we derived for minimal memory traffic is not reachable in practice. In this chapter, we explore several implementable techniques that ideally lessen both the amount and the performance impact of memory traffic.

The minimal traffic cache differs from traditional caches in four respects: block size (what data are fetched upon a miss), associativity (how data are mapped into the cache), replacement policy (what is thrown out of the cache), and write policy (how created values are handled). In this chapter, we explore techniques that address the first factor: what data are fetched. In Chapter 5, we address how data are mapped on-chip, and what data should be fetched upon a demand miss. In Chapter 6, we propose the DataScalar architecture, which (among other benefits) eliminates all inter-processor write traffic.

Because of long memory latencies and limited off-chip memory bandwidth, microprocessor designers have been placing successively larger caches on the processor dies with each new generation. The Dec Alpha 21364 [56], for example, will use essentially the same processor core as the 21264 [55], but with a faster clock, and significantly more aggressive on-chip and off-chip memory systems (including a large on-chip cache greater than one megabyte). In this chapter, we explore three policies that we designed to improve the memory system performance of large on-chip caches. The policies use information dynamically saved with the cache tag to track the long-term behavior of a block, attempting to improve how the block is managed each time it is fetched. The three policies are: *dual-size fetching*, in which the level-two cache issues a large (block) or small (subblock) request as needed, *subblock prefetching*, in which the L2 cache tries to bring in only the portions of a large block that will be needed, and *bus prioritization*, in which data that are to be speculatively loaded are brought from memory only when the interconnect is idle. At the end of this chapter, we evaluate the performance of all those policies together. In the following subsection, however, we simply measure what the parameters of large, traditionally managed L2 caches should be.

4.1 What to fetch

When designing a system, the architect must decide how much data the cache should fetch upon each miss, in other words, how large the cache block should be. The minimal-traffic cache used one-word blocks to prevent unnecessary data from ever being loaded. In practice, one-word blocks would result in dreadful performance, as all applications exhibit some spatial locality. Furthermore, increased address traffic would offset the reductions in traffic from smaller blocks. Fetching larger blocks, conversely, reduces the number of misses, improving performance (unless the block is so large that it pollutes the cache enough to result in a net increase in misses). However, the larger blocks also load more unnecessary traffic. Since both cache misses and superfluous traffic can hurt performance, there is an inherent tension between trading reduced misses for increased traffic and vice-versa.

Small on-chip caches have typically had block sizes in the range of 16 to 64 bytes. Since these caches were small, they had few blocks; thus blocks much larger than 64 bytes would have caused excessive pollution and a higher miss ratio. In these caches, the small blocks made efficient use of memory bandwidth while keeping the miss ratio low. We show in this section that having both low traffic and ideal miss ratios is difficult for large (a megabyte or more) caches.

To illustrate, we define two operating points for a cache's block size: the *performance point* and the *pollution point*. The performance point is the block size at which overall system performance is highest. Blocks larger than the performance point will cause reduced performance because of bus contention, whereas blocks smaller than the performance point will cause reduced performance because of more numerous misses. The pollution point represents the block size at which the miss ratio, and not absolute performance, is minimized. Blocks larger than the pollution point will cause more misses due to cache pollution, whereas blocks smaller than the pollution point cause more misses because they are not exploiting spatial locality as well.

Since cache pollution becomes less of a problem for larger caches (since there are more blocks of a given size), the pollution point will tend toward larger blocks. For multi-megabyte caches, the pollution point may well be at block sizes significantly larger than the performance point. For the rest of this chapter, we perform experiments assuming a large, on-processor L2 cache, with the processor technology targeted approximately five years hence (circa 2003, in line with the Intel and SIA projections [102, 136]). We assume a target system as described in Chapter 2, with the following parameters: the processor core we simulated was a 2GHz, dynamically scheduled, 8-way issue superscalar core. We assumed a 256-entry RUU, with a corresponding 128-entry load-store queue. We assumed that the core contained six integer ALUs, three integer multipliers, six FP ALUs, two FP multiply/dividers, and six ports to memory. The branch misprediction penalty was three cycles, and we assumed a huge (128K entry) gshare branch predictor, in an attempt to gain the accuracy that branch predictors will doubtless have five years hence. For the memory system, we simulated 64KB, 32-byte block, 2-way set associative split instruction and data caches (similar to the recently announced Compaq Alpha 21364), which were virtually indexed and physically tagged, and accessible in a single cycle. We simulated split 8KB instruction and data TLBs, each two-way set associative. We assumed a 256-bit cache bus, clocked at the core speed, with a single cycle required for arbitration/turnaround. We simulated a 1MB physically indexed, physically tagged L2 cache, assuming a 10-cycle hit penalty and a write-allocate, write-back policy. For the physical memory, we simulated a detailed Direct Rambus channel [30] and subsystem to service L2 misses off-chip. We assumed four simply interleaved RDRAM channels, each clocked at 500MHz, with two bytes per channel transmitted on both the rising and falling edges of the clock. We simulated all resources in the RDRAM chips, including precharge penalties and page hits on open senseamps, bank conflicts, and access pipelining.

In Table 4-1, we show the pollution and performance points for a number of the SPEC95 benchmarks running on the described target system. We omitted several of the benchmarks (m88ksim, li, ijpeg, fpppp) because their working sets fit nearly completely in the L2 cache, making optimizations to reduce the impact of L2 misses useless. In the table, we vary the L2 cache block size across the columns, from 64 bytes to 4 Kilobytes. In each pair of rows, we show both the IPC (performance) and the L2 miss ratios for a particular benchmark. The shaded number in each IPC row indicates the block size with the highest performance (the performance point), and the shaded number in each miss ratio row indicates the block size with the lowest miss ratio (the pollution point). The results in this table have three notable implications:

- The best mean performance (average performance point) is at 256-byte blocks, and the lowest mean miss ratio (average pollution point) is at 4KB bytes. The performance points are thus significantly larger than block sizes for caches to date; only three of the thirteen benchmarks have performance points under 256-byte blocks.
- The performance points are highly application-dependent; they range from 64 bytes all the way to 4 KB. Selecting a block size at either extreme will lead to poor performance from a subset of the applications. Selecting a block size in the middle (e.g., 256 bytes) will also lead to degraded performance for a number of the applications.
- For almost half of the benchmarks, there is a significant gap between the performance and pollution points (ranging from factors of 4 to 32). This gap presents an opportunity: if the cache could fetch only those portions of the large blocks that are needed, the miss ratio could be reduced (since pollution is not an issue for these codes) without a corresponding reduction in performance due to bus contention.

These implications lead us to three requirements for large on-chip caches. (1) The block size should be larger than that of traditional caches, (2) the caches should be managed in such a way as to provide good performance across the entire range of applications, and (3) the cache could use intelligent fetching to improve performance beyond that of the performance point.

We measured the pollution and performance points for smaller caches (512KB) and fully associative caches with random replacement (both 512KB and 1MB). There were some slight

		Block size							
Benchmark	**Metric**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
126.gcc	IPC	1.498	1.522	1.536	**1.538**	1.521	1.468	1.328	1.010
	Miss ratio	3.600	2.640	1.880	1.320	1.020	**0.910**	0.940	1.210
129.compress	IPC	**1.264**	1.208	1.112	0.937	0.655	0.389	0.206	0.101
	Miss ratio	9.870	9.450	9.010	**8.400**	8.810	9.580	10.990	12.680
134.perl	IPC	1.711	**1.784**	1.782	1.718	1.578	1.343	1.029	0.635
	Miss ratio	5.770	3.910	2.930	2.450	2.250	2.190	**2.190**	2.490
147.vortex	IPC	2.078	**2.086**	2.051	1.938	1.703	1.185	0.631	0.238
	Miss ratio	6.670	5.390	**4.850**	4.970	5.390	7.020	9.390	13.010
101.tomcatv	IPC	1.492	2.048	2.451	2.694	2.833	2.908	**2.953**	2.929
	Miss ratio	33.970	17.070	8.590	4.310	2.170	1.100	0.570	**0.380**
102.swim	IPC	1.172	1.734	2.221	2.518	**2.554**	2.345	2.268	1.899
	Miss ratio	31.450	16.010	8.260	4.430	2.670	1.820	1.260	**1.140**
103.su2cor	IPC	1.853	2.395	2.707	2.882	2.979	**3.009**	2.990	2.780
	Miss ratio	13.860	7.030	3.720	1.940	1.010	0.530	0.290	**0.220**
104.hydro2d	IPC	0.568	0.898	1.236	1.528	1.702	**1.745**	1.736	1.296
	Miss ratio	50.320	33.290	19.340	10.030	5.110	2.670	1.410	**1.180**
107.mgrid	IPC	1.673	2.313	2.728	2.570	2.716	2.840	**2.901**	2.835
	Miss ratio	28.070	17.500	10.770	7.200	5.060	3.720	2.570	**1.480**
110.applu	IPC	1.229	1.847	2.327	2.586	2.716	**2.787**	2.333	1.439
	Miss ratio	43.240	30.380	18.190	9.570	4.950	2.730	1.920	**1.710**
125.turb3d	IPC	2.441	2.922	**3.239**	3.015	2.843	2.838	2.685	1.467
	Miss ratio	40.960	24.500	14.640	12.810	9.630	7.260	**5.680**	7.380
141.apsi	IPC	2.244	2.590	2.645	2.786	2.763	**2.859**	2.849	0.949
	Miss ratio	8.360	6.260	4.580	2.150	1.280	0.550	**0.310**	3.370
146.wave5	IPC	1.919	2.349	2.643	2.712	**2.821**	2.671	1.236	0.455
	Miss ratio	8.760	5.540	4.000	1.900	**1.240**	1.510	2.880	3.790

Table 4-1: Performance versus pollution points, 1MB 4-way set associative L2 cache

changes in the pollution and performance points when running with the alternative parameters. For the 512KB cache, the random replacement caused pollution to occur with a lower block size than it did with the 1MB cache, slightly decreasing the average gap between the performance point and pollution point for six of the benchmarks. The exception to that trend was applu, which had an identical pollution point but experienced a higher miss ratio for the fully associative cache, thus lowering the performance point. Increasing the cache capacity from 512KB to 1MB tended to increase both the pollution and performance points, which caused the gap to shrink slightly in four cases (gcc, vortex, applu, and wave5) and grow in two others (compress, turb3d). All in all, high associativity and halving the cache size did not qualitatively change the relationship between the performance and pollution points. The relative stability in the average size of the performance/pollution point gap indicates that the cache requirements listed above are applicable to a range of large on-chip caches.

4.2 Dual-size fetching

The first policy we propose is *dual-size fetches*, in which the cache dynamically decides whether to fetch a large block (spatial locality is high, so the consumed bus traffic will not be wasted) or a smaller block (spatial locality is low). Supporting multiple block sizes can make for complex and difficult hardware design (particularly when addressing fragmentation and packing issues). Here we describe two hardware-elegant methods of implementing this policy. The first is to map small blocks into the cache, and fetch a number of blocks when spatial locality was high (as proposed by Johnson and Hwu [69]). We evaluate the second, which is to implement a subblocked cache in which either a subblock or a block may be fetched upon a miss. We set the block to the block size of the pollution point (the data is mapped into the cache at a granularity that minimizes the miss ratio, on average 4KB) and we set the subblock size to the block size of the performance point (data is transferred at a granularity that maximizes performance, on average 256B).

The dual-size fetch policy (DSF) maintains state describing the characteristics of a block after the block has been evicted from the cache. Since we are using blocks equivalent to the page size in our target system, this state can be stored for fast access, and maintained as an extension to the TLB entries until the TLB entry is evicted. When an entry is evicted from the TLB, the system may store the per-block information in a special region of physical memory or, in theory, as a part of the page table itself. The former solution limits overhead to be proportional to the size of physical memory, whereas the latter would be proportional to the size of touched virtual memory (but is conceptually a cleaner solution). In our simulations, we assume that the extra state is stored per physical page.

DSF stores one bit of state (called a *fetch bit*) and a counter (three bits) per block to determine whether, on a block miss, only the requested subblock should be loaded, or whether the entire block should be loaded all at once. Upon a miss, the fetch bit is examined to decide

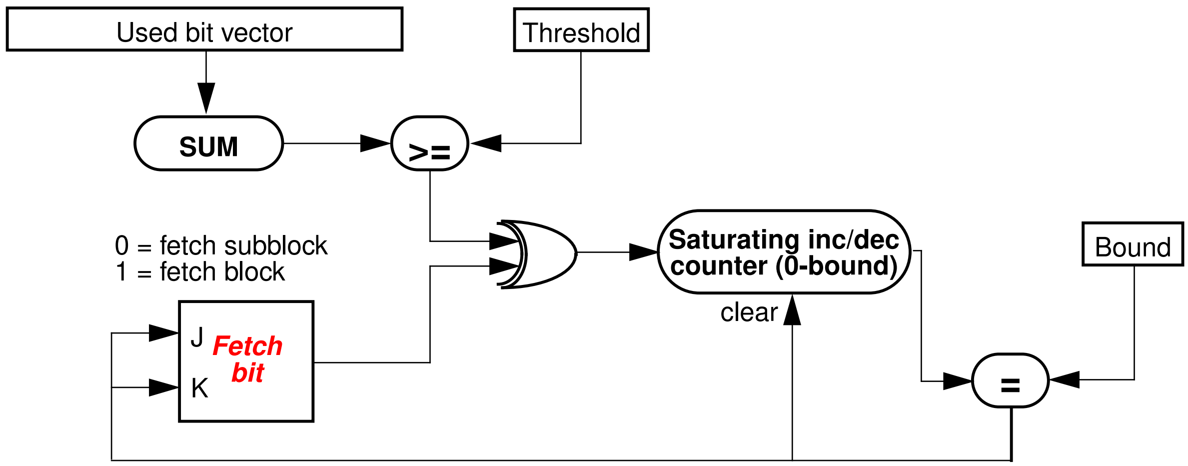


Figure 4-1: Logic for dual-size fetch policy

what to fetch (a zero results in a subblock fetch, whereas a one results in a block fetch). Upon a replacement of a block, DSF updates the state for the victim, which may or may not result in the fetch bit being toggled. This processing occurs off-line and not on any access critical path.

In Figure 4-1, we depict the logic that updates the fetch bit. When **X** is evicted from the cache, the hardware counts the number of *used* subblocks (shown by the **SUM** function). Note that the used bit vector, maintained as a part of the cache state, is distinct from the subblock valid bit vector (*i.e.*, a subblock may be valid but never used). If that number is greater than or equal to a predetermined threshold, the three-bit saturating counter associated with **X** is incremented (if the valid subblocks are less than the threshold, the counter is decremented). If the counter reaches a hardware-specified bound, the fetch bit is toggled and the counter is cleared. The XOR gate is used to allow the policy to work in the reverse direction; the same logic thus handles promotion and demotion.

The cache can thus dynamically determine blocks for which spatial locality is high (because numerous subblocks are valid when the block is evicted), and will eventually fetch the entire block upon a miss. If DSF dictates that a block should be fetched in its entirety, and then few of the fetched subblocks actually get used before replacement, DSF will adapt in the other direction, eventually fetching only a subblock at a time for the block in question.

In Table 4-2, Table 4-3, and Table 4-4, we show the effects that DSF has upon L2 misses and traffic. In each table, we list nine cache organizations along the columns of the table. They are:

Benchmark	L2 cache	Unit	256B	4KB	SUB	threshold-bound 2-2	2-4	4-2	4-4	8-2	8-4
126.gcc	512K, 4sa	Miss	0.064	0.46	1.91	1.40	1.45	1.53	1.59	1.66	1.69
		Traff.		7.47	1.89	3.01	2.76	2.52	2.33	2.19	2.09
	512K, fa	Miss	0.064	0.47	2.06	1.47	1.53	1.60	1.68	1.75	1.83
		Traff.		7.44	1.99	3.17	2.94	2.66	2.47	2.29	2.19
	1M, 4sa	Miss	0.023	0.38	2.24	1.48	1.54	1.59	1.65	1.74	1.80
		Traff.		5.62	2.09	2.95	2.81	2.70	2.60	2.47	2.39
	1M, fa	Miss	0.024	0.39	2.62	1.69	1.76	1.81	1.90	2.00	2.11
		Traff.		5.78	2.39	3.37	3.26	3.09	2.96	2.78	2.67
129.compress	512K, 4sa	Miss	0.007	0.69	3.73	2.56	2.69	2.74	2.90	2.97	3.13
		Traff.		10.62	3.33	4.81	4.46	4.27	3.97	3.81	3.63
	512K, fa	Miss	0.017	0.12	1.66	0.92	0.93	0.94	0.95	0.97	0.99
		Traff.		2.09	1.65	1.82	1.82	1.81	1.80	1.78	1.77
	1M, 4sa	Miss	0.003	0.07	1.08	0.62	0.61	0.62	0.61	0.68	0.61
		Traff.		1.16	1.09	1.11	1.11	1.11	1.11	1.10	1.11
	1M, fa	Miss	0.004	0.07	1.11	0.62	0.64	0.62	0.64	0.63	0.65
		Traff.		1.21	1.11	1.14	1.14	1.13	1.13	1.13	1.13
134.perl	512K, 4sa	Miss	0.139	0.67	1.50	1.27	1.29	1.30	1.33	1.33	1.35
		Traff.		9.87	1.49	1.93	1.73	1.68	1.59	1.58	1.54
	512K, fa	Miss	0.154	0.67	1.55	1.31	1.37	1.37	1.42	1.41	1.45
		Traff.		9.96	1.53	2.11	1.92	1.79	1.69	1.63	1.60
	1M, 4sa	Miss	0.109	0.58	1.44	1.19	1.21	1.23	1.25	1.27	1.27
		Traff.		8.39	1.39	1.85	1.67	1.60	1.50	1.49	1.46
	1M, fa	Miss	0.122	0.59	1.57	1.29	1.37	1.35	1.43	1.40	1.47
		Traff.		8.62	1.52	2.16	1.98	1.83	1.75	1.67	1.62
147.vortex	512K, 4sa	Miss	0.140	1.76	2.72	2.64	2.69	2.70	2.71	2.71	2.71
		Traff.		26.56	2.44	3.14	2.63	2.57	2.46	2.46	2.46
	512K, fa	Miss	0.113	2.32	3.53	3.34	3.39	3.45	3.48	3.51	3.52
		Traff.		34.63	3.15	4.23	3.70	3.37	3.24	3.17	3.15
	1M, 4sa	Miss	0.076	1.99	3.32	3.15	3.23	3.27	3.31	3.31	3.32
		Traff.		29.29	2.89	4.04	3.32	3.13	2.93	2.94	2.90
	1M, fa	Miss	0.077	2.08	3.63	3.30	3.37	3.48	3.53	3.60	3.61
		Traff.		30.85	3.18	4.75	4.16	3.58	3.37	3.24	3.19

Table 4-2: Dual-size fetch functional results, part 1

a cache with 256B blocks, a cache with 4KB blocks, a subblocked cache (4KB blocks, 256B subblocks), and a similar subblocked cache that implements DSF. For the dual-size fetch cache, we present results for six combinations of different values for the threshold and bound depicted in Figure 4-1 (2-2, 2-4, 4-2, 4-4, 8-2, 8-4). For example, the 4-2 experiment would increment the counter when four or more subblocks had been used when a block was evicted, and would promote the block to fetching the whole block when the counter reached the bound of two. Higher values of either will be less likely to promote blocks. The default policy is for all blocks to load only a subblock at a time.

						threshold-bound					
Benchmark	**L2 cache**	**Unit**	**256B**	**4KB**	**SUB**	**2-2**	**2-4**	**4-2**	**4-4**	**8-2**	**8-4**
101.tomcatv	512K, 4sa	Miss	0.074	0.06	1.03	0.55	0.55	0.55	0.55	0.55	-0.00
		Traff.		1.05	1.03	1.05	1.05	1.05	1.05	1.05	-0.00
	512K, fa	Miss	0.078	0.07	1.03	0.56	0.56	0.56	0.57	0.56	0.57
		Traff.		1.10	1.03	1.06	1.06	1.06	1.06	1.06	1.06
	1M, 4sa	Miss	0.073	0.06	1.03	0.56	0.56	0.56	0.56	0.56	0.56
		Traff.		1.05	1.03	1.05	1.05	1.05	1.05	1.05	1.05
	1M, fa	Miss	0.075	0.06	1.03	0.55	0.56	0.55	0.56	0.55	0.56
		Traff.		1.07	1.03	1.05	1.05	1.05	1.05	1.05	1.05
102.swim	512K, 4sa	Miss	0.092	0.14	1.02	0.58	0.57	0.58	0.57	0.58	0.57
		Traff.		2.44	1.03	1.09	1.09	1.08	1.09	1.09	1.09
	512K, fa	Miss	0.096	0.14	1.03	0.88	0.92	0.89	0.93	0.90	-0.00
		Traff.		2.47	1.04	1.57	1.14	1.56	1.12	1.55	-0.00
	1M, 4sa	Miss	0.090	0.14	1.04	0.68	0.59	0.68	0.59	0.68	0.58
		Traff.		2.37	1.05	1.16	1.12	1.16	1.12	1.16	1.11
	1M, fa	Miss	0.092	0.13	1.03	0.89	0.92	0.89	0.93	0.90	-0.00
		Traff.		2.30	1.05	1.58	1.15	1.58	1.13	1.57	-0.00
103.su2cor	512K, 4sa	Miss	0.055	0.11	1.14	0.66	0.67	0.67	0.68	-0.00	0.70
		Traff.		1.80	1.14	1.28	1.28	1.27	1.27	-0.00	1.26
	512K, fa	Miss	0.058	0.09	1.11	0.64	0.65	0.65	0.66	0.67	0.69
		Traff.		1.53	1.11	1.24	1.23	1.22	1.21	1.20	1.19
	1M, 4sa	Miss	0.036	0.08	1.07	0.60	0.60	0.60	0.60	0.62	0.62
		Traff.		1.28	1.06	1.15	1.15	1.15	1.15	1.15	1.14
	1M, fa	Miss	0.041	0.08	1.09	0.61	-0.00	0.62	0.63	0.63	0.64
		Traff.		1.37	1.09	1.18	-0.00	1.16	1.16	1.15	1.14
104.hydro2d	512K, 4sa	Miss	0.095	0.08	1.01	0.58	0.58	0.58	0.57	0.58	0.58
		Traff.		1.30	1.01	1.09	1.09	1.08	1.09	1.08	1.09
	512K, fa	Miss	0.098	0.08	1.02	0.61	0.64	0.62	0.65	0.63	0.66
		Traff.		1.32	1.02	1.14	1.15	1.13	1.15	1.13	1.15
	1M, 4sa	Miss	0.090	0.08	1.02	0.58	0.58	0.58	0.57	0.58	0.58
		Traff.		1.26	1.02	1.09	1.09	1.09	1.09	1.09	1.09
	1M, fa	Miss	0.086	0.08	1.04	0.61	0.63	0.61	0.63	0.62	0.64
		Traff.		1.27	1.04	1.13	1.14	1.13	1.14	1.12	1.13

Table 4-3: Dual-size fetch functional results, part 2

For each benchmark in the tables, we list four caches in separate rows: varying the size between 512KB and 1MB, and varying the associativity between 4-way (with an LRU replacement policy) and full (with a random replacement policy, since LRU is not practical to implement in fully associative caches, particularly lower-level caches). For each cache, there are two rows, in which we show how the misses (higher row) and the traffic (lower row) vary across the different cache organizations. The misses and traffic are normalized to that of the 256B block cache for each pair of rows. The column containing the misses for the 256B block

						threshold-bound					
Benchmark	**L2 cache**	**Unit**	**256B**	**4KB**	**SUB**	**2-2**	**2-4**	**4-2**	**4-4**	**8-2**	**8-4**
107.mgrid	512K, 4sa	Miss	0.079	0.12	1.10	0.62	0.63	0.63	0.66	0.75	0.76
		Traff.		2.02	1.10	1.40	1.36	1.34	1.33	1.16	1.15
	512K, fa	Miss	0.089	0.10	1.08	0.63	0.65	0.70	0.72	0.77	0.92
		Traff.		1.73	1.07	1.38	1.36	1.34	1.35	1.27	1.17
	1M, 4sa	Miss	0.069	0.10	1.16	0.66	0.64	0.67	0.65	0.77	0.78
		Traff.		1.72	1.14	1.40	1.40	1.38	1.40	1.21	1.21
	1M, fa	Miss	0.075	0.09	1.11	0.64	0.66	0.70	0.71	0.78	0.90
		Traff.		1.60	1.10	1.35	1.32	1.36	1.30	1.32	1.19
110.applu	512K, 4sa	Miss	0.091	0.22	1.14	0.72	0.72	0.73	0.72	0.74	0.72
		Traff.		3.72	1.15	1.22	1.22	1.22	1.21	1.21	1.21
	512K, fa	Miss	0.097	0.08	1.01	0.61	0.63	0.63	0.66	0.66	0.70
		Traff.		1.42	1.01	1.18	1.15	1.17	1.12	1.15	1.10
	1M, 4sa	Miss	0.087	0.13	1.08	0.64	0.64	0.65	0.65	0.66	0.66
		Traff.		2.28	1.08	1.17	1.15	1.15	1.15	1.15	1.15
	1M, fa	Miss	0.090	0.08	1.00	0.60	0.62	0.62	0.65	0.64	0.68
		Traff.		1.29	1.00	1.14	1.10	1.14	1.08	1.13	1.07
125.turb3d	512K, 4sa	Miss	0.110	0.83	1.82	1.42	1.44	1.61	1.62	1.63	1.64
		Traff.		13.63	1.83	3.01	3.01	2.00	2.01	1.86	1.86
	512K, fa	Miss	0.092	0.77	1.59	1.41	1.52	1.44	1.55	1.47	1.57
		Traff.		12.24	1.57	1.98	1.78	1.71	1.65	1.67	1.61
	1M, 4sa	Miss	0.098	0.38	1.52	1.03	1.02	1.23	1.23	1.30	1.31
		Traff.		6.24	1.52	2.67	2.69	1.74	1.75	1.57	1.55
	1M, fa	Miss	0.083	0.22	1.17	0.86	0.94	0.97	1.10	1.03	1.14
		Traff.		3.72	1.17	1.91	1.78	1.52	1.35	1.30	1.22
141.apsi	512K, 4sa	Miss	0.107	0.53	1.73	1.63	1.64	1.63	1.64	1.64	1.64
		Traff.		8.11	1.66	1.76	1.71	1.70	1.70	1.70	1.70
	512K, fa	Miss	0.033	0.47	1.92	1.53	1.62	1.57	1.66	1.61	1.70
		Traff.		6.78	1.78	2.05	2.01	2.00	1.96	1.96	1.92
	1M, 4sa	Miss	0.015	0.09	1.80	1.32	1.38	1.33	1.38	1.32	1.40
		Traff.		1.41	1.64	1.74	1.72	1.70	1.71	1.69	1.69
	1M, fa	Miss	0.018	0.09	1.16	0.69	0.73	0.70	0.73	0.71	0.75
		Traff.		1.37	1.15	1.22	1.22	1.22	1.21	1.21	1.20

Table 4-4: Dual-size fetch functional results, part 3

cache contains the absolute (unnormalized) miss ratio for that experiment, to which the other columns are normalized.

Several trends are visible in this data. First, as expected, the miss rate generally goes up as DSF becomes more restrictive (harder to promote or demote pages, moving toward the right in the tables). The traffic, which generally increases as the miss rates are lowered, decreases as the policies become more restrictive. The fully associative runs with random replacement generally incur more misses than the 4-way set associative runs. DSF tends to eliminate fewer misses with the fully associative experiments, as the random replacement can evict blocks too early (while they are still in the working set), introducing less accurate state into the block counters. There is little correlation when comparing the effect of cache size against the efficacy of the policy; for many of the benchmarks, the policy is more effective at reducing misses for the larger 1MB cache; for others, DSF works better for the 512KB cache.

In terms of overall performance, DSF performs well in some cases and poorly in others. In every case, DSF reduces the miss ratio over a traditional subblocked cache, frequently with only a minor increase in traffic. However, the subblocked cache itself incurs a large performance penalty for some of the benchmarks when compared to a 256B block cache, which loads the same amount of data but has many more sets (multiplied by the subblocking factor) in which to store data. The performance penalty is particularly acute for the integer codes we measured, which tend to have finer-grain accesses and thus could benefit from having more sets. For gcc, the subblocked cache incurred twice as many misses, for perl, 50% more misses, and for vortex, three times as many misses. Turb3d and Apsi see 80% and 60% increases in misses, respectively. The other floating point codes we measured (tomcatv, swim, su2cor, hydro2d, mgrid, and applu) typically incur miss increases of more than 10% for a subblocked cache, primarily because there is a closer correlation between their pollution point and the block size.

In most cases, the penalty incurred by using a subblocked cache outweighs the gains from DSF, which incurs more misses than a 256B block cache for turn3d, gcc, perl, and vortex. For tomcatv, su2cor, hydro2d, and mgrid, DSF has lower miss ratios than a 256B block cache, but not nearly as low as those of a 4KB block cache (which shows little additional traffic because spatial locality is so high for these benchmarks). For swim and applu, DSF shows fewer misses than any of the alternatives, with minor additional traffic (roughly 30% fewer misses with 15% extra traffic for both benchmarks).

4.3 Subblock prefetching

DSF may be effective if most of a large block is used, but if discontiguous subblocks within a block are rarely accessed, the system could benefit from identifying those subblocks and not loading them upon a block miss. Ideally, the cache would fetch only those subblocks that will be accessed.

Hill [58] describes several prefetching policies for subblocked instruction caches: *remainder*, *wrap-around*, and *always*, which prefetch the next subblock (if the subblock referenced was not the last in the block), the next subblock (wrapping around if the referenced subblock is the last one), and fetching the next subblock (even if it resides in the next block) respectively. All of these policies initiate the prefetches on a reference. Hill also proposed [58] the SPUR prefetch algorithm, which waits for an idle bus cycle (similar to bus prioritization described in Section 4.5) to initiate a prefetch of the subblock adjacent to that which caused the last demand miss. In this section, we describe a scheme that differs from these prefetching schemes by fetching discontiguous sets of subblocks at once.

Kumar and Wilkerson proposed a policy called *spatial footprinting*, in which a (possibly discontiguous) set of subblocks are loaded upon a block miss [81]. We independently proposed a nearly identical policy that we called *subblock prefetching* (or *SBP*) [16]. SBP saves not just a bit and counter when some block **X** is evicted, as in DSF, but also the used bit vector representing the subblocks that were accessed while **X** was in the cache. If **X** shows enough consistency for the set of subblocks that are used among block misses to **X**, the SBP policy will begin fetching only those subblocks that were touched while **X** was last in the cache (plus the requested subblock, if it was not marked in the vector).

Since not every block is likely to show consistent usage patterns, we use a dynamic scheme (similar to DSF) to identify those blocks that do show consistent usage of subblocks. Upon a block miss, the SBP bit is examined to determined whether subblocks other than that requested should be fetched. When a block is replaced, the block's state is examined and saved, and the SBP bit is updated. We show the logic that performs this replacement analysis in Figure 4-2.

While a block **X** is in the cache, three bit vectors are maintained. The *valid bit vector* identifies those subblocks in **X** that are valid. The *used bit vector* identifies those subblocks that the processor has actually accessed. The *previous use vector* contains the subblocks that were used the last time that **X** was resident in the cache. When **X** is replaced, the hardware com-

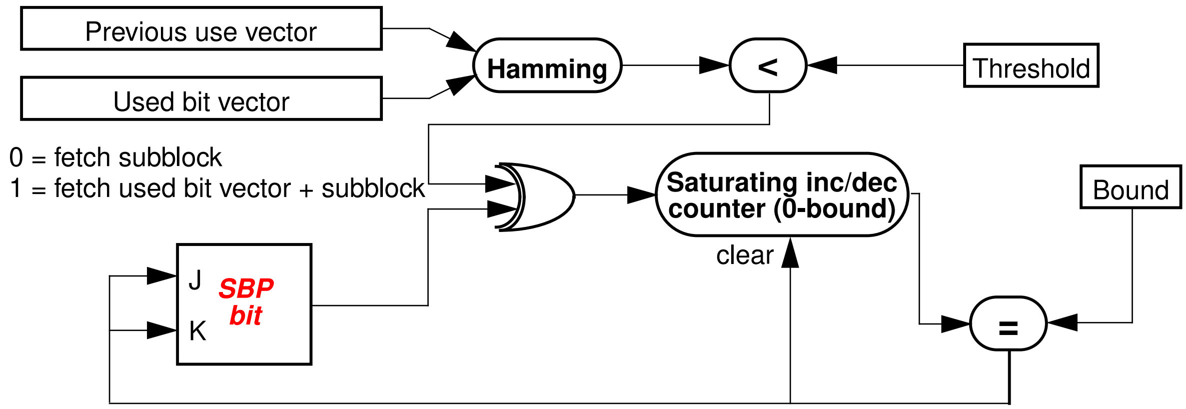


Figure 4-2: Logic for subblock prefetching policy

putes the hamming distance between the used bit vector and the previous use vector. If the Hamming distance is below some threshold, a saturating counter (similar to the dual-size policy) is incremented, otherwise the counter is decremented. If the counter reaches a certain value, **X** is marked as a candidate for subblock prefetching, and upon the next block miss to **X**, the subblocks that are marked in the previous use vector are loaded from memory.

When **X** is evicted, the used bit vector becomes the previous bit vector, and is then stored along with the counter in the TLB or separate table). Like DSF, SBP supports demotion of blocks from performing the used bit vector prefetching. Ideally, this policy will identify blocks that have consistent usage patterns, and subsequently refrain from fetching subblocks that are rarely used, thus reducing bus contention without significantly increasing misses. As an optimization (not shown in Figure 4-2), we require that more than one subblock be valid for the promotion counter to be incremented (in addition to requiring that the Hamming distance be sufficiently low).

In Table 4-5, Table 4-6, and Table 4-7, we show the misses versus traffic behavior for the SBP policy, which are formatted identically to the results shown for DSF in Section 4.2. Like DSF, SBP is unable, except in a few cases, to reduce the miss rate more than the increase caused by incorporating the subblocked cache. This phenomenon is particularly true for the integer benchmarks (gcc, perl, vortex), which lose considerable performance when the cache is subblocked. SBP does, however, demonstrate consistent improvement over the subblocked cache. SBP is also less effective than DSF at reducing the miss rate for most of the bench-

Benchmark	L2 cache	Unit	256B	4KB	SUB	threshold-bound 2-2	2-4	4-2	4-4	8-2	8-4
126.gcc	512K, 4sa	Miss	0.064	0.46	1.91	1.80	1.81	1.75	1.76	1.59	1.55
		Traff.		7.47	1.89	1.95	1.94	2.05	2.01	2.35	2.41
	512K, fa	Miss	0.064	0.47	2.06	1.91	1.95	1.83	1.87	1.71	1.73
		Traff.		7.44	1.99	2.09	2.07	2.17	2.16	2.30	2.31
	1M, 4sa	Miss	0.023	0.38	2.24	2.04	2.07	1.98	2.02	1.80	1.76
		Traff.		5.62	2.09	2.15	2.13	2.20	2.17	2.39	2.44
	1M, fa	Miss	0.024	0.39	2.62	2.36	2.42	2.27	2.33	2.12	2.16
		Traff.		5.78	2.39	2.46	2.45	2.51	2.50	2.59	2.58
129.compress	512K, 4sa	Miss	0.007	0.69	3.73	3.42	3.47	3.33	3.39	3.05	2.95
		Traff.		10.62	3.33	3.41	3.39	3.49	3.47	3.80	3.91
	512K, fa	Miss	0.017	0.12	1.66	1.36	1.39	1.34	1.36	1.30	1.31
		Traff.		2.09	1.65	1.67	1.66	1.67	1.67	1.68	1.67
	1M, 4sa	Miss	0.003	0.07	1.08	0.56	0.53	0.56	0.53	0.54	0.53
		Traff.		1.16	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	1M, fa	Miss	0.004	0.07	1.11	0.87	0.88	0.87	0.88	0.86	0.87
		Traff.		1.21	1.11	1.11	1.11	1.11	1.11	1.11	1.11
134.perl	512K, 4sa	Miss	0.139	0.67	1.50	1.38	1.38	1.36	1.36	1.32	1.31
		Traff.		9.87	1.49	1.55	1.52	1.82	1.67	2.00	1.90
	512K, fa	Miss	0.154	0.67	1.55	1.49	1.51	1.47	1.50	1.43	1.47
		Traff.		9.96	1.53	1.59	1.55	1.78	1.62	1.96	1.83
	1M, 4sa	Miss	0.109	0.58	1.44	1.32	1.32	1.30	1.31	1.25	1.25
		Traff.		8.39	1.39	1.44	1.41	1.64	1.57	1.84	1.79
	1M, fa	Miss	0.122	0.59	1.57	1.51	1.54	1.49	1.53	1.44	1.48
		Traff.		8.62	1.52	1.56	1.54	1.72	1.59	1.91	1.80
147.vortex	512K, 4sa	Miss	0.140	1.76	2.72	2.67	2.67	2.63	2.63	2.57	2.56
		Traff.		26.56	2.44	2.82	2.83	3.22	3.26	3.64	3.69
	512K, fa	Miss	0.113	2.32	3.53	3.33	3.34	3.25	3.24	3.19	3.18
		Traff.		34.63	3.15	3.56	3.56	3.90	3.92	4.08	4.08
	1M, 4sa	Miss	0.076	1.99	3.32	3.27	3.28	3.21	3.22	3.10	3.09
		Traff.		29.29	2.89	3.25	3.22	3.73	3.80	4.23	4.30
	1M, fa	Miss	0.077	2.08	3.63	3.40	3.41	3.29	3.29	3.19	3.18
		Traff.		30.85	3.18	3.50	3.48	3.82	3.80	4.08	4.09

Table 4-5: Subblock prefetch functional results, part 1

marks, since it loads less data into the cache speculatively. However, SBP is considerably more efficient at reducing misses without increasing traffic. We can quantify *policy efficiency* by calculating the ratio of the percent of misses reduced to the percent traffic increase:

$$\frac{(M_{\text{sb}} - M_{\text{SBP}})/M_{\text{sb}}}{(T_{\text{SBP}} - M_{\text{sb}})/T_{\text{sb}}} \tag{4-1}$$

where M_{sb} represents the L2 misses for the subblocked cache, and M_{SBP} represents the L2 misses for a subblocked cache with SBP. T_{sb} and T_{SBP} represent the total traffic for those two

						threshold-bound					
Benchmark	**L2 cache**	**Unit**	**256B**	**4KB**	**SUB**	**2-2**	**2-4**	**4-2**	**4-4**	**8-2**	**8-4**
101.tomcatv	512K, 4sa	Miss	0.074	0.06	1.03	0.54	0.54	0.54	0.54	0.54	0.54
		Traff.		1.05	1.03	1.03	1.03	1.03	1.03	1.03	1.03
	512K, fa	Miss	0.078	0.07	1.03	0.79	0.80	0.79	0.80	0.79	0.80
		Traff.		1.10	1.03	1.03	1.03	1.03	1.03	1.04	1.03
	1M, 4sa	Miss	0.073	0.06	1.03	0.55	0.56	0.55	0.56	0.55	0.56
		Traff.		1.05	1.03	1.03	1.03	1.03	1.03	1.03	1.03
	1M, fa	Miss	0.075	0.06	1.03	0.79	0.79	0.79	0.79	0.79	0.79
		Traff.		1.07	1.03	1.03	1.03	1.03	1.03	1.03	1.03
102.swim	512K, 4sa	Miss	0.092	0.14	1.02	0.75	0.70	0.75	0.70	0.75	0.70
		Traff.		2.44	1.03	1.16	1.20	1.16	1.20	1.17	1.20
	512K, fa	Miss	0.096	0.14	1.03	0.93	0.99	0.93	0.97	0.91	0.95
		Traff.		2.47	1.04	1.07	1.06	1.07	1.06	1.08	1.07
	1M, 4sa	Miss	0.090	0.14	1.04	0.78	0.76	0.77	0.76	0.76	0.75
		Traff.		2.37	1.05	1.19	1.23	1.19	1.23	1.19	1.22
	1M, fa	Miss	0.092	0.13	1.03	0.94	0.98	0.94	0.96	0.92	0.94
		Traff.		2.30	1.05	1.06	1.06	1.06	1.06	1.07	1.07
103.su2cor	512K, 4sa	Miss	0.055	0.11	1.14	0.81	0.80	0.80	0.78	0.77	0.74
		Traff.		1.80	1.14	1.18	1.18	1.18	1.19	1.19	1.20
	512K, fa	Miss	0.058	0.09	1.11	0.91	0.93	0.91	0.92	0.89	0.90
		Traff.		1.53	1.11	1.13	1.13	1.13	1.13	1.14	1.14
	1M, 4sa	Miss	0.036	0.08	1.07	0.73	0.72	0.72	0.70	0.70	0.67
		Traff.		1.28	1.06	1.09	1.09	1.09	1.09	1.09	1.10
	1M, fa	Miss	0.041	0.08	1.09	0.87	0.88	0.86	0.88	0.85	0.86
		Traff.		1.37	1.09	1.10	1.10	1.10	1.10	1.10	1.10
104.hydro2d	512K, 4sa	Miss	0.095	0.08	1.01	0.62	0.61	0.62	0.60	0.61	0.60
		Traff.		1.30	1.01	1.07	1.07	1.08	1.07	1.08	1.07
	512K, fa	Miss	0.098	0.08	1.02	0.84	0.87	0.84	0.86	0.83	0.85
		Traff.		1.32	1.02	1.06	1.05	1.06	1.05	1.06	1.06
	1M, 4sa	Miss	0.090	0.08	1.02	0.64	0.62	0.63	0.61	0.63	0.61
		Traff.		1.26	1.02	1.07	1.08	1.08	1.08	1.08	1.08
	1M, fa	Miss	0.086	0.08	1.04	0.84	0.86	0.84	0.86	0.83	0.85
		Traff.		1.27	1.04	1.06	1.06	1.06	1.06	1.07	1.06

Table 4-6: Subblock prefetch functional results, part 2

caches, respectively. Informally, this metric measures how successful a policy is at reducing misses while increasing traffic as little as possible (or vice-versa, decreasing traffic while minimally increasing misses). In Table 4-9, we show the policy efficiencies for DSF and SBP. The efficiencies shown were calculated for 1MB, 4-way set associative caches, with threshold and bound values of 2 for both DSF and SBP. The table shows that the policy efficiencies are indeed significantly higher for SBP in all cases but two; swim and vortex (and with vortex, they are nearly identical, and uniformly poor). Note that this metric does not quantify the

						threshold-bound					
Benchmark	**L2 cache**	**Unit**	**256B**	**4KB**	**SUB**	**2-2**	**2-4**	**4-2**	**4-4**	**8-2**	**8-4**
107.mgrid	512K, 4sa	Miss	0.079	0.12	1.10	0.79	0.78	0.77	0.77	0.73	0.70
		Traff.		2.02	1.10	1.15	1.14	1.17	1.16	1.21	1.23
	512K, fa	Miss	0.089	0.10	1.08	1.00	1.04	0.96	1.02	0.82	0.83
		Traff.		1.73	1.07	1.10	1.08	1.10	1.08	1.12	1.12
	1M, 4sa	Miss	0.069	0.10	1.16	0.82	0.79	0.79	0.78	0.74	0.73
		Traff.		1.72	1.14	1.19	1.19	1.19	1.20	1.23	1.23
	1M, fa	Miss	0.075	0.09	1.11	1.04	1.07	1.01	1.06	0.84	0.85
		Traff.		1.60	1.10	1.12	1.10	1.12	1.10	1.13	1.13
110.applu	512K, 4sa	Miss	0.091	0.22	1.14	0.87	0.83	0.86	0.80	0.82	0.78
		Traff.		3.72	1.15	1.18	1.19	1.19	1.28	1.27	1.29
	512K, fa	Miss	0.097	0.08	1.01	0.88	0.91	0.87	0.90	0.84	0.87
		Traff.		1.42	1.01	1.03	1.02	1.03	1.02	1.04	1.03
	1M, 4sa	Miss	0.087	0.13	1.08	0.71	0.71	0.70	0.68	0.67	0.66
		Traff.		2.28	1.08	1.13	1.13	1.13	1.14	1.15	1.18
	1M, fa	Miss	0.090	0.08	1.00	0.85	0.87	0.84	0.87	0.83	0.85
		Traff.		1.29	1.00	1.02	1.01	1.02	1.01	1.02	1.02
125.turb3d	512K, 4sa	Miss	0.110	0.83	1.82	1.66	1.65	1.65	1.64	1.63	1.62
		Traff.		13.63	1.83	2.03	2.04	2.08	2.08	2.34	2.34
	512K, fa	Miss	0.092	0.77	1.59	1.49	1.47	1.47	1.45	1.46	1.44
		Traff.		12.24	1.57	1.70	1.71	1.73	1.74	1.78	1.76
	1M, 4sa	Miss	0.098	0.38	1.52	1.33	1.33	1.31	1.31	1.27	1.27
		Traff.		6.24	1.52	1.61	1.61	1.67	1.65	1.93	1.95
	1M, fa	Miss	0.083	0.22	1.17	1.12	1.15	1.08	1.09	0.99	0.99
		Traff.		3.72	1.17	1.21	1.19	1.26	1.24	1.36	1.34
141.apsi	512K, 4sa	Miss	0.107	0.53	1.73	1.31	1.31	1.31	1.30	1.30	1.29
		Traff.		8.11	1.66	1.73	1.73	1.73	1.73	1.74	1.74
	512K, fa	Miss	0.033	0.47	1.92	1.60	1.60	1.58	1.58	1.55	1.55
		Traff.		6.78	1.78	1.87	1.87	1.88	1.88	1.90	1.90
	1M, 4sa	Miss	0.015	0.09	1.80	1.32	1.29	1.33	1.28	1.32	1.26
		Traff.		1.41	1.64	1.68	1.69	1.69	1.69	1.69	1.70
	1M, fa	Miss	0.018	0.09	1.16	0.96	0.98	0.95	0.98	0.94	0.96
		Traff.		1.37	1.15	1.16	1.15	1.16	1.16	1.16	1.16

Table 4-7: Subblock prefetch functional results, part 3

absolute performance of a policy in terms of miss reduction, simply how efficient the policy is at balancing misses and traffic.

4.4 Unifying DSF and SBP

Since the SBP policy is generally more efficient at balancing traffic and misses than the DSF policy, but the DSF policy shows a much larger absolute reduction in the number of misses, we implemented a policy that incorporates both DSF and SBP. The policy works as follows:

both sets of state are maintained and updated upon each block eviction as shown in Figure 4-1 and Figure 4-2. (The total new state required equals 34 bits per block, about 0.1%.) In the unified policy, we append the fetch bit to the subblock prefetch bit, and use those two bits to decide what to fetch upon a block miss. If the state contains 11 or 10, we use the SBP policy (the SBP bit overrides the fetch bit). On a 01, we fetch the block, and on a 00, we fetch only the requested subblock.

In Table 4-8, we show the results of functional simulations comparing misses and traffic for DSF, SBP, and the two together. As in the previous tables, we show the absolute miss rate for 256B block caches, and then relative misses and traffic for all other runs, normalized to those of the 256B block cache runs. For two of the benchmarks (apsi and compress), the unified policy shows a significant reduction in misses (9% and 22%, respectively) above and beyond that offered by the best of either DSF or SBP. For several of the other benchmarks, we see small reductions in misses with unified (1% for turb3d, su2cor, and gcc) coupled with slight reductions in traffic as well (1%, 2%, 2%, and 3% for gcc, tomcatv, mgrid, and applu, respectively). Only for one case (swim) is the miss ratio larger for the unified policy than for the minimum of DSF and SBP (in this case, it is higher than DSF by 7%).

In the third column of Table 4-9 we list the policy efficiencies of the unified DSF/SBP policy. We see that the policy efficiencies (except for swim) all fall in between those of DSF and SBP. The efficiencies tend to be much closer to those of DSF, except for the two cases in which the unified misses are lower than either of the two policies alone (compress and apsi). In these cases, the policies are working synergistically. In many of the others, highly populated blocks that do not show tightly consistent subblock usage patterns dominate the policy, which causes the unified policy to fetch full blocks rather than discontiguous sets of subblocks.

Most of the reduction in misses comes from the DSF policy, although the unified policy occasionally provides an additional reduction in misses and slight reductions in total traffic. These miss reductions come at the expense of added traffic. In the next subsection, we describe a mechanism for mitigating the performance impact of this additional traffic.

Benchmark	Metric	256B	4096B	Subblocked	DSF	SBP	Unified
126.gcc	Misses	0.023	0.38	2.24	1.48	2.04	1.47
	Traffic		5.62	2.09	2.95	2.15	2.93
129.compress	Misses	0.003	0.07	1.08	0.62	0.56	0.46
	Traffic		1.16	1.09	1.11	1.09	1.10
134.perl	Misses	0.109	0.58	1.44	1.19	1.32	1.19
	Traffic		8.39	1.39	1.85	1.44	1.86
147.vortex	Misses	0.076	1.99	3.32	3.15	3.27	----
	Traffic		29.29	2.89	4.04	3.25	----
101.tomcatv	Misses	0.073	0.06	1.03	0.56	0.55	0.56
	Traffic		1.05	1.03	1.05	1.03	1.03
102.swim	Misses	0.090	0.14	1.04	0.68	0.78	0.73
	Traffic		2.37	1.05	1.16	1.19	1.25
103.su2cor	Misses	0.036	0.08	1.07	0.60	0.73	0.59
	Traffic		1.28	1.06	1.15	1.09	1.14
104.hydro2d	Misses	0.090	0.08	1.02	0.58	0.64	0.59
	Traffic		1.26	1.02	1.09	1.07	1.08
107.mgrid	Misses	0.069	0.10	1.16	0.66	0.82	0.66
	Traffic		1.72	1.14	1.40	1.19	1.37
110.applu	Misses	0.087	0.13	1.08	0.64	0.71	0.64
	Traffic		2.28	1.08	1.17	1.13	1.17
125.turb3d	Misses	0.098	0.38	1.52	1.03	1.33	1.02
	Traffic		6.24	1.52	2.67	1.61	2.58
141.apsi	Misses	0.015	0.09	1.80	1.32	1.32	1.20
	Traffic		1.41	1.64	1.74	1.68	1.72

Table 4-8: Trading off misses and traffic for a 1MB, 4-way set associative L2

	Policy efficiency		
Benchmark	**DSF**	**SBP**	**Unified**
126.gcc	0.821	2.906	0.847
129.compress	24.633	119.008	50.446
134.perl	0.527	2.660	0.529
147.vortex	0.129	0.121	-----
101.tomcatv	25.591	451.727	440.828
102.swim	3.206	1.938	1.556
103.su2cor	5.055	12.780	5.910
104.hydro2d	5.799	6.970	6.605
107.mgrid	1.898	7.037	2.150
110.applu	4.540	7.678	4.686
125.turb3d	0.431	2.234	0.468
141.apsi	4.243	9.883	6.397

Table 4-9: Policy efficiencies; 1MB 4-way set associative L2, threshold and bound = 2

While these schemes can improve the performance of the subblocked cache, the subblocked cache itself takes enough of a performance hit, due to cache pollution (particularly for the finer grained codes) that even with the optimizing policies, it often does not outperform a regular cache. The performance penalty of the subblocked cache may be reduced by mechanisms that allow data to be mapped into the cache at a finer granularity. One possible solution is the decoupled sector cache [103], which associates multiple tags with each block. In Chapter 5, we propose a different solution, which maps data into the cache at a subblock granularity, but uses block-sized tags to keep track of the data.

4.5 Bus prioritization

Speculative loading of subblocks (as determined by DSF and SBP) can worsen performance when higher-priority requests experience longer queueing delays as a result of the speculative loading. Conversely, if no demand fetches are pending, and the bus is otherwise idle, there is no penalty (other than consumed power) for loading subblocks that may soon be needed.

We have implemented a policy called *bus prioritization* that harvests otherwise wasted cycles on the Rambus channel. When DSF or SBP identify subblocks that might be good candidates for prefetches (during a block miss) only the processor-requested subblock is actually requested from main memory. The non-critical subblocks are buffered for loading when the Rambus channel is idle. They are loaded into a circular queue structure that we call a *soft prefetch* queue, depicted in Figure 4-3. An address tag and subblock bit vector are stored in each soft queue entry. We call the queue *soft* because its contents represent prefetch hints only; the tail pointer can overwrite the head pointer at any time if the queue is full. The queue thus simply buffers addresses that might be good candidates for prefetching. This queue bears some resemblance to how a non-blocking cache buffer should be implemented for fetching large blocks. The difference between the two are twofold: (1) *what* data are chosen for fetching (bus prioritization uses the SBP and DSF policies, as opposed to fetching large, albeit prioritized, blocks on every transfer), and (2) that the queue is *soft*; data may not be fetched if the bus is highly utilized and fetches are overwritten in the soft queue.

In addition to the soft prefetch queue, there are also 2 *hard prefetch* MSHRs, which hold actual prefetch requests issued to the Rambus channel. When one of the prefetch MSHRs is freed, the soft prefetch queue is accessed and, if non-empty, a subblock request is moved to

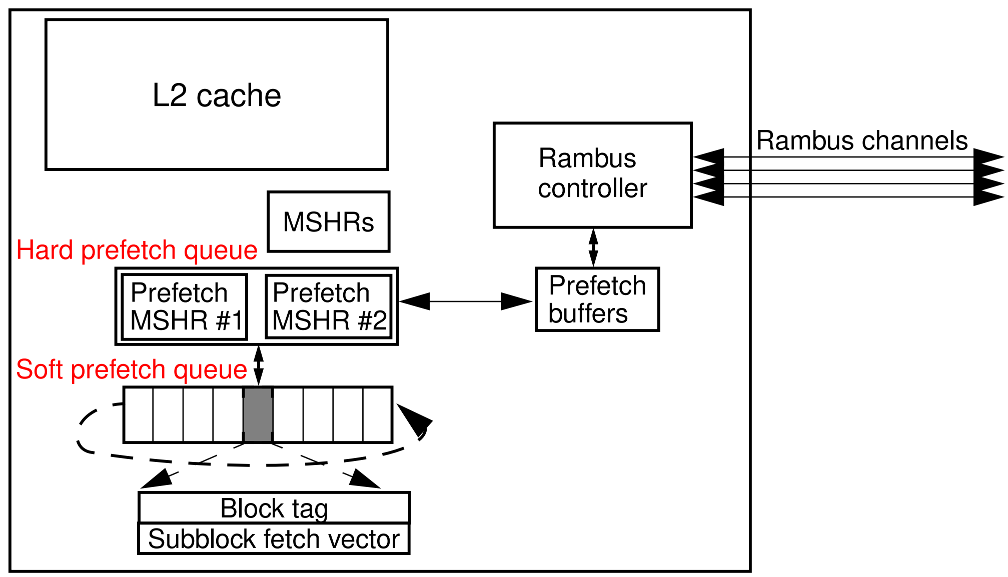


Figure 4-3: Datapath for bus prioritization

the MSHR, and the prefetch request is sent to the Rambus controller. The Rambus controller buffers up to two prefetch requests, only initiating one when the channel is idle. The prefetches can wait indefinitely if demand fetches keep arriving at the Rambus controller. Once the prefetch initiates, however, it is not superseded by arriving requests. Two prefetch MSHRs are sufficient to ensure that a prefetch is always in progress when the bus is otherwise idle, so long as there are subblocks to prefetch. When the processor requests a subblock that is held in the soft prefetch queue, it is removed from the queue (the valid bit associated with the requested block is cleared). When the processor requests a subblock that is in a prefetch MSHR, an upgrade signal is sent to the Rambus controller. The upgraded request then ceases to be superseded by other demand fetches.

This policy attempts to find a balance between two extreme endpoints. At one extreme, all data are fetched with equal priority, lowering the L2 miss ratio but possibly causing long queueing delays for demand fetches, which get queued behind speculative subblock fetches. At the other extreme, no subblocks are fetched speculatively, guaranteeing less queueing delay for demand fetches, but resulting in more L2 misses. With bus prioritization, the longest delay that any demand fetch will see as a result of a speculative subblock fetch is sixteen processor cycles (in our simulated implementation), which occurs when no demand fetches are queued, so the Rambus controller initiates a speculative subblock fetch, and right after that initiation, a demand fetch request arrives.

We measured the execution performance of our traffic policies with and without bus prioritization. The system parameters were identical to those described in Section 4.1 (Direct Rambus, 8-way issue, dynamically scheduled core, etc.) We ran timing simulations for all these policies, and graph the performance results in Figure 4-4. On the y-axis we show performance (measured in IPC), and on the x-axis we display one cluster of seven bars for each benchmark. The left-most bar in each cluster represents an ideal L2 that never misses (but still incurs a 10-cycle hit penalty). The next three bars represent the performance of a 256-byte block cache, a cache with the block size set at the performance point for that benchmark, and a 4KB block cache, respectively. The fifth bar represents the performance of the base subblocked cache (4KB blocks, 256-byte subblocks). The sixth bar shows the performance of our unified policy on the subblocked cache, and the right-most bar shows the performance resultant from adding bus prioritization to the unified policy.

As expected from our functional results, the unified policy breaks even with or outperforms the subblocked cache in most cases (particularly swim, mgrid, su2cor, and compress). Bus prioritization improves performance further in every case except for compress. In two cases (swim and mgrid), the bus prefetching improves performance over that of the "performance point" blocksize by a significant margin (10%). For many of the other benchmarks, however, the subblocked cache degrades performance enough that even with bus prioritization, performance is still lower than a "vanilla" 4-way set associative cache with 256 byte blocks (compress, gcc, vortex, apsi, and turb3d). In vortex, the unified policy itself reduces performance below that of even the subblocked cache, as there is little consistent spatial locality for the DSF and SBP policies to exploit. The bus prioritization regains some of this performance loss, finding idle cycles with which to bring unneeded (in vortex) data across the Rambus channels. Another interesting result can be seen in this graph: in two cases (su2cor and apsi), the "perfect" L2 actually has *lower* performance than some of the other experiments. This aberration occurs because the perfect L2 returns certain blocks too quickly (blocks that would otherwise

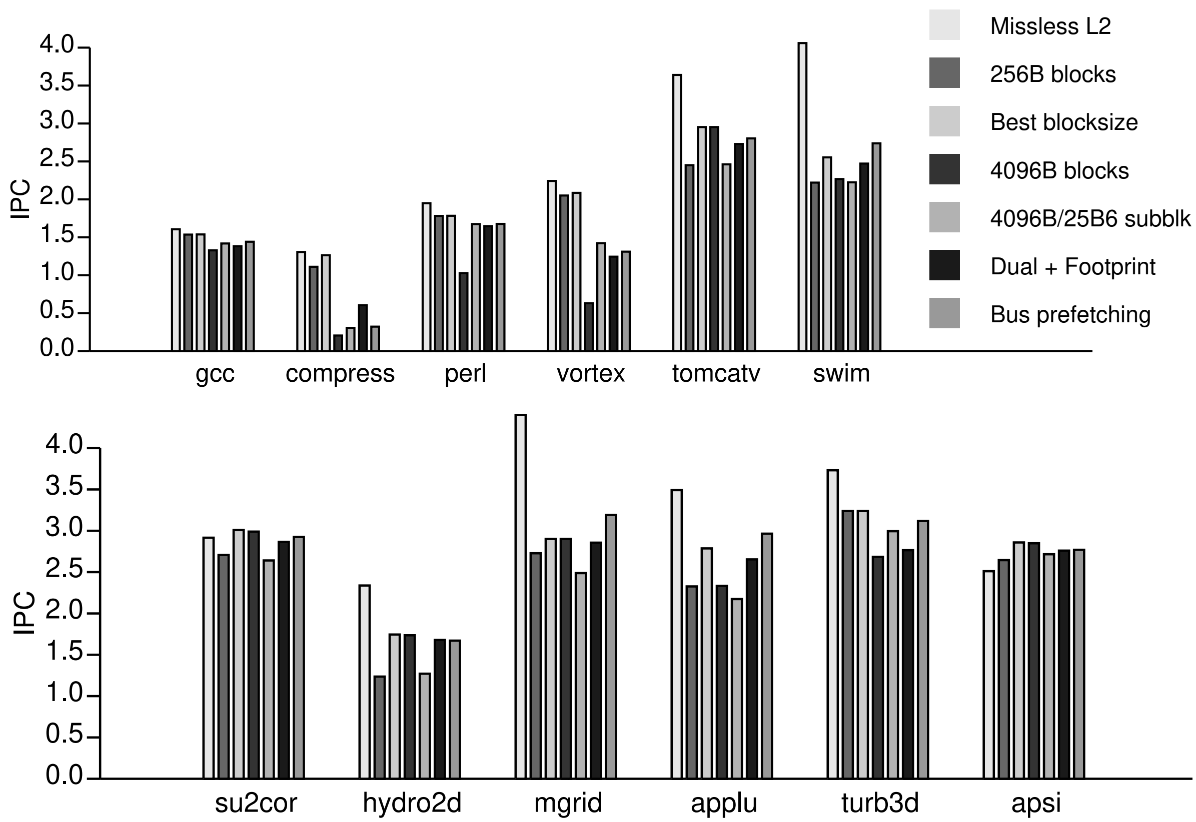


Figure 4-4: Performance of traffic optimization schemes

have missed in the L2); those blocks evict data in the L1 data cache that the processor still needs for a short time, causing a conflict miss. Thus, the L1 miss rates are higher for the "perfect" L2 runs; those extra misses are the source of the performance loss.

Overall, these policies show potential to improve performance. However, the limitations of the implementation (mapping inflexibility) forces the data to be mapped into the cache at a coarse granularity, which results in non-competitive performance (except in two cases). One alternative is to map small blocks into the cache and manage the behavioral state coupled to larger logical regions, as proposed by Johnson and Hwu [69]. While this scheme would require extra buffering near the cache to hold active state (since the state couldn't be stored in the tag array), that extra buffering would be proportional to the cache size and is a possibility. This implementation would increase the number of conflicts generated from the extra blocks being loaded into the cache.

Another alternative to mitigating the mapping granularity problem would be to implement a decoupled sector cache [103], associating multiple tags with each block. The decoupled subblocked cache has the potential to work synergistically with the proposed policies, improving performance above and beyond that attainable with a fixed block size. In the next section, however, we propose a different solution to supporting these policies with a finer-grain cache mapping. Our solution uses indirect indexing to provide flexibility in the cache mapping: mapping data into the cache at a subblock granularity and reducing conflicts, but using tags at the granularity of a block to keep the policy state associated with the blocks. We show that the combination of the traffic policies and the indirect cache provides outstanding performance, which is true for neither of the two individually.

Chapter 5

Merging Caches and Physical Memory

In Chapter 4, we evaluated a number of policies for improving the performance of 1MB caches. Caches of this size will soon appear; the Compaq Alpha 21364 will have 1.5 MB of on-chip cache [56], as will the HP PA-8500. On-processor memory capacities will grow substantially larger than one or two megabytes, however. Intel estimates that microprocessors will contain 350 million transistors by 2006, and well over a billion by 2010 [136]. Most of these transistors will be devoted to memory cells in one form or another—a recent collection of articles on possible directions for microprocessors were unanimous in predicting that the bulk of on-chip transistors will be organized as memory storage [11]. Large on-chip memories (that we will henceforth call *MOPs,* for "memory on processor") are desirable because they reduce both the number of times long memory latencies are incurred and off-chip traffic. What is unclear is how these large MOPs—from megabytes to tens and hundreds of megabytes—will be organized.

Caches and physical memory are managed quite differently, even though they perform similar functions: buffering subsets of frequently used regions of data from a lower level of the memory hierarchy (whether from main memory or disks). As we shall describe below, these future MOPs will come to resemble past physical memories more than caches, both in terms of access times and capacities. As they grow more similar—in terms of critical parameters and ratios—to the physical memories of yore, and less similar to the original caches, using some of the management mechanisms from physical memory may enhance overall performance. In this section, we evaluate a few possible paths by which MOPs may evolve into hybrids of traditional caches and main memories.

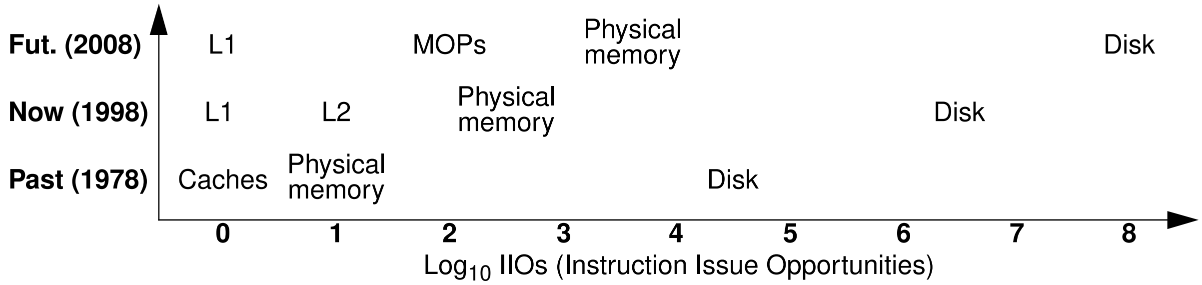


Figure 5-1: Access penalties for levels in the memory hierarchy

The access penalties of MOPs, measured in *instruction issue opportunities*, are beginning to resemble those of physical memories from two decades ago. We consider instruction issue opportunities to be the number of instructions that could be issued by the processor while an access to that level is being serviced.

Cache memories were originally designed to provide low latency access to a small number of operands, which is a role quite different from that which MOPs will play. To illustrate the difference, in Figure 5-1 we show access penalties for various levels in the memory hierarchy in 1978, today, and estimated for a decade hence. In this figure, we calculate instruction issue opportunities as the product of the access time of that memory level, the processor clock rate, and the sustained instructions per cycle. For 1977, we assumed a CPI of ~10, 5 MHz clocks, and disk latencies of 50 ms. For 2007, we estimate disk latencies at 5 ms, large on-chip access penalties at 5 ns, 4GHz clocks, and a sustained IPC of 10 (about what is needed to stay on current performance curves).

By these estimates, the expense of accessing a MOP in 2008 will approach that of accessing main memory today, and accessing physical memory in 2008 is growing close to that of a disk access in 1978. Furthermore, a 2008 MOP will be considerably more expensive to access than physical memory was in 1978.

In addition to access penalties, the capacities of future MOPs (with respect to the rest of the memory hierarchy) will fall somewhere between the traditional sizes of caches and physical memory. Unlike today, on-chip memories may eventually contain a substantial fraction of the physical memory capacity. In Figure 1-5 we show the percentage of processor transistors that are allocated to cache memories, for numerous recent processors. This percentage, already

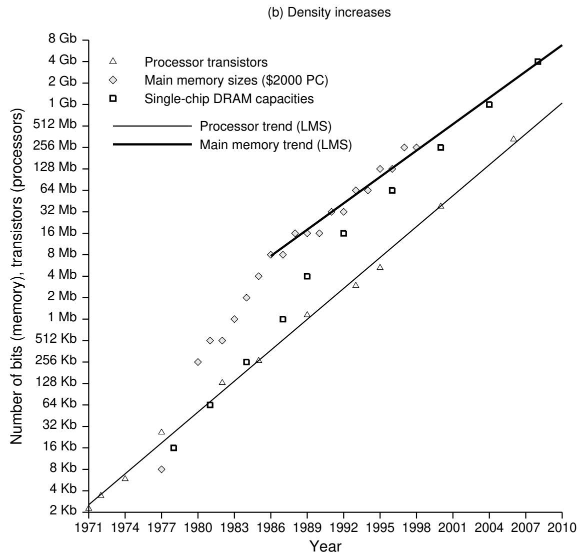


Figure 5-2: Trends in microprocessor memory hierarchies

high, continues to grow. As this trend continues, future processors will have the bulk of their transistors devoted to memory. There will always be fast level-one caches, with a large bank (or banks) residing under the level-one caches.

Given that future processors will be mostly memory, the capacity of the on-processor memories will track processor capacities. In Figure 5-2, we show how processor capacities will scale compared to single DRAM chip capacities and main memory capacities (for medium-cost PCs). Main memory sizes are growing more slowly than both on-processor densities and DRAM densities. Main memory size is primarily driven by operating system and application working set sizes, rather than semiconductor processing technology. It is possible, assuming logic and DRAM processes remain distinct, that we will see systems with one processor and one DRAM chip (for medium-range systems, but not servers or high-end workstations) [97]. The difference in capacity between the two chips will thus be approximately the ratio of their respective sizes times the difference in the density of dense memory structures on the two chips. For example, according to Figure 5-2, a future processor in 2010, which is mostly memory, will have an eighth of the capacity of main memory, assuming that the chip areas are similar (the actual difference is likely to be more, since SIA projects that DRAM dies will be twice the size of processor dies by 2010, whereas now the processor, on average, is about 10%

larger [102]). New processes may affect the slopes of these lines considerably; we discuss the process issue in more depth in Section 5.4.

Regardless of whether support for dense on-processor memory cells arises, it is quite possible that future MOPs will contain a substantial fraction of the system memory. Conventional wisdom states that the MOP will be organized simply as a giant level-two cache [56]. In this section, we question that assumption, and discuss three types of hybrid memory systems:

- *Logical hybrids*, which combine various mechanisms from both caches and physical memories, to realize higher overall performance. We propose and evaluate one logical hybrid in Section 5.2.
- *Physical hybrids*, which use physically distinct a part of the on-chip memory as a level-two cache and a part as a fraction of main memory. We discuss physical hybrids in Section 5.3.
- *Unified hybrids*, which can treat portions of on-chip storage as either physical memory or as a cache (or both simultaneously). We discuss this type of hybrid briefly and do not evaluate it experimentally as we do the previous two.

In the next subsection, we describe a taxonomy that captures the differences between typical caches and physical memory, treating them as endpoints on a spectrum. In the rest of this chapter, we discuss logical hybrids, physical hybrids, unified hybrids, and complete processor/memory integration.

5.1 A taxonomy for memory hierarchies

Memory hierarchies exist to provide the illusion of memory that is both fast and large. While caches and physical memories perform the same function in a memory hierarchy, the two structures are optimized quite differently due to the constants involved. Physical memory has traditionally been organized to minimize disk accesses [27], since going to disk is so expensive. Physical memory is thus fully associative, replacements are handled using sophisticated software schemes, and the blocks (pages) are large to amortize the overhead of the mechanical latencies incurred upon misses. Furthermore, inclusion is often relaxed, as a page may sometimes exist in main memory but not on the disk (swap in Solaris is one example).

Caches, conversely, have traditionally been organized to provide fast access to a small set of operands. Cache lines are typically small (since early caches had few lines and cache miss penalties were small), they use bits of the address to index into the cache (for faster access), and they generally hold copies of blocks that exist at lower levels of the hierarchy (since caches have traditionally been much smaller than physical memories, the cost of the duplicated bits was small) [106].

As both the MOP capacity (absolute and relative to physical memory), and hit/miss times change qualitatively, the best design may lie in between the traditional definitions of cache and physical memory. To examine this space, we categorize a generalized level of the memory hierarchy by the following five components, and discuss the components in the context of MOPs.

- **Block size** (large or small): as MOPs grow to a larger fraction of the system memory, pollution will decrease (since there is a larger total number of blocks). In theory, coarser-grain blocks could be mapped into the MOP without hurting performance. However, as we have seen in Chapter 4, coarser-grained mappings can hurt performance for fine-grained applications. Ideally, data could be transferred *and stored* at a coarse or fine-grain, depending on the application, but mapped at a coarse grain.
- **Associativity** (low or high): long off-chip delays will make high associativities desirable, to reduce (or eliminate) the chance of mapping conflicts. Already long hit latencies may make a slight additional penalty for reduced misses palatable. Furthermore, sophisticated replacement policies could exploit the added flexibility that full associativity provides.
- **Indexing** (direct or indirect): a block may be found either by indexing into a set and doing a direct compare of the tag with one or more stored tags, or by performing a table lookup to obtain the pointer to the operand's exact location. Indirect access memories have more flexibility with respect to allocation and mapping, but at the cost of serializing the accesses. Conversely, direct access memories have limited associativity, but generally

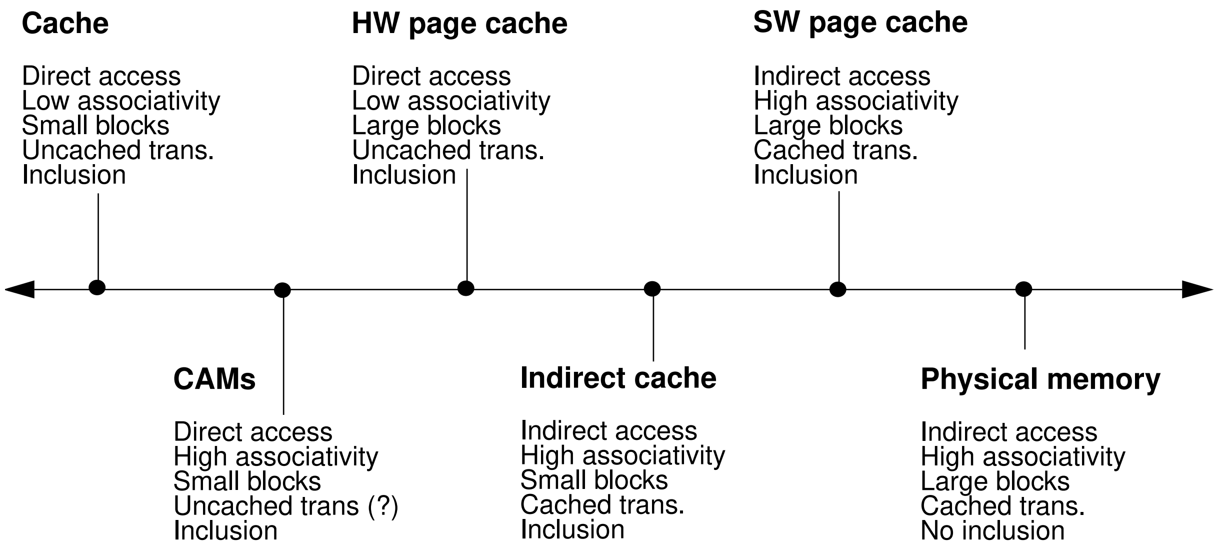


Figure 5-3: A sample of points in the taxonomy space

allow parallel accesses to tags and data. Traditional caches use direct access, but the flexibility of indirection may be superior for MOPs with long miss penalties.

- **Cached translations** (cached or uncached): caching commonly used translations (as in a TLB) could reduces the overhead of lookups for data. Cached translations are not limited to indirect access memories; designers could conceivably cache translations for a CAM (they have also proposed caching translations to speculate on which block within a set should be driven before performing the tag compare) [20].

- **Inclusion** (enforced or not): enforcing the principle of inclusion means that a given level of the hierarchy contains no data not also contained in the level below. For memories that are not substantially smaller than the level below, enforcing inclusion would prove wasteful. Enforced inclusion simplifies the control necessary to handle inter-cache communication (L1 to L2, for example) and is thus worthwhile when a large size disparity exists between two levels in the hierarchy.

In Figure 5-3, we list some logical hybrids in the space in between the extremes of traditional caches and traditional physical memories. Content-addressable memories (CAMs) implement high associativities while retaining direct access. Hardware page caches are organized much the same as traditional caches, except that they map full pages instead of smaller lines. In the indirect cache scheme (evaluated in the next subsection), small cache lines are accessed using a table lookup and TLB-like structure to provide full associativity. Finally, software page caches behave much like physical memory, except that they duplicate the pages in the cache and in physical memory [85]. This list is intended to be illustrative, not exhaustive.

5.2 A logical hybrid - the Indirect Cache

While there are many points in the taxonomy space, many of them are not good fits for technological trends. Both CAMs and traditional, direct-indexed fully associative caches are not well-suited for large on-chip structures. They either consume significant power (CAMs) or exhibit high latency if the large number of tag compares is serialized (trading off latency for power consumed and design complexity). A hardware page cache (in which the pages were accessed by indexing the tags, like conventional cache lines) would incur extra conflicts due to the restricted mapping, which would generate extra loading of the large pages and exacerbate bandwidth limitations (as evidence, the performance of a hardware page cache with 4KB blocks can be seen for ten benchmarks at the end of this section, in Figure 5-6). A software page cache may perform better than the hardware page cache if the fully associative organization resulted in fewer misses, but would still likely incur performance losses due to high traffic volumes.

We have identified one candidate for a competitive logical hybrid, which we call an *Indirect Cache Extended*, or ICE. The ICE manages an on-chip cache similar to how physical memories are managed: a hash table holds the mappings of where blocks reside in the ICE, and a tag cache holds a subset of recently referenced cache mappings (like a TLB) for fast access in the common case. The translations used to map data into the ICE are not identical to those used to map physical pages into memory; the location of blocks in the ICE are determined by the controller that manages the ICE, not the virtual or physical addresses of the block in question. To our knowledge, the first computer to re-map memory from physical store was the Atlas [74, 105], which allocated 32 pages in core memory, and took a fault when a requested datum was out of core, at which point it would load the page from drum memory, choose a victim from core with a software scheme, and perform the replacement.

The design goals of the ICE were twofold: (1) to provide full associativity, allowing policies to creatively exploit the mapping flexibility, while at the same time compensating for the extra overheads of providing high associativity (and incurring lower penalties than a CAM or direct-indexed fully associative cache), and (2) to pack data efficiently into the cache, not incurring the pollution penalty of a subblocked cache, while still being able to exploit the traffic policies we presented in Chapter 4. Efficient handling of different-sized fetches is important.

In Figure 5-3, we display the organization of the base ICE. As with physical memory, the indices into the data array are held in a table (analogous to a page table) that we call the *tag store*. For fast access, a subset of the indices are held in a *tag cache*, which is analogous to a TLB in a virtual memory system. On a tag cache hit, if the valid bit is set, the data index is used to access the data array. On a tag cache miss, the tag store, which is kept in pinned blocks in the data array, is accessed to find the requested block. If found, the entry is loaded from the tag store into the tag cache. If the tag is not found in the tag store, the system requests the block from main memory. We note that we are not performing virtual memory address translation here; the ICE uses physical tags, and the data indices are restricted to the ICE (they are not part of the virtual memory system).

The main source of overhead incurred by an ICE, which is not intrinsic to an ordinary cache, is the extra latency needed to access and manage the indexing table (the tag store). Rather than cycle through the inverted tag table (which is effectively a chained hash table) on each data array lookup, the tag cache provides a lower-latency access path for the majority of accesses. Even with the tag cache, there are still three sources of overhead. The first is the serialization of the tag cache access and the data lookup. The second is the time required to process tag cache misses; *i.e.*, to access the tag store to find the mapping (or determine that the block is not in the store). The third source of overhead is that associated with performing more complex replacement. All three result from the added flexibility provided by the ICE mechanism;

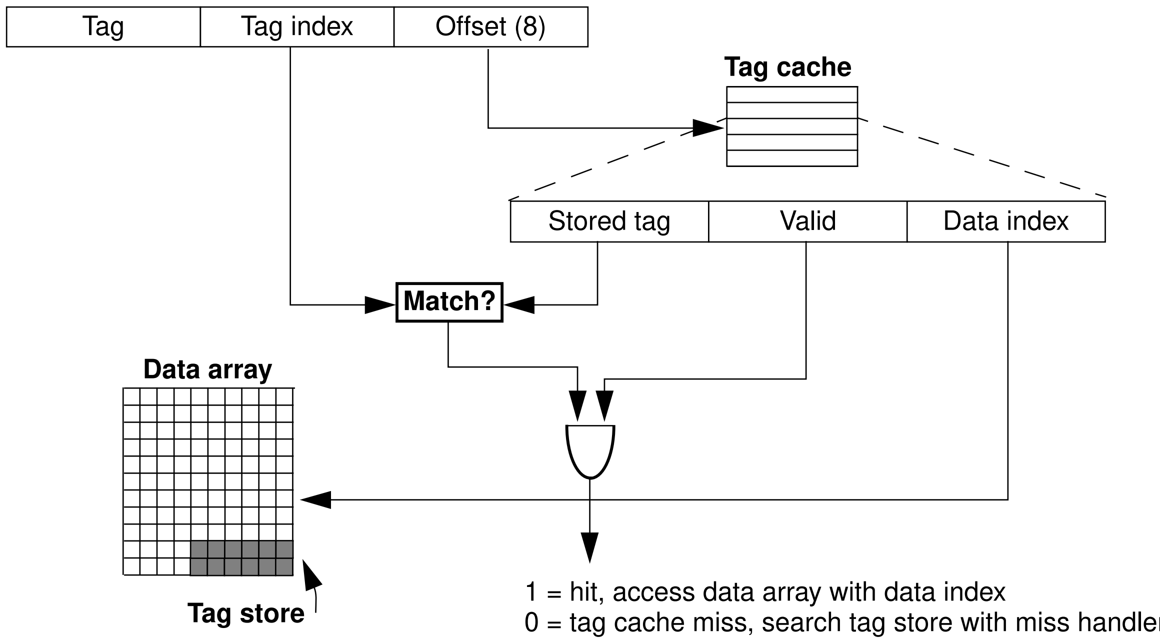


Figure 5-3: Organization of the base ICE

the challenge is to reduce these overheads sufficiently that they are overcome by the benefits of the mapping flexibility. We address each in subsections below.

5.2.1 Additional hit latency

The ICE reduces average tag store latency by keeping frequently used mappings in the tag cache. Conventional set-associative caches can generally perform the tag lookup and data lookup in parallel. However, some modern set-associative caches do the tag and data accesses serially: the Alpha 21364 contains a 1.5MB, 6-way set associative L2 cache [56], for which the tag and data accesses are processed serially, due to power and timing constraints. Such a cache would have no intrinsic access time advantage over the ICE. To compare against a cache that does do the accesses in parallel, we ran some simulations in which we increased the ICE access time by one cycle, and found that the performance impact was negligible. Finally, if there were cases where an extra cycle or two on the hit path did impact performance, it would be possible to speculate by accessing the data array in advance of obtaining the data index (based on the previous access). This is less likely to be useful for large caches, but is a possible avenue to explore.

5.2.1.1 Tag cache misses

A potentially worse source of overhead is the latency required to fill the tag cache upon a tag cache miss. In joint work, Reinhardt came up with an efficient organization to handle tag cache misses quickly. In the organization that he proposed, the tag store is organized as a hash table (similar to an inverted page table in conventional microprocessors, such as in the POWER and PowerPC architectures [65, 129]). As in the PowerPC architectures, the size of the hash table was set to be twice as large as the power of two greater than or equal to the number of physical mapped regions (physical pages in PowerPC and cache blocks in ICE). In the PowerPC architecture, each hash table entry maps to one *page table entry group* (PTEG), which holds 8 mappings that are searched linearly for a match. If the match fails, a secondary hash function generates a different address, which searches a second PTEG. If a match is not found in the second PTEG, the page is not in physical memory and a page fault occurs.

The ICE implementation assumed a similar model, but searched adjacent entries in the hash table instead of grouping multiple entries into a single PTEG. ICE also used hardware to accelerate the hash table search. To reduce the latency for resolution of misses, the ICE implementation had multiple comparators placed by the read-out rows of the memory banks holding the tag store. Upon a tag cache miss, the appropriate hash table entry for the given tag is read out, with the rest of its row in the memory bank. The comparators search for the tag in both the indicated hash table entry plus the adjacent entries in the row, thus scanning several possible locations of the tag simultaneously (in addition to the PowerPC, this solution bears some resemblance to clustered hash tables [121]). We depict a diagram of Reinhardt's scheme in Figure 5-4, showing how the tag is hashed to get the hash table index, which is then accessed and read out (the whole row) to multiple comparators, looking for tag matches.

We implemented the proposed organization, and ran simulations to compare the performance impact of a perfect tag cache (which never misses) to a finite tag cache. We set the capacity of the finite tag cache to be smaller than the size of the tag array needed for a comparable, traditional cache (4-way set associative, 1MB L2 with 256B blocks). The tag cache we used was a 4-way associative tag cache with 2K entries (each of which maps a 256B block in the data array). The hash table held twice as many entries as needed to map the blocks into the

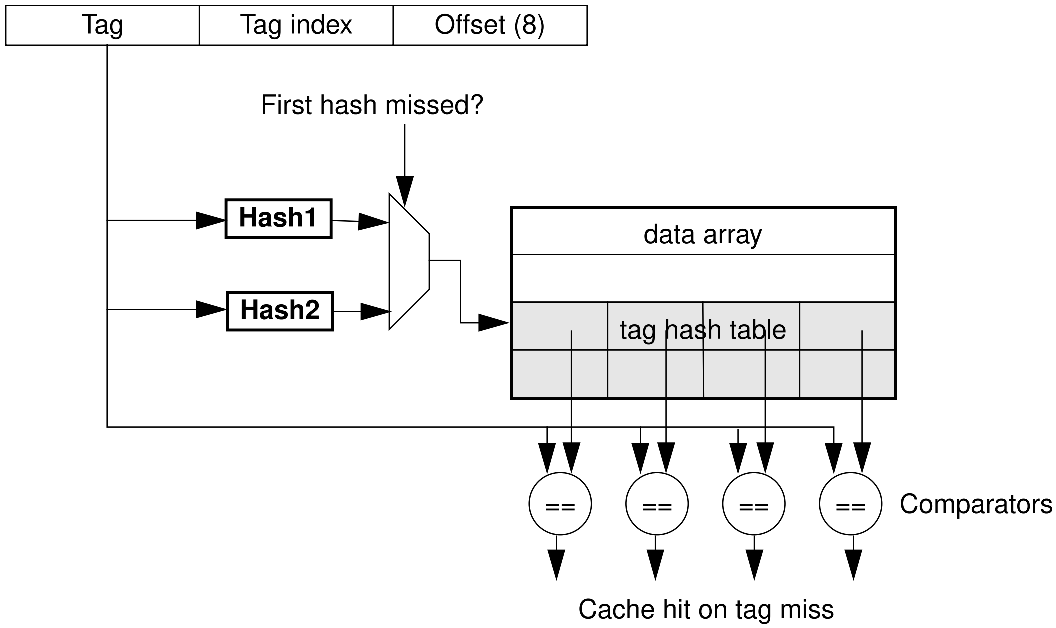


Figure 5-4: Accelerating tag cache misses

cache. Since the tag cache is smaller than the conventional tag array, and the hash table takes up what would otherwise be data blocks in the data array, the *gross cache size* [61] of the ICE, is less than the gross cache size of the conventional cache.

We show the results of this comparison in Table 5-1, in which we list the IPCs of ICEs with perfect and imperfect tag caches. In this table, we normalized the performance numbers to the performance of a comparable cache (1MB, 4-way set associative L2 cache with 256B blocks). The worst performance losses are 3.3% and 3.1% degradations (compress and applu), but the others are much smaller. Two of the benchmarks (apsi and wave5) even show improved performance with the imperfect tag cache, because of reduced cache thrashing in the level one data cache. In all simulation results that we present in this section, we assume that 256 blocks (64KB) of the data array were allocated to hold the tag store. Those blocks were not used to hold data, and were thus factored into ICE performance.

5.2.1.2 Complex replacement

The third source of overhead is handling more complex replacement when the cache is highly associative and managed by software. In our experiments, we assumed a hardware con-

Benchmark	gcc	compress	perl	vortex	tomcatv	swim	su2cor
IPC (perf. tag cache)	1.007	0.892	1.007	1.032	0.962	1.003	1.008
IPC (real)	0.999	0.862	1.001	1.022	0.954	0.992	0.998
Benchmark	**hydro2d**	**mgrid**	**applu**	**turb3d**	**apsi**	**wave5**	
IPC (perf. tag cache)	0.993	0.995	1.020	1.015	1.026	0.996	
IPC (real)	0.973	0.977	0.988	1.015	1.033	1.000	

Table 5-1: Performance impact of an imperfect tag cache (1MB ICE)

troller that was tightly coupled to the cache, which was designed and implemented by Reinhardt [16]. He also proposed the replacement policy that we used to evaluate the ICE, a low-overhead policy called *generational replacement*, which is a frequency-based policy that groups blocks into one of several prioritized “bins”. This policy—which was designed to counter the effect of “filtering” that the L1 caches do on the reference stream reaching the L2—is described in considerable more detail elsewhere [16]. In Table 5-2, we show the relative number of misses that the ICE incurs with generational replacement, as opposed to a 4-way set associative cache (with LRU replacement) of the same size. The number of misses is only slightly lower on average than the baseline, so while the generational replacement algorithm is competitive, it is not a source of high performance gains in the results we show later in this section. (For our simulations, we assume that the policy code and data structures are pinned in the data array, and that the replacement handlers run while a miss is being serviced [82]). We also assume the replacement handler has enough bandwidth to handle multiple simultaneous outstanding misses before they return.

To increase the coverage of the space we can map in the tag store, we evaluated the use of subblocked tags, analogous to the complete subblocking of the TLB proposed by Talluri and Hill [120]. In our complete design, each tag maps 4KB instead of the 256B as described above, with 16-way complete subblocking within the tag. Thus each subtag has its own data index and valid bit, and each tag maps sixteen 256B blocks in the data array. We limited the tag cache size to be smaller than the equivalent tag store size for our baseline (1MB, 4-way 256B block) cache. This bound resulted in a 4-way associative tag cache with 512 entries. We can thus cover 2MB with the tag cache, but since the number of tags is reduced to only 512 entries, there is a resultant increase in tag cache misses. In Table 5-3, we display the effects

Benchmark	gcc	compress	perl	vortex	tomcatv	swim	su2cor
Normalized misses	0.917	1.364	0.983	0.778	1.075	1.007	0.963
Benchmark	**hydro2d**	**mgrid**	**applu**	**turb3d**	**apsi**	**wave5**	
Normalized misses	1.023	1.010	0.990	0.875	0.728	0.978	

Table 5-2: Relative misses for the ICE (compared to 1MB, 4-way set associative LRU)

Benchmark	gcc	compress	perl	vortex	tomcatv	swim	su2cor
Normalized misses	1.024	1.185	1.038	0.964	1.073	1.012	1.070
Normalized IPC	0.997	0.966	1.012	1.007	0.970	1.027	0.972
Benchmark	**hydro2d**	**mgrid**	**applu**	**turb3d**	**apsi**	**wave5**	
Normalized misses	1.064	1.043	0.999	0.869	0.779	0.971	
Normalized IPC	0.984	0.973	0.953	1.020	1.033	1.044	

Table 5-3: Performance impact of 16-way subblocked tags)

that the subblocked tag cache has on (a) the number of misses, and (b) performance measured in IPC. Both metrics are normalized to those of the base ICE with non-subblocked tags. The results show small increases and decreases in both performance and misses: at worst an 18% increase in tag cache misses (compress), and at best, a 22% reduction in tag cache misses (apsi). While the subblocked tags do not provide across the board performance increases, they do permit us to combine the ICE with the traffic policies from Chapter 4, as we describe in the next subsection.

5.2.2 Coherence issues

The ICE uses physical addresses to index into the tag cache and hash table, since in our simulations the primary caches are virtually indexed and physically tagged. Thus, the ICE is still capable of snooping on a bus, and transactions that are snooped from a bus may still be examined in the tag cache, as quickly as a conventional cache would examine them in the tag array. The tag cache may also be duplicated to provide extra bandwidth for snooping. The ICE will incur extra overhead when snooped transactions cause tag cache misses, which are likely to be more frequent than those caused by the reference stream from the local processor. It is unlikely that tag cache misses caused by snooping should cause a tag cache fill, although a small, separate buffer to cache snooped translations may reduce the overhead of tag cache misses for blocks with certain types of access patterns (such as migratory sharing).

5.2.3 Performance analysis

In this section we evaluate both the ICE and the ICE combined with the optimizing traffic policies. We show that DSF and SFP are compatible with the ICE, since they will not suffer from the performance penalty of having a subblocked data cache. Since the ICE tags can be subblocked, the traffic optimization policies may still operate on transfer blocks, but the transfer blocks are now packed more efficiently into the cache. Also, since the ICE demonstrates a lower miss ratio due to full associativity and generational replacement, the Rambus channels are more often free, and the bus prioritization has more opportunity to bring data across the channel speculatively, while keeping latency for critical requests low.

In Figure 5-5, we plot the performance (in instructions per cycle) of a 1MB ICE for ten SPEC95 benchmarks. Our simulation parameters (processor core, L1 caches, physical memory, buses) are identical to those described in Section 4.1. For each benchmark, we compare four experiments, shown by the four-bar clusters in Figure 5-5. From left to right, the first three bars in each cluster represent the base ICE, the subblocked tag ICE, subblocked tag ICE with DSF, SFP, and bus prioritization (which we will call ICE++). The fourth bar represents the performance of our baseline system with a traditional, 1MB, 4-way set-associative L2 cache for which the block size is set at the performance point on a per-application benchmark (*i.e.*, the best block size is chosen for each benchmark).

The figure shows that, as also shown in Section 5.2.1.2, the subblocked tags have little effect on ICE performance, causing two slight improvements (compress, swim, and wave5) and two minor degradations in performance (vortex and applu). The addition of the traffic optimization policies, however, makes a large difference for several benchmarks (mgrid, hydro2d, and applu), slight improvements for several others, and two minor performance drops for perl and vortex (we hypothesize that the additional latency required to complete prefetch transactions when a demand fetch arrives is responsible for this drop).

The most significant result in this figure is the fact that ICE++ nearly equals or exceeds the performance of the performance-point cache in every case but one (compress). The ICE++

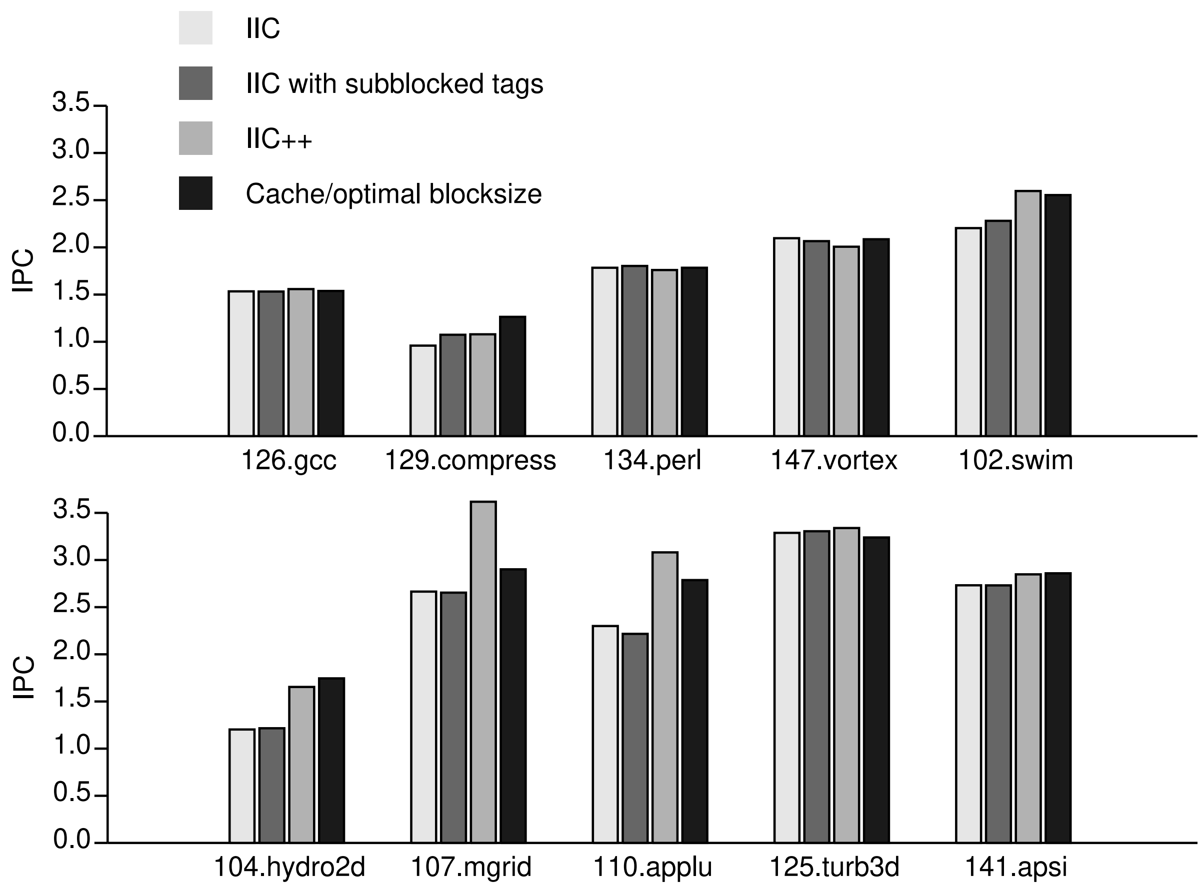


Figure 5-5: Performance of an ICE with traffic optimization schemes

performance of seven of the other benchmarks is extremely close to the performance-point cache (apsi, swim, turb3d, hydro2d, vortex, gcc, apsi) or significantly better (mgrid, applu). We computed the speedup of the ICE++ over the cache at the performance point for each benchmark, and found that the mean speedup of ICE+ over traditional caches with benchmark-specific block sizes is under 1.6%. This result is significant because it indicates that, on average, ICE++ performs better than any cache, no matter the block size (for the benchmarks we studied).

That result does not show how ICE++ would fare against traditional caches when their block size was fixed across all applications. In Table 5-4, we list the mean speedup (a ratio of IPC measurements) that ICE++ obtained across all benchmarks (those listed in Figure 5-5) over cache with a fixed block size. For example, in the first column of Table 5-4, the number repre-

Block size	64	128	256	512	1024	2048	4096
Mean speedup	0.27	0.16	0.09	0.08	0.12	0.18	0.28

Table 5-4: Mean speedup (across SPEC95) of ICE++ over 1MB, 4-way set assoc. caches

sents the mean speedup that ICE++ showed over all our SPEC95 benchmarks running on 4-way set associative, 1MB L2 caches with 64-byte blocks. The mean speedups range from a low of 0.08 (512-byte blocks) to a high of 0.28 (4KB blocks). In Figure 5-6, we plot the performance of traditional caches in IPC—assuming the same simulation parameters as used elsewhere in this chapter—as a function of block size on the x-axis. We assume that the traditional caches are 1MB and 4-way set associative. Each line represents the IPC for one benchmark as the L2 block size is increased. The individual points represent the IPC for the ICE++. We placed each ICE point on the x-axis at the performance point for that benchmark; each point will appear at the same position on the x-axis where the blocksize curve for that benchmark peaks. Note that the ICE uses a constant block size, and is thus invariant on the x-axis; they are placed at different x -coordinates for illustrative purposes.

Each ICE++ point is simply a heavier or dark-filled version of the mark used in the line for a given application. This graph illustrates our earlier claim: the ICE++ performance for each benchmark is close to (or above) the peak of the traditional cache curve for each benchmark. At the 64-byte point on the x-axis, the only ICE point is the inverted triangle, at just over 1.05 IPC (and somewhat under the corresponding point for compress, at 1.2 IPC). The ICE points for perl and vortex are superimposed at 128-byte blocks (at about 2.0 IPC). At 256-byte blocks is the turb3d point. The gcc point is at 512 bytes, the swim point is at 1KB, the applu, hydro2d, and applu points are at 2KB, and finally the mgrid point is at 4KB. Most are above the peaks of their corresponding curves; the exceptions are compress, vortex, and hydro2d, which are all reasonably close to the peaks of their application performance curves.

These results show that the performance of ICE++ is *stable*: it shows significant improvements in actual performance when compared against any specific block size. When this result is coupled with the previous result—that ICE++ significantly outperforms the best conventional cache for several of our benchmarks—it shows that ICE++ offers both high perfor-

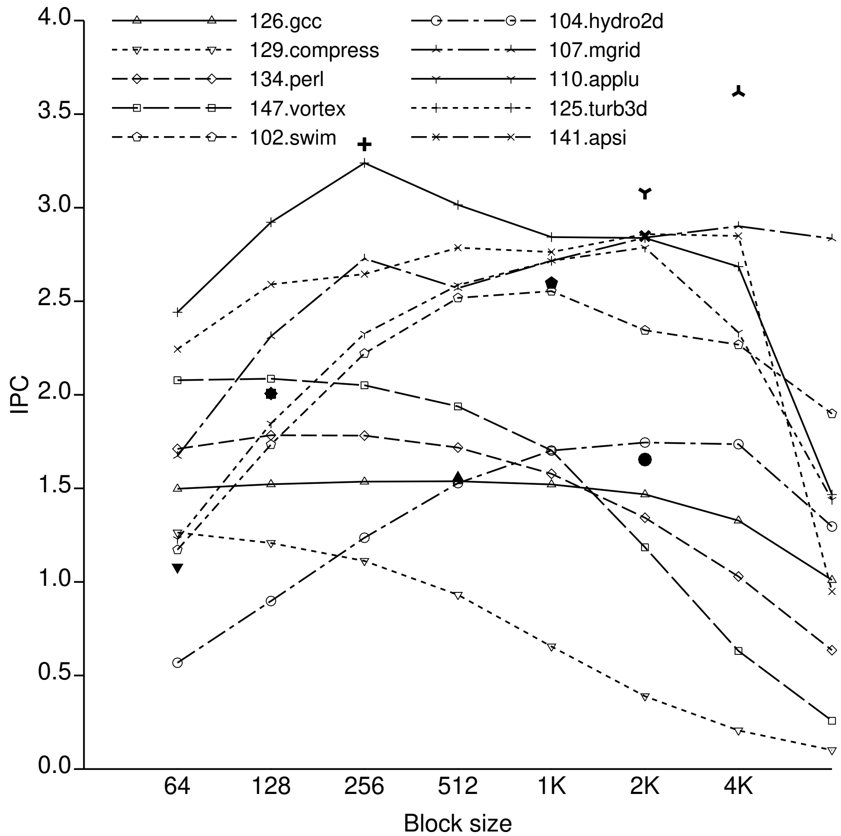


Figure 5-6: Comparing ICE++ to traditional caches

mance and high stability across varied workloads. This performance stability and improvement comes from the synergy of the traffic policies and the base ICE; neither of which does nearly as well individually.

5.3 Physical hybrids

At the beginning of this chapter, we described how MOPs may grow to encompass a substantial fraction of main memory, particularly if support for denser memory cells becomes incorporated into logic manufacturing processes. Should this scenario arise, with MOPs holding a significant fraction of the total system memory, there will be three desirable characteristics for the MOPs:

1. MOPs should not enforce inclusion, since the total system memory could be increased significantly if inclusion were not enforced (inclusion simplifies caching policies for a strict

hierarchy, which would no longer be as applicable if a logical level of the hierarchy was divided into on- and off-chip banks).

2. MOPs should still minimize the off-chip accesses, which will be considerably more expensive than accesses to the memory on-chip.
3. MOPs should allow for fine-grain off-chip accesses; loading a page at a time, for example, will cause poor performance for applications that show little spatial locality (for example, the SPEC95 integer codes in Figure 5-6).

In this section, we perform a brief evaluation of *physical hybrids*, in which a MOP is divided into two physically distinct structures. One of the structures is an on-chip extension of physical memory, and the other is an L2 cache for off-chip data (analogous to the RAC in the Stanford DASH [83], albeit in a uniprocessor context).

We measured the miss rates of our benchmarks running on five different simulated organizations: (1) all on-chip memory is a fast fraction of physical memory, (2, 3, 4) three physical hybrids, in which 1/2, 1/4, and 1/8 of the on-chip memory are a cache for off-chip data, respectively, with the remainder of the on-chip storage in each case going to physical memory, and (5) all on-chip memory is a cache, and all physical memory is off-chip. For any portion of the chip devoted to physical memory, we increased the simulated capacity by 20% to compensate for the fact that physical memory would incur smaller area overhead than a cache (smaller tag overhead, no comparators, etc.) The actual overhead for cache support is non-linear with respect to cache size. We intend the 20% to be a crude first-order approximation that should be refined in subsequent studies, in which specific implementations are evaluated. We selected the pages for the on-chip fraction of physical memory by profiling them, and mapping those pages that had the highest total static reference counts to the on-chip memory. We chose a block size of 256 bytes for the cache portions of the MOP, consistent with the earlier experiments in this chapter, but assumed a direct-mapped cache due to the large size of these caches [59].

In Table 5-5, we list the global miss rate for the data segment (number of misses divided by the total number of references) for each organization. For each benchmark, we present results assuming MOPs with capacities equal to 1/2, 1/8, and 1/32 of the data set size. The “all phys-

Benchmark/ % data set	Fraction of on-chip cache				
gcc	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0002	0.0002	0.0003	0.0006	0.0063
1/8	0.0017	0.0002	0.0025	0.0038	0.1522
1/32	0.0051	-----	X	X	0.4813
perl	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0003	0.0003	0.0005	0.0006	0.0064
1/8	0.0006	0.0007	0.0009	0.0011	0.0226
1/32	0.0012	0.0013	0.0018	0.0021	0.0353
vortex	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0002	0.0002	0.0002	0.0003	0.0017
1/8	0.0010	0.0005	0.0009	0.0017	0.0060
1/32	0.0028	0.0025	0.0040	0.0058	0.0213
swim	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0026	0.0025	0.0024	0.0024	0.0687
1/8	0.0035	0.0029	0.0031	0.0036	0.1279
1/32	0.0044	0.0039	0.0047	X	0.1497
su2cor	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0002	0.0001	0.0002	0.0002	0.0074
1/8	0.0010	0.0007	0.0010	0.0021	0.0497
1/32	0.0024	0.0033	0.0039	X	0.1610
applu	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0044	0.0043	0.0044	0.0044	0.0432
1/8	0.0053	0.0052	0.0052	0.0059	0.0987
1/32	0.0061	0.0063	0.0071	X	0.1353
turb3d	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0008	0.0009	0.0008	0.0009	0.1122
1/8	0.0019	0.0017	0.0018	0.0026	0.3509
1/32	0.0029	0.0030	0.0037	0.0054	0.4111
wave5	**All**	**1/2**	**1/4**	**1/8**	**None**
1/2	0.0007	0.0006	0.0006	0.0007	0.0264
1/8	0.0024	0.0010	0.0013	0.0019	0.0926
1/32	0.0040	0.0026	0.0050	0.0080	0.1933

Table 5-5: Global miss rates for physical hybrid experiments

ical memory" experiment performs quite badly, since there is no buffering of off-chip data. For a MOP at 1/32 of the data set size, the ratio of off-chip accesses to total accesses is as high as 0.41 (turb3d) and 0.48 (gcc). However, the combined physical memory/cache experiments exhibit miss ratios comparable to those of the all-cache experiments. Even when only 1/8 of the MOP area is devoted to a cache for off-chip data, the miss rates are comparable to all-cache. When 1/4 of the MOP is devoted to a cache, the total off-chip miss rate is equal to or better than "all-cache" in fully half of the measured cases.

The physical hybrids thus have the potential to provide competitive performance at a lower cost if MOPs grow to be a sizable fraction of the total system memory. The caveat is that we used profiling to choose which pages to map on-chip. An interesting research question is whether heuristics that infrequently promote pages to be on-chip (or demote them) based on dynamic usage patterns (similar to reactive NUMA [36]) could approach or even outperform the static, profiled mapping of pages.

5.4 Processor/memory integration

As on-chip storage capacity grows, and system integration on the processor die increases, the possibility exists that all physical memory will eventually end up on the processor die, with processor interfaces connecting only to I/O. In Figure 5-2, we track the trend in processor capacity versus DRAM capacity, and show that they are slowly converging. Extrapolated sufficiently far, one might assume that complete processor/memory integration was likely. However, the 1997 SIA Roadmap [102] project that the capacities of processors and DRAM chips will diverge quickly for two reasons. First, SIA projects that DRAM chips, with areas currently slightly smaller than processor chips (10%), will grow to be twice as big by 2012. Second (and more important), the density differential between packed logic-process SRAM cells and DRAM cells is projected to grow rapidly. Current estimates range from a factor of 21 [40] to 25 [102]. SIA projects density differentials of *73* by 2009 and *94* by 2012.

For full integration to occur (or even a substantial fraction of the memory residing on-chip, as discussed in Section 5.3), the manufacturing process used to make processors will need to incorporate support for dense cells (thin gate oxides, support for 3-D stacked or trench capacitors, and multiple layers of polysilicon). Conversely, processors could start to be manufactured in a more DRAM-like process, with support for fast gates (and more levels of metal wiring) added. While such hybrid processes may be used for embedded systems at the low end (for which great cell density or gate speed may not be needed), for high-end processors they must either offer a performance potential commensurate with the cost of developing the new

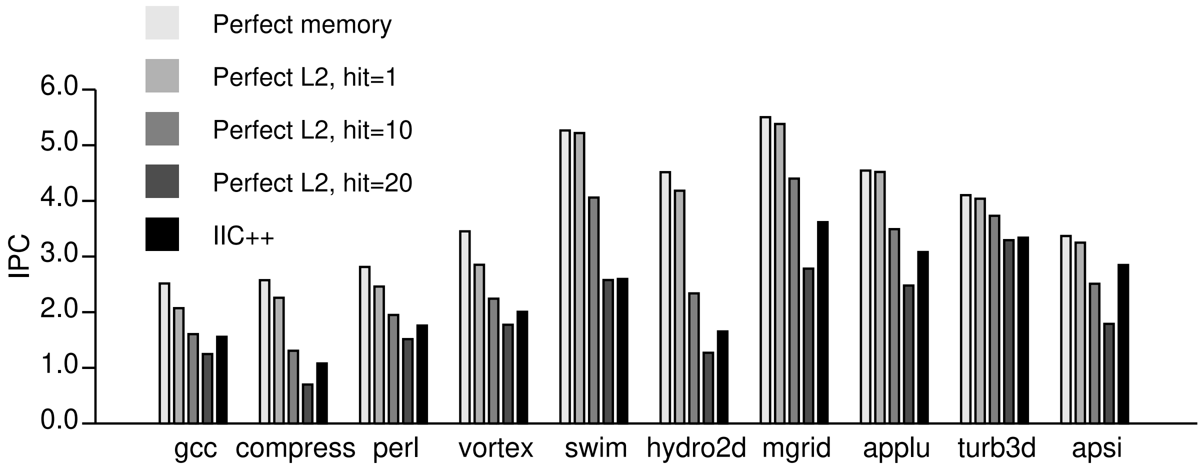


Figure 5-7: Performance of perfect L2 caches

process, or be developed in a different market and then move over to high-end processor design once the development costs have been recouped.

We performed a limited set of experiments to evaluate the potential (at least for our experimental setup) of having all of the physical memory on-chip. In Figure 5-7, we display the performance, measured in IPC, for five different experiments per benchmark, represented by the five bars in each cluster. The left-most four assume various ideal memories. The first represents a perfect memory system (all accesses return in one cycle). The second, third, and fourth assume a system with the same processor and L1 caches described earlier in this chapter, but with the all physical memory in place instead of the L2 (we call this "perfect L2"). These three caches effectively never miss (i.e., all of the physical memory is on-chip where the L2 would be), but have hit latencies of 1 cycle, 10 cycles, and 20 cycles, respectively. The right-most bar, representing the fifth experiment, shows the performance of the ICE++ organization described in Section 5.2.

We see that there is not a large performance differential between the ICE++ runs and an integrated system with the same access time (10 cycles) as the ICE. On average, this performance differential is 13.3%. In one case (apsi), we again see the effect where longer delays for some operations (L2 misses) result in less L1 thrashing, a lower L1 miss rate, and thus better performance for the ICE than for an ideal L2. Only one benchmark (swim) shows a large performance gap between an ideal, 10-cycle on-chip memory and the ICE. If the time required to access the denser store increases, due to heavier banking or larger data arrays, the ICE may actually outperform an integrated system; the ICE (with 10-cycle access) outperforms an ideal L2 with a 20-cycle access penalty in *every case*.

Our result of a mean 13.3% performance gap is consistent with the results reported by the Berkeley IRAM group, in which they showed a negligible performance improvement for a large IRAM chip (a mean of 4%) and a result comparable to ours (16%) for a small IRAM chip, over a conventional alternative [42]. These results indicate that (for these benchmarks and target system assumptions) there is not a sufficient performance gain to justify any costly process support. Different applications may result in different conclusions, of course. Given the large real estate that will be available on future chips, processor designers will likely be able to implement as many processors on-chip as makes sense, either for increasing the performance of a specific job, or for throughput-oriented processing. In either case, on-chip buffering and off-chip bandwidth are likely to be the resources that limit how many cores may be placed on a chip and used effectively. If that were the case, modifying the process to support denser memory cells on-chip—and thus higher off-chip effective bandwidths—would probably be worthwhile.

In the nearer term, however, the projected disparity between the two technologies—coupled with the dropping number of DRAM chips in (uniprocessor) systems—makes the scenario in which the system consists of two main chips likely. One chip will be the processor, optimized for speed and throughput, and the other will be optimized for density. The two will likely be closely coupled, perhaps in a single package or in a multi-chip module. This package would offer both dense storage and fast processing, and could be used as a building block for larger systems.

If all of the DRAM for a small system is packaged closely to a processor, the question arises as to how more memory should be added, and/or how the system can be extended using that single package as a building block. Also, if the DRAM moves onto the processor die because of numerous processors on the main processor die, how those multiple units can be used to accelerate a single application is an important question. In the next chapter, we propose a class of architectures called memory-centric architectures, that address the two questions posed above: how to transparently run codes on systems with multiple processing units, each of which is closely coupled to some local memory.

Chapter 6

Memory-Centric Architectures

In Chapter 1, we discussed how both memory hierarchies and distributed processing could increase the width of the processor/memory interface cost-effectively. In the previous two chapters, we described techniques to improve PMI performance in a traditional memory hierarchy. In this chapter, we describe a class of architectures, called *memory-centric architectures*, which provide a PMI that is both distributed and transparent to the programmer and compiler.

A large number of distributed PMI architectures (which includes all parallel processors) have been built in the past. Traditional parallel processors, whether shared-memory or message-passing machines, were primarily proposed and/or built not to improve performance across the PMI, but because codes needed more functional units than could be cost-effectively provided in one chip. When computational capability was the system bottleneck, the use of multiple inexpensive, commodity processors was the best way to improve performance, so long as parallel binaries were available. Future architectures will face a different problem as discussed in Chapter 1: not the challenge of providing enough functional unit throughput on a given chip, but the dual challenges of building architectures that can move the data across the PMI at a sufficiently high rate, and finding ways to map the computation onto these architectures.

The current model of a centralized PMI will allow performance to scale acceptably so long as two conditions hold: first, that the processing core has sufficient work to do (ILP or perhaps other lightweight threads) to tolerate cache miss latencies, and second, that the processor has enough bandwidth to load changes to the cache working set without excessive queueing. Both of these conditions are growing more difficult to meet. Cache miss latencies are growing, making it harder for the processor to find enough work to tolerate those latencies. The latencies are growing for two reasons: first, because of increasing DRAM access latencies (which is a market-driven problem that is solvable in the long term), and second, because of growing relative delays with smaller wires [86], which is a less tractable problem in the long term. As we have discussed extensively in this dissertation, off-chip bandwidth is difficult to scale with on-chip performance.

One class of architectures that can mitigate these two problems is memory-centric architectures, in which processing power is distributed uniformly into physical memory, and the local availability of data is what individual processors use to drive decisions dynamically. As we show in this chapter, this class of solutions can reduce both access latencies and the bandwidth required for a balanced system. For memory-centric architectures to be commercially feasible, two conditions must hold. First, processing capability must be inexpensive; the system cost must be dominated by communication and storage costs. Second, communication from one part of physical memory to another must be slow (if physical memory is partitioned into regions, inter-region communication must be slower than intra-region communication). Both of those conditions are becoming more true; the fraction of the CPU die devoted to actual computation is shrinking rapidly (see Figure 1-5), and wire delays will eventually make communication latency proportional to intra-chip distance. The processor of 2010 will have die area sufficient to hold 300 Pentiums, making processors (as implemented today) effectively free.

These memory-centric architectures may be implemented at several levels of granularity: *intra-chip* (small processors strewn along the copious storage on billion-transistor architectures), *inter-chip* (distributing physical memory among multiple chips, and placing a processor on each chip, effectively making them IRAM chips), and *inter-box* (building clusters of workstations, each of which contains a processor and a fraction of the total system physical memory, connected by a local bus that is faster than the inter-processor interconnect). In this dissertation, we evaluate memory-centric architectures at the inter-chip (IRAM) level, but do not imply that the other foci could not be viable candidates as well.

In this chapter, we first describe our historical inspiration for this class of architectures, the Massive Memory Machine, in Section 6.1. We then describe the DataScalar architecture, an asynchronous derivative of the MMM, in Section 6.2. We present the results of our DataScalar performance analysis in Section 6.3.[1]

6.1 The Massive Memory Machine

The DataScalar work was inspired by the Massive Memory Machine, and is effectively an asynchronous version of the MMM, updated to work with modern processors and communication topologies. The MMM was a synchronous, SISD architecture that connected a number of minicomputers with a global broadcast bus [45]. Each computer contained a large memory (for the time), which was some fraction of the total program memory. Each operand in memory was thus *owned* by only one processor (*i.e.*, each processor resided in the physical memory of only one processor). All computers ran the same program in lock-step, and the owner of each operand broadcast it on the global bus when accessed.

6.1.1 Operation of the MMM

This broadcast model was called *ESP* (which actually does stand for "extra sensory perception") in the MMM work. We depict an example of synchronous ESP in Figure 6-1. One processor (the *lead* processor) executes slightly ahead of the others while it is broadcasting (initially processor 3 in Figure 6-1). When the program execution accesses an operand that the lead processor does not own, a *lead change* occurs. All processors stall until the new lead processor catches up and broadcasts its operand. In Figure 6-1, a lead change occurs at cycle seven, when processor 2 begins broadcasting w_5.

The MMM supported two classes of physical memory, to which we shall refer as *replicated* and *communicated*. Replicated memory is duplicated at every node, with identical contents. Communicated memory is owned by one node only; there are no copies of communicated

1. Most of the exposition and all of the results in this chapter were taken direcly from previously published work [15].

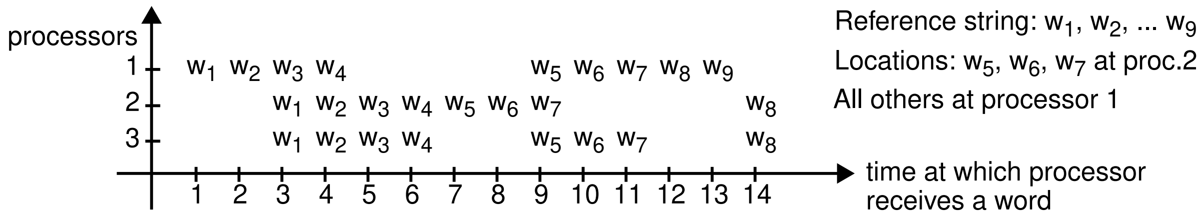


Figure 6-1: Operation of the ESP Massive Memory Machine (from [45])

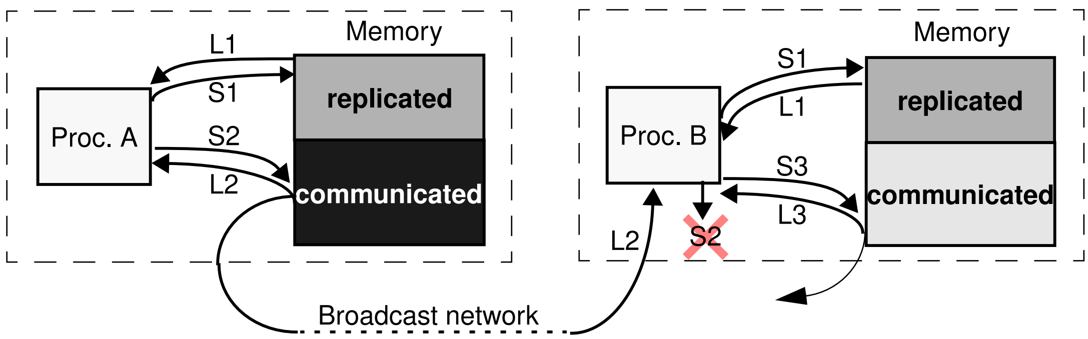


Figure 6-2: Replicated vs. communicated memory

locations at other processors. In Figure 6-2, we depict the set of memory operations possible in ESP. At processor B, there are three sets of loads and stores. The first load/store pair is to locations in replicated memory. The second load/store pair is to remote communicated locations, owned by processor A. The third load/store pair is to locations in communicated memory that is owned by processor B. Load 1 is serviced locally by both processor A and processor B, as they both have copies. Store 1 completes on both processors, overwriting their respective local copies. Load 2, owned by processor A, is broadcast and received by processor B without B issuing a request. When processor B issues store 2, it discards the store without completing it, since processor A generates the same store value, and overwrites the only copy with that correct value. When processor B issues load 3, it consumes it and also broadcasts it on the network, since B is the owner and the operand resides in communicated memory. When processor B issues store 3, it overwrites its local copy without sending it to other processors, since it is the owner and has the only copy.

The ESP execution model has several advantages over a conventional execution model. First, since all communication is one-way, no requests ever need to be sent, which reduces access latency. Second, writes never appear on the global bus, which may reduce bus traffic (since all processors are running the same program, they all generate the same store values, which need complete only on the owning processor). Third, since the MMM is fully synchronous, and all processors generate the address for each successive operand, no addresses need to be sent with the data on the global bus.

6.1.2 Limitations of the MMM

The Massive Memory Machine may have been an interesting idea for its time, but its heavy reliance on synchronous behavior renders it conceptually incompatible with today's systems. The limitations of systems even at that time were such that it would show little, if any, performance improvements over conventional alternatives [52]. In fact, Jim Goodman wrote the following after visiting Princeton for a site review in December of 1984:

> The article and discussions with the authors did not convince me that the novel ESP architecture is worth further study. In particular, I see little point in the project to simulate ESP with microprocessors.

The DataScalar architecture, described in the next subsection, addresses the limitations of synchronous ESP, as well as solving the problems associated with running the ESP execution model on modern processors.

6.2 DataScalar Architectures

The DataScalar architecture benefits from asynchronous ESP because consecutive dependent memory operands at a processor may be processed quickly. Dependence chains local to a processor will be traversed at local speeds and broadcast to participating nodes, independent of on which processor the chains reside. Ideally, each processor handles local dependence chains simultaneously, moving the entire computation ahead at a faster rate. We call a segment of a dependence chain local to one processor a *datathread*.

In this subsection, we describe the benefits associated with the base DataScalar model (ESP and datathreading) in detail. We describe how ESP reduces off-chip traffic, and we show how datathreading offers the potential for reductions in memory latency. Later in this chapter, we present simulation results that quantify each of these benefits.

6.2.1 Asynchronous ESP (traffic reduction)

DataScalar systems enjoy, and extend, the benefits of ESP that MMM obtained. ESP reduces traffic—thereby increasing effective bandwidth—by eliminating both request traffic and write traffic from the global interconnect. ESP, asynchronous or otherwise, does not further reduce the number of read operands that must be communicated off-chip over that of a conventional architecture.

ESP-based systems eliminate request traffic because ESP uses a *response-only* (or *data-pushing*) model. Since all processors run the same program, if one processor issues a load to an address, all the other processors will eventually issue that same load. The owner is therefore assured that when it broadcasts the load, all other processors will consume it. Conversely, when a processor issues a load to a datum that it does not own, it can buffer the request on-chip, and the matching data will eventually arrive. Thus, requests need never be sent off-chip. Similarly, when a store is generated at all nodes, only the owner of that address need complete the store on-chip. Since every chip is generating the value locally, created store values never need be sent off-chip. All processors will complete the store if the address is a replicated location. If the address is cached at all nodes, the store will complete in the cache, and the eventual write-back (or write-through) operation will be dropped at nodes that do not own that address. Note that there are none of the traditional cache consistency issues, since every processor is running the same program.

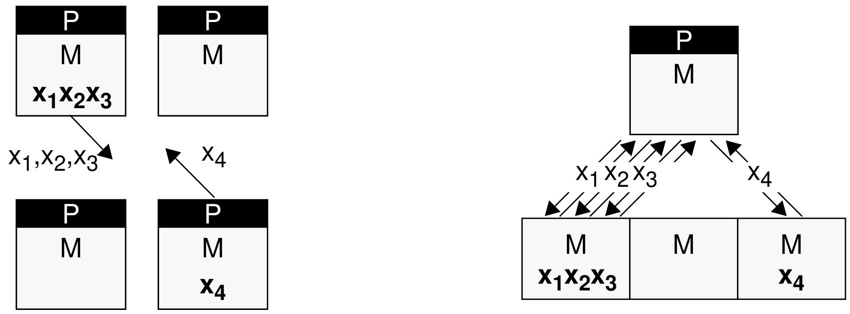
In a synchronous implementation of ESP, tags need not be broadcast with data—every processor is generating the same instruction stream in the same order, so tags can be inferred from the order in which the broadcasts are received. DataScalar systems do not enjoy this benefit; the out-of-order issue processors will all issue multiple broadcasts in an unpredictable order. In addition, more than one processor generally will be attempting to broadcast at any given time. This lack of predictability means that data must be broadcast along with their addresses and/or some other identifying tags (multiple instances of the same address may require supplementary tag information, such as a sequence number).

6.2.2 Datathreading (latency reduction)

ESP-based systems reduce memory latency by making all off-chip communications one-way only. These savings might be large if the remote communication time dominates the memory request latency, or small if the memory access latency and/or memory system queueing delays dominate the request latency.

ESP-based systems offer the potential for further reductions in memory access latencies, however. Consider a stream of accesses to memory locations, each address of which is dependent on the value of the previous address (e.g., pointer chasing). When two or more dependent addresses reside in one processor's local memory, that processor may fetch those values without incurring any off-chip latencies. Those values may then be sent to the other processors by pipelining the broadcasts, incurring only one off-chip delay on the critical path. All processors thus complete the processing of those addresses faster than would a traditional system.

To illustrate this concept, we depict a simple example in Figure 6-3a shows a four-chip DataScalar system in which each MOP contains a quarter of the program's physical memory. Figure 6-3b shows a more traditional organization, in which one MOP holds a quarter of the program's memory and traditional DRAM chips hold the other three-quarters. In both systems, operands x_1, x_2, x_3 all reside on one chip, and operand x_4 resides on a different chip. The address of each x_{i+1} is dependent on x_i. One processor in the DataScalar system can access the first three without a single off-chip access, and then pipeline the broadcasts of those three operands to the other nodes (the broadcasts will be separated by the memory access time, of course). There will be a serialized off-chip access between x_3 and x_4 (analogous to a lead change in the MMM), and then x_4 will be broadcast. The system thus incurs two serialized off-chip delays. The traditional system, conversely, incurs two serialized off-chip accesses (one request, one response) for each operand, for a total of eight in this example The traditional system would incur zero off-chip delays if all the operands happened to reside in the on-chip quarter of the memory, as opposed to a minimum of one for a DataScalar system.



(a) DataScalar system:
Pipelined broadcasts
Serialized off-chip accesses: ***2***

(b) Traditional system:
Request/response for each operand
Serialized off-chip accesses: ***8***

Figure 6-3: Comparing off-chip access serializations

We call a series of accesses to consecutive local dependent operands a *datathread*. If the operands are not dependent, then a traditional system could simply pipeline multiple non-blocking accesses, obtaining them in two serialized off-chip crossings. When a dependence spans two nodes, we view that point as initiating a datathread migration from one node to the other, beginning the access stream of that thread at the new node. The overhead of migrating this conceptual thread is one serialized off-chip access. The cost of maintaining inexpensive datathread migrations is precisely that of maintaining SPSD execution— broadcasting loads and performing computation redundantly at all nodes.

Another conceptual view of asynchronous ESP execution is that from each processor's perspective, it is the main processor, and the others are simply intelligent prefetch engines residing in the main memory modules. From this perspective, the broadcasts the processor sends are merely the state the prefetch engines need to continue performing the accurate prefetching. Since this is a homogenous system, each processor will have this view of the others, of course.

The Massive Memory Machine was able to exploit only one datathread at any time; when a lead change occurred, a new datathread began at the new leader (in Figure 6-1, operands w_1-w_4, w_5-w_7, and w_8-w_9 would constitute three datathreads, assuming each operand is dependent on the previous one). DataScalar systems, because they implement asynchronous ESP with out-of-order issue at each node, may have multiple datathreads running concurrently. DataScalar systems do not require special support for datathreads, since they transparently exploit the locality already inherent in reference streams. However, programs would benefit from special support to increase datathread length or raise the number of datathreads executing concurrently.

6.2.3 Implementation issues

In this subsection, we address three of the implementation issues that must be solved for DataScalar systems to have good performance: caching, speculation, and broadcasts. The discussion in this subsection is in the context of the processor datapath shown later in this chapter, in Figure 6-6.

6.2.3.1 Cache correspondence

In Section 6.1.1 we described static replication of data, in which heavily used pages are copied at each processor running as a DataScalar machine. Static replication is limited in that it cannot use run-time information to reduce off-chip accesses—caches are universally used precisely because this run-time information is so crucial. Dynamic replication, therefore, is crucial to the competitiveness of DataScalar systems.

Dynamic replication in a DataScalar system is analogous to caching in a uniprocessor; processors take a broadcast operand or block of data, and decide to cache the data locally for a period of time (the difference is that multiple processors are all caching the same data instead of just one). However, replicating data dynamically is more complicated than simple caching. The goal of replication is to improve average memory access latency by reducing the number of broadcasts (which are analogous to cache misses in a uniprocessor). If the owner of a datum decides not to broadcast it upon a load, assuming it to be replicated, *every other node must still have that operand*, or deadlock will result. Conversely, if the owner broadcasts the operand and other nodes already have that operand locally, superfluous messages may fill up the queues on the remote nodes (depending on the broadcasting/receiving implementation). Certainly unnecessary broadcasts will waste bandwidth.

One solution for this problem is for all nodes in a DataScalar system to keep exactly the same set of dynamically replicated data, choosing to stop replicating a datum at the same point in the access stream. Furthermore, these nodes should ideally make the decisions about what to keep replicated and what to throw out based on *local information only*—requiring continuous remote communication solely to reduce the number of broadcasts would make DataScalar systems non-competitive.

While many solutions are conceivable, in this dissertation we describe only the solution that we have implemented. Our solution is to fold the decisions about what to replicate dynamically into the first-level caches—a block is considered to be dynamically replicated so long as it is in those caches.[1] If a level one cache miss occurs for communicated data, the owner must broadcast that line to the other nodes. This solution implies that no node may ever miss on a communicated line if another node hits on that line for the same load. We call this the *cache correspondence* problem; data must be kept *correspondent* in the primary caches to prevent deadlocks.[2]

Keeping the caches correspondent is a non-trivial problem. Dynamically scheduled processors will send loads to the cache in different orders, and will also send different sets of instructions (when branch conditions take longer to resolve at some processors than others, allowing more mis-speculated instructions to issue). If two loads to different lines in the same cache set are issued in a different order at two processors, that set will replace different lines, and the caches will cease to be correspondent.

Our solution is to update the primary cache state only when a memory operation is *committed*, not when it is issued. To maintain correct program semantics, instructions must be committed in the same order at all processors, even though they may be issued in different orders. This solution also prevents mis-speculated instructions from affecting the cache contents. Although the caches are updated at instruction commit, broadcasts on misses are still sent out when a load is issued (this policy will result in extra required tag bandwidth).

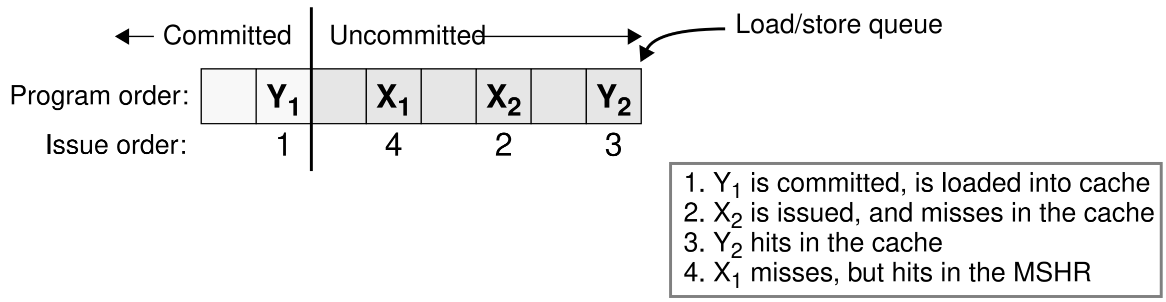
We implement this solution with a structure called a *Commit Update Buffer* (CUB). We envision separate CUBs for instructions and data (ICUBs and DCUBs), but in this paper we

1. It is possible to use lower levels of a multi-level cache hierarchy to perform dynamic replication. We chose to use only the level-one caches because our particular solution requires a tight coupling of the cache tags and the load/store queue in the processor.
2. Stefanos Kaxiras was a co-inventor of the cache correspondence scheme we present in this paper.

only evaluate a DCUB. When a cache miss returns, rather than loading the data into the cache, the line is placed into an entry of the DCUB, and a pointer to that entry is placed in the load/store queue at the entry of the load that generated the miss. Memory operations to the same line are serviced by the data in the DCUB (loads may still be serviced by stores farther ahead in the load/store queue). When a memory operation is committed, the cache tags are updated, and, if necessary, the line is loaded from the DCUB into the cache. A DCUB entry is deallocated when the last entry in the load/store queue that uses that line is committed. In addition to a pointer to the DCUB entry, each entry in the load/store queue contains state that represents whether the instruction missed in the correspondent tags at issue time.

This extra state is necessary because updating the cache at commit time only is sufficient to guarantee cache correspondence, but not to guarantee identical hit/miss behavior at all processors. Since instructions may issue at different times across processors, the same instruction will issue at different commit points in the instruction stream across the processors, causing some to hit and others to miss in their caches. By saving whether a hit or miss occurred at issue time, we can compare that event with the correct commit-time event, and take corrective action if there is a disparity. Corrective actions include issuing a late broadcast (if the node is the owner, and took a false miss), or re-reading the commit-update buffer for the data (if the node is not the owner, and took a false hit).

We show a simple example in Figure 6-4. Two addresses, **X** and **Y**, conflict in the cache. Instructions commit from left to right. The second load to **X** ($\mathbf{X_2}$) misses when issued, but would have hit at commit time if the instructions were issued in program order (because $\mathbf{X_1}$ would have already generated the miss). This is an example of a *false miss*. Analogously, $\mathbf{Y_2}$ hits at issue time because $\mathbf{Y_1}$ had just been committed, but should have missed at commit time (e.g., at another processor, $\mathbf{Y_2}$ might issue after $\mathbf{X_1}$ is committed, causing a miss at issue time instead of a hit). We call this a *false hit*, and deal with it by generating a reparative miss when this situation is detected at commit time (a reparative miss consists of a reparative broadcast by the owner, or a squash to the local receive queue by a non-owner of that datum). We deal with false misses by recognizing that any sequence of accesses to the same line will generate



X and Y are accesses to two lines that conflict in the cache

False miss: X_2 missed at issue but would have hit if in-order issue
False hit: Y_2 hit at issue but would have missed if in-order issue

Figure 6-4: Cache correspondence example

only one miss ($\mathbf{X_1}$ and $\mathbf{X_2}$ in this example). If $\mathbf{X_1}$ issues after $\mathbf{X_2}$, we can "assign" the miss generated by $\mathbf{X_2}$ to $\mathbf{X_1}$, thus ensuring that all processors will generate only one miss for that line.

6.2.3.2 Speculative execution

Fine-grain speculative execution is now present in state-of-the-art processors, and a successful DataScalar architecture must be compatible with speculation. Much of the promise we see in DataScalar comes from out-of-order execution, which enables multiple processors to race ahead simultaneously on different instruction sequences. However, speculation must be tightly controlled: if remote bandwidth is one of the important (and heavily utilized) resources, frequent superfluous broadcasts would hinder performance. The two endpoints for speculative policies are (1) to hold onto speculative broadcasts until the speculative condition is resolved, and (2) to send the broadcast immediately upon issue, and eventually then send a corresponding squash if the load that generated the broadcast is squashed. The former conserves bandwidth at the expense of added latency, while the latter consumes bandwidth while reducing latency (again trading off global bandwidth for reduced remote latency, just as we explored in Chapter 4). A promising approach is to assign confidence values to speculative loads; loads with high correctness confidence should be broadcast and squashed if incorrect, whereas loads with low confidence should be held locally until the speculative condition is resolved.

If broadcasts are sent speculatively, they will remain in the remote receive queues until explicitly deallocated. One method for clearing them from the remote receive queues is for the sending processor to send squashes when the misspeculation was resolved. This method, however, consumes remote bandwidth. A more elegant approach is similar in spirit to our cache correspondence protocol. In the alternative approach, we use local information only to clear receive queues of stale broadcasts that will not be consumed. With each broadcast, we send a tag that is also buffered in the receive queue. The tag consists of a counter that is incremented every time the RUU cycles around (as it is a circular queue). Whenever the highest numerical slot in the RUU is committed (*i.e.*, the RUU cycles around), the counter is incremented and sent to the receive queue. Any receive queue entry whose tag is less than the counter is deallocated (the deallocations can be done in parallel). The counter thus becomes part of the process state and the precise interrupt mechanism, since all of the counters and RUU positions must be made correspondent once a DataScalar task is being restarted after having taken a precise interrupt.

6.2.3.3 Inter-chip communication

Because of the symmetric nature of the DataScalar execution model, all communicated values must be broadcast to all nodes. In general, broadcast operations are both expensive and not scalable. On certain interconnects—such as on a ring or bus—they may be effected with only minor additional cost, though reliable delivery and error recovery are inevitably more complicated for broadcast operations.

Broadcasts on a bus are free, since every bus transaction is an implicit broadcast. However, the very feature that makes broadcasts cheap—the centralized nature of a bus—makes the bus an unlikely candidate for the high-performance interconnect of the future. However, the demise of the bus has been much slower than predicted, and buses may persist for some time to come.

Ring operations, such as the IEEE/ANSI standard Scalable Coherent Interface [66, 111] seem well-suited for this kind of operation. On a ring, operations are observed by all nodes if the sender is responsible for removing its own message. We envision a ring interconnect because of the high-performance capability [101], but broadcast on a ring is complicated by the fact that operands originating at different processors are received at other nodes in different orders. A simple tag can sort out data to different addresses, but the issue is complicated when two accesses to the same datum are broadcast close in time. Complications also arise whenever certain data items must be rebroadcast (e.g., because a receive queue is full), or cancelled.

One technology that may be an excellent match for DataScalar programs running on large systems is optical interconnects. One of the properties of free-space optical interconnects is that they have extremely cheap (essentially free) broadcasts. For massively parallel systems that use optical interconnects, the SPSD execution model may be a good way to reduce the execution time spent in serialized code, thus improving scalability [10].

6.2.4 Other pertinent issues

In this subsection, we discuss the issues of cost and required software support for a DataScalar system.

Cost: Conventional systems today typically consist of a single processor and a collection of memory chips. Each of these components comprise a significant fraction of the total cost of the system. A DataScalar system would consist of a collection of identical chips, each of which costs more than a conventional DRAM chip, but less than a processor chip. When comparing the cost of a DataScalar system and a traditional system with one processor and “dumb memory” (such as the comparison in Figure 6-5), the DataScalar system becomes cost-effective when the performance it adds outstrips the cost of the additional processors.

Wood and Hill showed [131] that for a parallel system to be cost-effective, the *costup* (the relative increase in total cost as more processors are added) should be less than the *speedup* (the relative increase in performance as more processors are added). When memory or interconnect costs dominate those of the additional processors, the system may still be cost-effective even if the speedups are comparatively small.

A majority of the die of most modern processors is devoted to memory, even though the total cache capacity for each is generally only in the tens of kilobytes. We believe that the ratio of on-chip memory area to total chip area will continue to grow in the future, making the relative expense of the processing logic shrink over time. If true, this trend will make memory and packaging the dominant costs of future systems. DataScalar architectures could thus be cost-effective, even though the speedups they provide are much less than linear.

Software support: To the extent that an executing program is non-deterministic, operating system code can be executed in the same manner as user code. Synchronous exceptions, such as for an unaligned address, would be observed at slightly different times at different processors, but would cause no special problems. Consider the case in which a write causes a page fault. Since only one processor actually performs a write to communicated data—the other processors all simply discard their result—only the owning processor would observe the page fault. If the other processors did not recognize the page fault, they might proceed beyond the fault point indefinitely. This problem can be avoided by making sure that all processors have the same page table entries, and actually check for exceptions on every memory operation. (The check could be accomplished by requiring that the store be successfully written into the primary, correspondent cache before being committed.) Thus each processor would observe this page fault. However, asynchronous events could potentially cause difficulty if they are not observed at precisely the same point by all processors. External interrupts, likewise, must be injected into the system with care to assure that all processors observe them at the same point in their execution.

6.3 Evaluating DataScalar architectures

In this section we evaluate the feasibility of DataScalar architectures, in terms of their potential to outperform conventional alternatives. We first quantify (through functional simulation) the amount of traffic that they reduce, which is substantial. Next, we measure the number of consecutive memory operands that fall on a single node, on average, to see how often lead changes occur. Finally, we present the timing results of a full implementation, running with two and four processors.

6.3.1 Traffic reduction

We measured the extent to which the ESP execution model reduces remote communication. With our simulation environment, we simulated a 64-Kbyte, two-way set-associative, write-allocate, write-back, on-chip level-one data cache (this size is consistent with typical cache sizes at the time that SPEC95 was released). We measured the aggregate miss traffic from the cache, and calculated the fraction of traffic that remained once write-backs and requests were eliminated. In Table 6-1, we show this measured fraction for fourteen of the SPEC95 benchmarks. We show both total traffic eliminated, and the reduction in the total number of distinct messages (we count a request/response pair as two transactions). The table shows that, for this cache size, ESP eliminates roughly 0.15 to 0.50 of the off-chip traffic in bytes, and from 0.52 to 0.75 of the individual transactions (because no requests are sent, the transaction reduction will always be at least 0.50).

These results indicate that—for systems in which memory bandwidth is at a premium—implementing ESP is likely to improve performance, or reduce the required system cost to achieve the same performance. These results focus solely on bus traffic reduction—they do not address the performance penalties associated with necessitating broadcasts on interconnects other than buses.

6.3.2 Datathread lengths

In Table 6-2 we show experimental results that measure the mean number of loads falling consecutively on a single node. This is an approximation of datathread length, since we do not account for dependences. All results presented here assumed a four-processor system. These simulations also used the SimpleScalar tools and assumed a cache configuration identical to that presented in Section 6.2.1. For each benchmark, we replicated 32 4-Kbyte pages on each node. We selected the pages to replicate using static profiling. For each benchmark, we saved the number of accesses to each page, sorted the pages by number of accesses, and chose the 32 most heavily accessed pages. We distributed the communicated pages among the nodes round-robin, in blocks with sizes ranging from 4 to 32 pages. The sizes of the distributed blocks of

Metric	m88ksm	gcc	compress	li	perl	vortex		
Traffic	.14	.19	.54	.39	.32	.21		
Transactions	.52	.55	.74	.66	.62	.56		
Metric	**tomcatv**	**swim**	**hydro2d**	**mgrid**	**applu**	**turb3d**	**fpppp**	**wave5**
Traffic	.16	.39	.33	.31	.38	.40	.17	.46
Transactions	.52	.66	.62	.61	.65	.66	.53	.70

Table 6-1: Fractions of off-chip data traffic reduced by ESP

data are shown for each benchmark in the first column of Table 6-2. For each benchmark, we tried to maximize the distribution block size (to improve datathread length) while still keeping it smaller than 1/4 of both the text and the largest data (globals, heap, stack) segment. This action prevented either segment from being completely contained at one processor, a situation which would make the datathread length equal to the number of references.)

The next four columns in Table 6-2 show the distribution of replicated pages among the four segments. Columns seven through nine show the mean (arithmetic) datathread lengths using three different definitions of datathreads. All three methods count consecutive references on a node, beginning the count upon the first reference to a communicated datum local to some node, ending (and restarting) the count upon the next reference to communicated data local to a different node. Column seven approximates datathread lengths using all references to memory (e.g., all cache misses). The second and third columns compute datathread length using only instruction and data references to memory, respectively.

The right-most column shows the average number of contiguous accesses to replicated pages in main memory. High numbers of references to replicated pages will extend average datathread lengths. If references to replicated data are frequent, the threads will tend to be long.

The average datathread lengths in Table 6-2 are high for instructions—over 20 in every case. These large numbers are partially due to the replication of a high percentage of the text pages, which is significant for most programs (li, tomcatv, m88ksim, turb3d, and fpppp have average code datathreads in the hundreds or thousands, and each has from 1/3 to 1/2 of the code replicated across all processors). However the high spatial locality generally found in code reference streams also serves to increase the datathread length.

Benchmark	Dist. size (Kb)	Replicated pages (128Kb)				Datathread length approximation			
		text	global	heap	stack	total	text	data	repl.
tomcatv	32	22	6	2	2	42.3	31486.7	6.7	21.7
swim	32	7	24	0	1	2.1	60.2	2.1	1.0
hydro2d	32	25	5	0	2	1.7	176.9	1.6	1.1
mgrid	32	4	27	0	1	1.5	31.4	1.5	1.0
applu	32	23	8	0	1	2.6	43.3	2.6	1.0
m88ksim	64	16	10	5	1	157.3	859.2	69.1	16.2
turb3d	64	19	12	0	1	1.7	1541.6	1.6	1.1
gcc	256	25	1	0	6	7.4	23.9	4.5	1.2
compress	16	6	25	0	1	103.5	41.7	134.7	1.3
li	16	17	2	12	1	841.2	777.2	2027.1	208.4
perl	128	26	2	3	1	7.6	34.5	4.1	2.1
fpppp	64	27	4	0	1	165.6	755.9	33.7	3.7
wave5	64	17	14	0	1	6.4	171.6	5.9	1.7
vortex	128	27	2	1	2	5.5	21.0	2.9	1.9

Table 6-2: Approximate datathread measurements for a four-processor system

Each row shows the experimental parameters for each benchmark, followed by the results. The first column contains the granularity at which communicated data are distributed round-robin across the processors. The second through fifth columns show the number of pages (4KB each) from each segment that were replicated for each benchmark. The right-most four columns show the arithmetic mean of our datathread length approximations for all reads, all reads to code and data separately, and reads to replicated memory, respectively.

Data reference thread lengths that we see tend to be shorter than the instruction thread lengths. They are low (less than 3) for some of the floating point codes (swim, applu, turb3d, mgrid, and hydro2d). Although floating-point codes tend to have high spatial locality, our approximation of datathreads is cut by interleaved accesses to arrays residing at different processors (*e.g.*, $c[i] = a[i] + b[i]$). Also, some of the spatial locality is filtered out by the cache. The three other floating-point codes have higher average datathread lengths, however, ranging from about 6 to 33. The integer codes tend to have higher datathread lengths than do the floating-point codes. The datathread length for li is high because most of its data set is replicated. The others show average datathread lengths from about three to over 130.

These results show that many programs will be able to exploit datathreading. Ideally, each processor in a DataScalar system will run ahead of the others, finding multiple needed operands and instructions locally, and sending them to the other processors early—sometimes even before the other processors have resolved those addresses.

6.3.3 Performance evaluation

We evaluated a DataScalar system consisting of multiple integrated processor/memory (IRAM) modules connected via a global bus. In Figure 6-5 we show the DataScalar and conventional system organizations that we compare (for a four-node processor system). A traditional system (Figure 6-5a) being compared against a four-processor DataScalar machine (Figure 6-5b) would thus have one-fourth of its main memory on-chip and three-fourths off-chip. We hold the bus, packaging (number of chips), and physical memory storage constant. The DataScalar system contains extra processors and level-one caches, so the total chip area in the DataScalar system is higher (but how much higher depends on the fraction of each chip consumed by the processor and L1 data cache).

In Figure 6-6 we show a diagram of the high-level datapaths present in our simulated DataScalar implementation. We assume split primary instruction and data caches. We replicate the program text at each node, obviating the need for dynamically replicated instructions (and therefore a speculative correspondence protocol). We do support dynamic replication of data, so a DCUB, not the accesses themselves, updates the data cache tags and storage. We assume a fast on-chip main memory, which is insufficiently large to hold an entire program data set, but which is fast enough to eliminate the need for a level-two cache.

We use a simple queue to buffer broadcasts being placed on the global bus. The process of receiving broadcasts is more involved. We call the broadcast-receiving structures (previously called receive queues) that we simulate *Broadcast Status Holding Registers*, or BSHRs. We

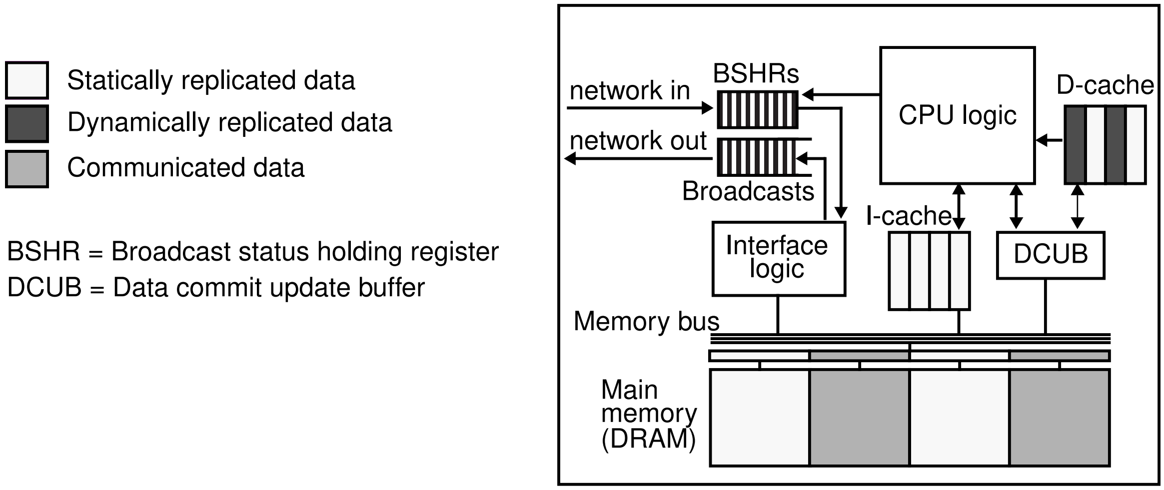


Figure 6-6: Simulated DataScalar chip datapath

implement the BSHRs as a circular queue. When a broadcast arrives from the network, the

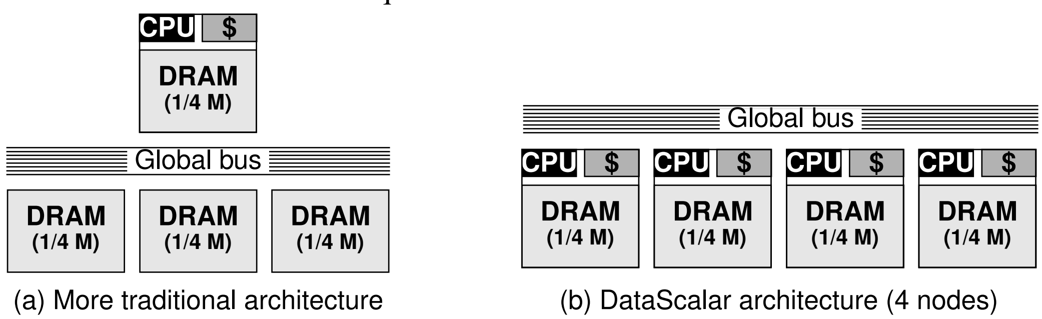


Figure 6-5: Comparing two IRAM organizations

BSHR performs an associative search on that address. If a match occurs, the earliest entry matching that address in the queue is freed and the data are forwarded to the processor. If no match occurs, the BSHR allocates the next entry in the queue and buffers the data. In this case, when the processor issues the request for the data, it finds them waiting in the BSHR, and effectively sees an on-chip hit.

Level-one cache misses become broadcasts if the missing cache line is in communicated memory, and the processor is the owner of that cache line. The miss allocates a BSHR entry if, at a given processor, the miss is to a line that is both communicated and unowned by that processor. In Figure 6-6 we show a datapath from the processor to the BSHR queue; this path is used to squash BSHR entries allocated due to false misses.

To obtain performance results for DataScalar systems, we extended the SimpleScalar out-of-order processor simulator with multiple target contexts. The simulator switches contexts after executing each cycle (i.e., it simulates cycle n for all contexts before simulating cycle $n + 1$ for any context). Unlike the other simulations in this dissertation, we assumed a single-level page table that was locked in physical memory, as opposed to residing in virtual space. Each page table entry has one bit that determines ownership of a communicated page (only one processor will have the ownership bit set for a communicated page; the bit for that page is cleared in the page table entries of all other processors). Address translation thus also produces the ownership status of a page, to more quickly determine the action that must be taken upon a primary cache miss.

For all our experiments, we simulated a processor similar to our timing experiments in Section 5.2 (8-way issue, dynamically scheduled, etc.). The two significant differences are that we assumed a 1GHz processor, instead of the 2GHz used in the previous chapter, and that we assumed perfect branch prediction. Modern branch predictors are already quite accurate, however, and we have no way of knowing what prediction techniques will be prevalent in future processors, or the extent to which these processors will engage in aggressive speculation. This assumption simplified our handling of the BSHRs. Assuming perfect branch prediction will also increase the measured IPC, due to the absence of branch misprediction penalties (the IPC of future processors is likely to be even higher as they engage in speculation that is much more aggressive than branch prediction [114]).

On-chip memories are likely to be significantly faster than DRAMs are today. Using sub-banking, with hierarchical word- and bit-lines, will enable DRAM banks to have access latencies that are comparable with those of cache memories. Current high-density (1 Gb) DRAM prototypes, the processes of which are optimized for density and not speed, have access latencies in the low 30's of nanoseconds [62, 135]. On-chip DRAM banks implemented in hybrid memory/logic processes are likely to be significantly faster.

For our simulations, we assume a memory hierarchy on-chip that is just two levels. The first level is split instruction and data caches, 64KB each with single-cycle access. The caches are direct-mapped (for speed) and the data cache implements a write-back, write-noallocate policy. We believe that this write policy is superior to write-allocate in an ESP-based system (with a write-allocate protocol, a write miss requires sending an inter-processor message, only to overwrite the received data). Both caches are fully non-blocking and can support an arbitrarily high number of outstanding requests. The second level of the hierarchy is composed of high-capacity, on-chip memory banks that can be accessed in 8 ns. They are connected with a 256 bit bus that is clocked at the processor frequency. We assume that our off-chip bus is 128 bits wide and is clocked at 200 MHz (commodity parts that expect to do most of their computing and memory accesses on-chip are not likely to have support for extremely aggressive off-chip connections). We assume BSHRs with 3-ns access latencies and 128 entries. We assume a broadcast queue for the DataScalar simulations, which incurs a two-cycle access penalty before broadcasting data onto the global interconnect (the baseline architecture, similarly, buffers off-chip requests at a network interface that functions as a connection between the local and global buses, also incurring a two-cycle penalty).

As with the previous experiments, the benchmarks that we used were drawn from the SPEC95 suite [117]. This study was performed before we had defined the std input set, so we used the **test** input set in all cases. For some of the inputs, we reduced the number of iterations for some of the benchmarks, as in the **std** set, after performing an analysis to ensure that the reduced number of iterations did not perturb our results).

We simulated six of the SPEC95 benchmarks: go, mgrid, applu, compress, turb3d, and wave5. We ran each benchmark for 200 million instructions or to completion, whichever came first. We did not statically replicate any data pages; all pages were distributed round-robin across all nodes. We ran simulations for both two-processor and four-processor DataScalar systems. Each processor has sufficient capacity to hold one-half and one-fourth of the data set, respectively, for each benchmark.

We compared the Datascalar performance against two points: an identical processor with a perfect data cache (single-cycle access to any operand), and a more traditional system which has the same amount of on-chip memory as does one chip in each DataScalar experiment (as described earlier in this chapter). We thus compare a two-processor DataScalar execution against a system which has the same processor, half the memory on-chip, and half off-chip (to make a fair comparison, the buses are the same, and both systems update the primary data caches at instruction commit, not issue).

The traditional system is likely to benefit if all of the on-chip memory is devoted to a large second- or third-level cache. Such an organization may well outperform a DataScalar organization. A DataScalar system would thus be a better match for systems where multiple processors were available and coupled with regions of memory to begin with; i.e. the designer could use one processor as the sole processor and its local memory as a cache, treating the rest of the processor/memory regions as "dumb memory", or the designer could make use of those processors and run in DataScalar mode. Future partitioned processors (with copious computational capabilities spread across single chips) may be a better match for this execution model.

In Figure 6-7 we plot the instructions per cycle for each experiment. In the upper graph, we show the performance comparison of a two-node DataScalar system, and in the lower graph, we show a four-node DataScalar system. The actual IPC value resides atop each bar. We see that the performance benefits that the DataScalar system has to offer can be substantial, particularly for four nodes. The results are particularly striking for compress, in which the DataScalar system gains almost a doubling of IPC over the traditional architecture. That particular performance gain is so larger because compress, running with the test input, issues many more stores than loads (a ratio of 7:1). The writes and write-back traffic never needs to go off-chip in a DataScalar system. For all other benchmarks, the DataScalar system manages to capture much of the available ILP, approaching the IPC of the perfect data cache in some cases (specifically, wave5 and go).

The DataScalar system deals with a finer-grain distribution of memory better than does the traditional system; the drops in DataScalar performance when going from two-processor to four-processor systems are less than 0.05 IPC (the comparable drops in performance on the traditional system range from 0.1 to 0.6 IPC). The IPC for `wave` actually improves when running on four processors instead of two (the benefits of more processors running datathreads concurrently outweigh the additional off-chip communication). In only two cases (mgrid and turb3d with two nodes) does the DataScalar system perform worse than the traditional system. This abnormality results from poor correspondence protocol performance (a high rate of false

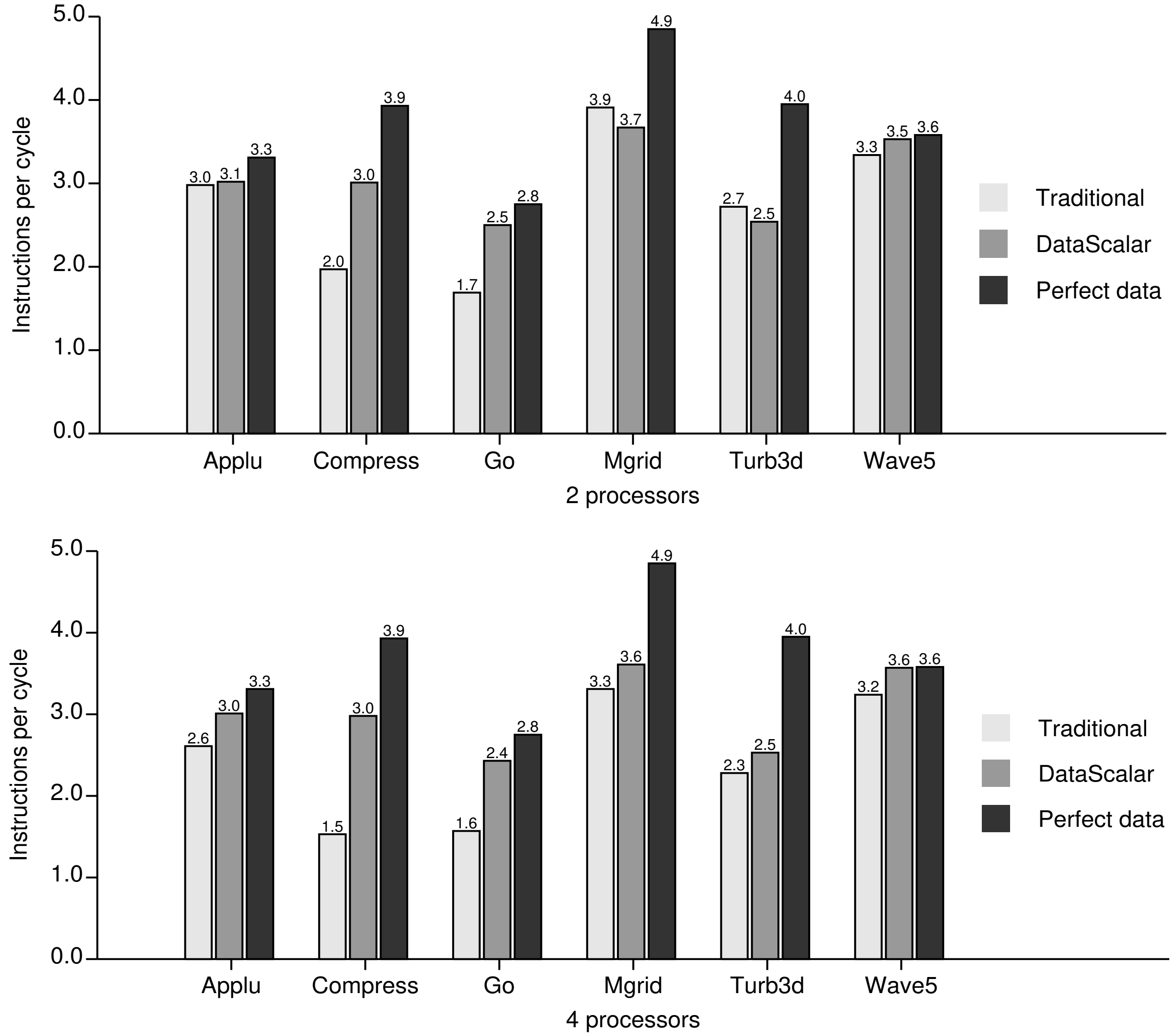


Figure 6-7: Timing simulation results of a DataScalar architecture

hits at one node causes the other node to stall frequently, waiting for the owner to commit the offending load and issue a reparative broadcast).

We present the results of a sensitivity analysis in Figure 6-8. The two benchmarks presented are go and compress, each of which was run to completion. For each benchmark, we plot results assuming the same parameters that we used for the experiments in Figure 6-7, except that we vary only one parameter in each graph. The parameters we varied were: data cache size, main memory access time, global bus clock speed, width of the global bus, and number of RUU entries. On each graph, we plot the IPC for the same five systems as we measured in

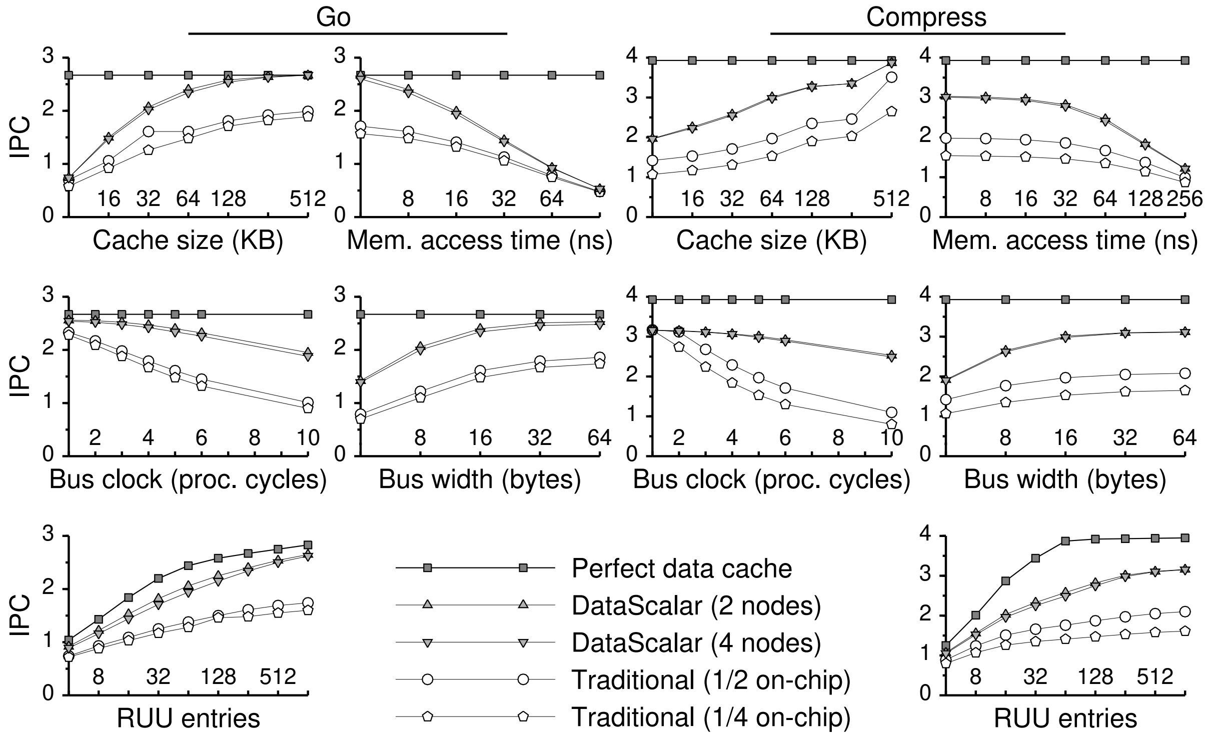


Figure 6-8: Sensitivity analysis of DataScalar experiments

Figure 6-7 (perfect data cache, two- and four-processor DataScalar machines, and traditional systems assuming one-half and one-fourth of the main memory on-chip).

We see that the DataScalar runs consistently outperform the traditional runs over a wide range of parameters. As expected, the performance of the two types of systems converges when memory bank access times come to dominate the latency of a memory request (because DataScalar systems reduce the overhead of transmitting the data, not accessing them). Conversely, when the speed differential between the global and on-chip buses grows, so does the disparity between DataScalar and traditional performance.

In Table 6-3 we list BSHR and broadcast queue statistics from the performance simulations. The parameters are the same as for the experiments reported in Figure 6-7. The numbers are the arithmetic mean across all nodes. The percentages are out of the total number of broadcasts (column one) and out of total BSHR accesses (columns two and three) In columns two and three, we list the percentage of broadcasts that were issued late, at commit time, due to false hits. These percentages will drop for larger caches, since the probability that a block will be replaced in between issue time and commit time is inversely proportional to cache size.

The middle column lists the percentage of BSHR entries that were squashed due to false hits. We note that mgrid and turb3d show the two highest percentages of late broadcasts by far, which confirms our hypothesis that poor behavior of the cache correspondence protocol was responsible for the slight two-node performance drops for these two benchmarks (shown in Figure 6-7).

Benchmark	**Late broadcasts**		**BSHR squashes**		**Data found in BSHR**	
(# of nodes):	2	4	2	4	2	4
applu	10%	9%	12%	12%	10%	7%
compress	11%	8%	16%	22%	8%	4%
go	9%	10%	12%	15%	19%	7%
mgrid	23%	21%	31%	31%	6%	4%
turb3d	38%	37%	59%	59%	3%	1%
wave5	9%	7%	11%	3%	3%	1%

Table 6-3: DataScalar broadcast statistics

The right-most column lists the percentage of remote accesses that were waiting in the BSHR for the local processor's request. Those values range from 2% to 9%, showing that at least some of the time, some effective datathreading is occurring, since a processor needs to be running significantly ahead of another to completely tolerate the transmission latency.

We have shown that memory-centric architectures, and DataScalar systems in particular, are feasible alternative system organizations. Cost issues aside, they generally outperform conventional alternatives. As communication grows in cost relative to computation, this class of architectures will become progressively more cost-effective. Whether the relative component costs shift enough to make DataScalar architectures clearly cost-competitive is an open question, and only time will tell.

Chapter 7

Conclusions

The processor/memory interface is a concept that is fundamental to computing. It must be balanced for best cost/performance, and is continually in need of readjustment with each new improvement in microprocessors, memory systems, and manufacturing technology. In this dissertation, we have shown that the memory system is limiting processor improvements. Providing a sufficiently high performance memory system is simple given unlimited cost, whereas improving processor performance, even given unlimited cost, is more difficult intellectually. As the relative costs of the memory system increase and those of processors decline, the problem of providing a good enough memory system—given cost constraints—becomes the paramount emerging challenge.

7.1 Summary

In this dissertation, we focused on the interface between the processing core and the memory system. Specifically, we examined how the volume of traffic moving across the PMI affects performance, and then proposed techniques and solutions to mitigate the adverse performance impact of that processor/memory traffic.

We first made a case, by analyzing technology and architectural trends, that memory bandwidth will be one of the dominant limits—and perhaps the paramount limit—of scaling microprocessor performance. We then proposed a performance breakdown that dissected execution time into three components: the time spent doing useful processing, the time spent stalling for bank access and transmission memory latency, and the time spent stalling for queueing delays and contention in the memory system. We showed that as microprocessors become more aggressive, with faster clocks and higher levels of ILP, the balance in our decomposition shifts. The fraction of time that processors spend stalling for memory grows significantly, accounting for over half of execution time in aggressive processors (memory stall time grew from an average of 27% to 48% for our least and most aggressive simulated processors, respectively). Furthermore, the balance of latency versus bandwidth stall time shifts for more aggressive processors; the higher-performance processors become much more bandwidth-bound. The bandwidth component of memory stall time grew from 54% to 63% from our least to most aggressive processor, resulting in over 30% of execution time being spent stalling because of memory contention.

Given these results, we proposed a construct called the *minimal-traffic cache* (MTC), to evaluate the potential for reducing unnecessary traffic by placing a lower bound on how much is actually needed. We proposed a related metric called *traffic efficiency*, which compared the traffic ratios of a traditional cache with those of a minimal-traffic cache. Our experimentally measured traffic efficiency results showed the MTC reduced traffic by sometimes large constant factors, with reductions ranging from factors of 2 to factors of 100. We broke this traffic reduction into the component factors of the MTC (fetch size, associativity, replacement policy, and write policy). Our results showed that each of the components can reduce traffic by large constant factors, but the degree to which they do are highly benchmark dependent; there is no "magic bullet" factor that can uniformly reduce traffic (although, naturally, read fetches are more important than the other three factors).

We then proposed a number of policies, targeted at large L2 caches, that attempt to trade off misses and traffic in such a way as to maximize performance. The dual-size fetch policy switched between fetching blocks and subblocks in a subblocked cache, depending on whether spatial locality was high (fetching whole blocks to reduce misses) or low (fetching subblocks to reduce traffic). We evaluated another policy, which we called subblock prefetching, which saved the subblocks used while a block was in the cache, and reloaded only those subblocks upon the next tag miss to that block. We then combined the two into a single policy. Finally, we extended those polices with *bus prioritization*, in which non-critical subblocks predicted by those policies were fetched only when the Rambus channel was idle, reducing contention delays for subblocks that were actually requested. Our results were mixed; the subblock prefetching policy did not reduce misses nearly as much as did the dual-size fetching, and the unified policy beat the two individually only in a few cases. Worse, the performance penalty of using a subblocked cache—necessary to implement our policies cleanly in a traditionally managed cache—was sufficiently high that the policies recouped the lost performance in only a few cases.

The next area that we explored was the organization and management of large on-chip memories. We discussed how the use of some mechanisms from virtual memory management (as opposed to traditional cache management) may be a good match for the on-chip memories in the near- and medium-term future. We proposed three classes of hybrids: *physical hybrids*, in which the processor chip contains some physical memory and some cache, physically separate; *logical hybrids*, in which a combination of cache and virtual memory mechanisms are used to manage a single structure uniformly; and *unified hybrids*, in which blocks of data in a single structure are either treated as cached data or virtual memory pages, depending on the management policies. We evaluated the former two, and merely described the third class of hybrids.

To explore the space of logical hybrids, we described a taxonomy that specified the major differences between cache and virtual memory mechanisms, and used this taxonomy to sift through a number of possible hybrids. We discarded most, but chose to evaluate one that looked promising: the *indirect cache (extended)*. The ICE used software address translation to access cache lines in a large L2 cache, and used a tag cache to speed the translation process. The performance results for the ICE were good in some cases, but, like the traffic policies, did not show consistent improvement over an aggressive baseline. When we evaluated the ICE with subblocked tags, and coupled that implementation with the unified traffic policies, however, we found that the two sets of techniques worked synergistically. Together, the two performed both uniformly and substantially better than the aggressive baseline (8% - 30%), and even outperforming (on average, by 1.6%) the baseline with per-benchmark optimal block sizes chosen. This synergy occurred because the ICE removed the main implementation drawback to the traffic policies: caching data at a coarse (block) granularity.

We presented a brief analysis of a physical hybrid, showing global miss rates for several organizations that had all on-chip memory managed as physical memory, all on-chip memory managed as a large L2 cache, and various points in between. For on-chip capacities that are a significant fraction (1/32 to 1/2) of the applications' data set sizes, we showed that having a large physical memory structure on-chip, with a smaller L2 cache for off-chip data, can be competitive with an all-cache scheme, and furthermore shows reduced miss rates in a few cases. The caveat is that we used application (and data set) specific profiling to choose the pages mapped on the processor chip in the physical memory banks (based on total frequency of accesses), and less intelligent static mappings are likely to incur significantly more misses.

With the traffic optimizing policies and the ICE, we have proposed implementable improvements that address fetch size and associativity, two of the four factors by which traditional caches and MTCs differ. We proposed two techniques (selective write validate and correlated replacement) that address the other two factors, but did not evaluate them in this dissertation.

The last major study we presented in this dissertation was the DataScalar architecture, which, among other benefits, eliminated write and request traffic from the global interconnect. We showed that it is possible to implement a working DataScalar system that achieves performance that is consistently higher than a competitive baseline. We proposed solutions for dealing with caching and speculation in an asynchronous implementation of the ESP execution model, and implemented them in our simulator. Our experiments with four processors showed speedups from 9% to 100% on unmodified serial binaries.

7.2 Looking back

In this final section, I discuss our results from the perspective at the end of the Ph.D. process, and describe what I consider to be the impact of this work, the impact thus far, and which portions of the work are likely to have the most impact in the future.

The memory bandwidth portion of my dissertation research, which appeared in ISCA'96, is probably the most cited so far, and thus ostensibly has been the most influential. The initial publication that described some of our ideas and philosophical framework (with the cumbersome title “The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors” [12]) was widely disseminated and read (particularly by industry, if anecdotal evidence can be believed). The report, while it contained some potentially good (if hazy) ideas, was fairly naive. I saw the low efficiency measurements, and concluded that the hardware caching paradigm was wasteful and that explicit cache management by the compiler could make much better use of the resources, and thus be more cost-effective. That viewpoint was certainly supported by Shen’s results [64], which showed that if you discarded the caching paradigm, and just focused on optimally managed values, you could obtain orders of magnitude more effective bandwidth. After months of trying, I was unable to come up with anything implementable that could outperform caches. Our report failed to recognize that once you have the software break the dynamic caching paradigm, you must work extremely hard just to break even. Recent and ongoing work at Wisconsin [90] (and elsewhere [126]) has had elements of this philosophy (separating values from the name space to bypass all of the baggage of virtual memory and the memory hierarchy), but they used data dependence prediction rather than compiler analysis.

The follow-on paper to our technical report, the ISCA memory bandwidth paper [13], had two main contributions. The first was the case that pin bandwidth was going to be the prime bottleneck in future systems. The second was to place and analyze lower bounds on cache traffic. I used experimental evidence to show that limited bandwidth, particularly pin bandwidth, was growing as a contributor to performance loss. However, my simulations assumed large, fast off-chip L2 caches, which essentially forced the memory bottleneck to the pin interface (main memory latencies were thus rarely incurred because of the large caches and small benchmarks, plus the bus width to the L2 was necessarily constrained, which is not the case for more modern L2 caches that are on-chip). That is why both my results and the IRAM group’s earlier results [42] showed such little gain from having full processor/memory integration; the truth is that once you have a large on-chip L2, the off-chip bandwidth is not nearly as much of a problem. I honestly believe that we are currently in an “inflection point”–eventually, designers will be able to place as much computational power on the chip as they have available off-chip bandwidth (which the on-chip storage will enhance). Right now, we are starting to see large on-chip storage, but not yet effectively unlimited processing power (modern processors still consume too large a percentage of the die, but their footprints on the die are shrinking quickly). In the short term, the only places that processor/memory integration could be realized are in other markets, which need systems with fewer chips for cost reasons (price/performance as opposed to performance). The IRAM group seems to be moving in such a direction, toward low-power, embedded processors that are designed for PDAs and the like.

My work in improving the performance of large L2 caches, with both the traffic optimizing policies and alternative management organizations, was initially disappointing. Caches (particularly large L2 caches) work well, particularly when you feed them small benchmarks like most of SPEC95. We (Steve Reinhardt and I) didn't see large or consistent gains from either the traffic policies or the ICE, until we put them all together, and saw (as our intuition had predicted) that they worked synergistically. We have some evidence that these techniques show even larger performance gains (using some traces obtained from Intel) than they did for SPEC, which is encouraging. The traces were instruction traces that contained between 19 and 149 million instructions, and ranged from 0.2 to 2.7 million distinct references. The two main drawbacks of this work are the complexity of implementing all of these techniques together (that requirement makes it much less likely that industry will give some of these ideas a try) and the persistent, per-tag state store. We need to explore the performance impact of having state for a finite number of physical tags, before industrial architects are likely to buy the results. Another question is whether some of these ideas should be applied to current caches without implementing everything together; if designers were going to build a subblocked cache for other reasons, then some of the traffic optimizing policies might make sense to include. Finding a way to get around the granularity issue in a hardware-managed cache (such as decoupled sectoring) could also permit the traffic policies to work well without requiring the ICE. Finally, something like the ICE could have impact if industrial designers have other needs for flexibility (partitioning the L2 for multiprogramming or multithreading, for example). Whether this work will have any industrial impact is too early to tell, but possible.

The DataScalar work was the part of my thesis that I think will have the most long-term impact. Near-term, it seems to have been largely ignored (although the ideas have been widely disseminated within a major microprocessor vendor). Although the performance results we reported in our ISCA paper were essentially meaningless (by the time the technology is a good match for a DataScalar-like system, the processors, workloads, and latencies everywhere will bear no resemblance to those we used), we did show that an asynchronous implementation of the ESP execution model could outperform a conventional alternative. For such an execution model to become cost-effective, the underlying system must have the correct sets of parameters and costs (distributed memory, cheap processors, high-latency communication). We evaluated the architecture in a MOP context, but given our performance results in Section 5.4, it seems unlikely that high-performance processors will be fully integrated with memory anytime soon (and designers would likely be unwilling to put in all of the necessary hardware and software support to run asynchronous ESP in low-end, low-cost embedded systems). The technology that may be a better match is the implementation of large, high-performance chips, which have multiple powerful, distributed computational units on-chip, but long delays to transmit global signals. Running in the base DataScalar mode all the time may be overkill, but having an asynchronous ESP mode, running part of the time (or running select subsets of the computation as a virtual DataScalar system) may be advantageous. My research plans for the next few years include trying to develop a system that uses these ideas to outperform all the alternatives both in terms of performance and cost/performance, in addition to showing that it is implementable.

While other parts of this dissertation may have had some short-term impact (the memory bandwidth work), and other parts have potential for some medium-term impact (the ICE), the DataScalar work is the only part of this research, in my opinion, which has any potential for fundamental, long-term impact on how computation and storage resources are organized.

References

[1] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] Thomas Alexander and Gershon Kedem. Distributed Prefetch-buffer/Cache Design for High Performance Memory Systems. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.

[3] A. Asthana, H. V. Jagadish, J. A. Chandross, D. Lin, and S. C. Knauer. An Intelligent Memory System. *Computer Architecture News*, 16(4):12–20, September 1988.

[4] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, Ames, CA, August 1991.

[5] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Bryan Black and John Paul Shen. Calibration of Microprocessor Performance Models. *IEEE Computer*, 31(5):59–65, May 1998.

[7] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. BLITZEN: a Highly Integrated Massively Parallel Machine. In *Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation*, pages 399–406, October 1988.

[8] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[9] Doug Burger, Todd M. Austin, and Steven Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. Technical Report 1308, Computer Sciences Department, University of Wisconsin, Madison, WI, July 1996.

[10] Doug Burger and James R. Goodman. Exploiting Optical Interconnects to Eliminate Serial Bottlenecks. In *Proceedings of the Third International Conference on Massively Parallel Processing Using Optical Interconnects*, October 1996.

[11] Doug Burger and James R. Goodman. Billion-Transistor Architectures. *IEEE Computer*, 30(9):46–48, September 1997.

[12] Doug Burger, James R. Goodman, and Alain Kägi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madison, WI, January 1995.

[13] Doug Burger, James R. Goodman, and Alain Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[14] Doug Burger, James R. Goodman, and Alain Kägi. Limited Bandwidth to Affect Processor Design. *IEEE Micro*, 17(6):55–62, December 1997.

[15] Doug Burger, Stefanos Kaxiras, and James R. Goodman. DataScalar Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 338–349, June 1997.

[16] Doug Burger, Steven K. Reinhardt, and Wei fen Lin. Alternative Designs for Large On-Chip Caches. Technical Report 1390, UWCS, Feb 1999.

[17] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Technical report, U.S. Army Ordinance Department, 1946.

[18] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[19] Steve Carr and Ken Kennedy. Blocking Linear Algebra Codes for Memory Hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, page ?, December 1989.

[20] J. H. Chang, H. Chao, and K. So. Cache Design of a Sub-Micron CMOS System/370. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 208–213, June 1987.

[21] Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[22] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen mei W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 69–73, November 1991.

[23] Jim Childers, Peter Reinecke, and Hiroshi Miyaguchi. SVP: A Serial Video Processor. In *Proceedings of the 1990 IEEE Custom Integrated Circuits Conference*, pages 17.3.1–17.3.4, May 1990.

[24] Daniel Citron and Larry Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, January 1995.

[25] Eugene L. Cloud. The Geometric Arithmetic Parallel Processor. In *Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation*, pages 373–381, October 1988.

[26] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 128–137, May 1994.

[27] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood

Cliffs, NJ, 1973.

[28] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[29] Jordi Cortadella and Teodor Jové. Dynamic RAM for On-chip Instruction Caches. *Computer Architecture News*, 16(4):45–50, September 1988.

[30] Richard Crisp. Direct Rambus Technology: The New Main Memory Standard. *IEEEM*, 17(6):18–27, December 1997.

[31] Stefanos Damianakis, Kai Li, and Anne Rogers. An Analysis of a Combined Hardware-Software Mechanism for Speculative Loads. Technical Report TR-455-94, Princeton University, Princeton, NJ, April 1994.

[32] Per-Erik Danielsson, Par Emanuelsson, Keping Chen, and Per Ingelhag. Single-Chip High-Speed Computation of Optical Flow. In *In IAPR International Workshop on Machine Vision Applications*, pages 331–335, November 1990.

[33] M. F. Deering, S. A. Schlapp, and M. G. Lavelle. FBRAM: A New Form of Memory Optimized for 3D Graphics. In *Proceedings of SIGGRAPH 94*, pages 167–174, Orlando, FL, July 1994.

[34] Duncan G. Elliott, W. Martin Snelgrove, Christian Cojocaru, and Michael Stumm. A PetaOp/s is Currently Feasible by Computing in RAM. In *In PetaFLOPS Frontier Workshop*, Washington DC, February 1995.

[35] Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm. Computational Ram: A Memory-SIMD Hybrid and its Application to DSP. In *Custom Integrated Circuits Conference*, pages 30.6.1–30.6.4, Boston, MA, May 1992.

[36] Babak Falsafi and David A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[37] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[38] Matthew Farrens and Arvin Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 128–137, May 1991.

[39] Matthew Farrens, Gary Tyson, and Andrew R. Pleszkun. A Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 338–347, April 1994.

[40] Richard C. Foss. Implementing Application Specific Memory. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 260–261, February 1996.

[41] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin,

Madison, WI, December 1993.

[42] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom ANderson, and Katherine Yelick. The Energy Efficiency of IRAM Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 327–337, June 1997.

[43] John W. C. Fu and Janak H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[44] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Jr. Frederick P. Brooks, John G. Eyles, and John Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancemts in Pixel-Planes. In *Proceedings of SIGGRAPH'85*, pages 111–120, San Francisco, CA, July 1985.

[45] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdes. A Massive Memory Machine. *IEEE Transactions on Computers*, C-33(5):391–399, May 1984.

[46] Glenn Giacalone0et0al. A 1MB, 100MHz Integrated L2 Cache Memory with 128b Interface and ECC Protection. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 370–371. IBM, February 1996.

[47] J. D. Gindele. Buffer Block Prefetching Method. *IBM Tech. Disclosure Bull.*, 20(2):696–697, July 1977.

[48] Gideon Glass and Pei Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of the 1997 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 115–126, June 1997.

[49] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: the Terasys Massively Parallel PIM Array. *IEEE Computer*, 28(3):23–31, April 1995.

[50] Maya Gokhale, Bill Holmes, Ken Iobst, Alan Murray, and Tom Turnbull. A Massively Parallel Processor-in-Memory Array and its Programming Environment. Technical Report SRC-TR-92-076, Supercomputer Research Centre - Institute for Defense Analyses, 17100 Science Drive, Bowie, MD, November 1992.

[51] James R. Goodman. Using Cache Memory To Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[52] James R. Goodman and Honesty C. Young. Comments on "A Massive Memory Machine". *IEEE Transactions on Computers*, C-35(10):907–910, October 1986.

[53] Sridhar Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[54] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-Directed Data Prefetching in Multiprocessor with Memory Hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 354–368, June 1990.

[55] Linley Gwennap. Digital 21264 Sets New Standard. *MPR*, pages 1–6, October 28 1996.

[56] Linley Gwennap. Alpha 21364 to Ease Memory Bottleneck. *MPR*, pages 12–15, October 26 1998.

[57] R. A. Heaton and D. W. Blevins. BLITZEN: a VLSI Array Processing Chip. In *Proceedings of the 1989 Custom Integrated Circuits Conference*, pages 12.1.1–12.1.5, San Diego, CA, May 1989.

[58] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, November 1987.

[59] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(1), January 1998.

[60] Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood. Wisconsin Architectural Research Tool Set. *Computer Architecture News*, 21(4):8–10, August 1993.

[61] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 158–166, June 1984.

[62] Masashi Horiguchi0et0al. An Experimental 220MHz 1Gb DRAM. In *Proceedings of the 1995 International Solid-State Circuits Conference*, pages 252–253. Hitachi, February 1995.

[63] L. P. Horwitz, R. M. Karp, R. E. Miller, and A. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[64] Andrew S. Huang and John P. Shen. A Limit Study of Memory Requirements Using Value Reuse Profiles. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 71–81, December 1995.

[65] IBM Microelectronics and Motorola. *PowerPC 601: RISC Microprocessor User's Manual*, 1993.

[66] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[67] J. M. Jennings, E. W. Davis, and R. A. Heaton. Comparative Performance Evaluation of a New SIMD Machine. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 255–258, October 1990.

[68] Lizy Kurian John, Raghuveer Reddy, Vijay Kammila, and Peter Maurer. Investigating the Use of Cache as a Local Memory. In *Proceedings of the 1995 International Conference on High Performance Computing*, 1995.

[69] T.L. Johnson and W.W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[70] Norman P. Jouppi. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[71] Norman P. Jouppi and Parthasarathy Ranganathan. The Relative Importance of Memory Latency, Bandwidth, and Branch Limits to Performance. In *Workshop on Mixing Logic and DRAM, held at the 24th International Symposium on Computer Architecture*, June 1997.

[72] Toni Juan, Dolors Royo, and Juan J. Navarro. Dynamic Cache Splitting. In *Proceedings of the XV International Conference of the Chilean Computer Society*, November 1995.

[73] Richard Eugene Kessler. *Analysis of Multi-Megabyte Secondary CPU Caches*. PhD thesis, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706-1685, July 1991.

[74] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-Level Storage System. *IRE Transactions, EC-11*, 2:223–235, April 1962.

[75] Peter M. Kogge. EXECUBE - A New Architecture for Scalable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages I77–I84, August 1994.

[76] Peter M. Kogge, Toshio Sunaga, Hisatada Miyataka, Koji Kitamura, and Eric Retter. Combined DRAM and Logic for Massively Parallel Systems. In *Proceedings of the 1995 Conference on Advanced Research in VLSI*, pages 4–16, Chapel Hill, NC, March 1995.

[77] Leonidas I. Kontothanassis, Rabin A. Sugumar, G. J. Faanes, James E. Smith, and Michael L. Scott. Cache Performance in Vector Supercomputers. In *Proceedings of Supercomputing '94*, pages 255–264, November 1994.

[78] Christoforos Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, , Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[79] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[80] D. J. Kuck and B. Kumar. A System Model for Computer Performance Evaluation. In *Proceedings of the International Symposium on Computer Performance, Modeling, Measurement, and Evaluation*, pages 187–199, March 1976.

[81] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998.

[82] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[83] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multi-

processor. *IEEE Computer*, 25(3):63–79, March 1992.

[84] J. S. Liptay. Structural Aspects of the System/360 Model 85 II: The Cache. *IBM Systems Journal*, 7(1), 1968.

[85] Philip Machanick, Pierre Salverda, and Lance Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy. In *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, October 1998.

[86] Doug Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, September 1997.

[87] Geoffrey D. McNiven and Edward S. Davidson. Analysis for Memory Referencing Behavior For Design of Local Memories. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 56–63, May 1988.

[88] Hiroshi Miyaguchi, Hujime Krasawa, and Xhinichi Watanabe. Digital TV with Serial Video Processor. *IEEE Transactions on Consumer Electronics*, 36(3):318–326, August 1990.

[89] Andreas Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[90] Andreas Moshovos and Guri Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[91] David Nagle, Richard Uhlig, Trevor Mudge, and Stuart Sechrest. Optimal Allocation of On-chip Memory for Multiple-API Operating Systems. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 358–369, April 1994.

[92] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, March/April 1997.

[93] David Patterson, Tom Anderson, and Kathy Yelick. The Case for IRAM. In *Proceedings of HOT Chips 8*, Stanford, CA, August 1996.

[94] Andrew R. Pleszkun and E. S. Davidson. Structured memory access architecture. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 461–471, August 1983.

[95] Charles Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, January 1995.

[96] Betty Prince. Memory in the fast lane. *IEEE Spectrum*, 31(2):38–41, February 1994.

[97] Steven A. Przybylski. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures*. MicroDesign Resources, Sebastopol, CA, 1994.

[98] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Ap-

proach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[99] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[100] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[101] Steven L. Scott, James R. Goodman, and Mary K. Vernon. Performance of the SCI Ring. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 403–414, May 1992.

[102] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors. 1997.

[103] André Seznec. Decoupled Sectored Caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 384–393, April 1994.

[104] Toru Shimizu et al. A Multimedia 32b RISC Microprocessor with 16Mb DRAM. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 216–217. Mitsubishi Electric Co., February 1996.

[105] Daniel P. Siewiorek, C. Gordon Bell, and Annel Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982.

[106] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[107] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Real-Time Signal Processing IV*, pages 241–248, 1981.

[108] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[109] James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[110] James E. Smith and Andrew R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[111] IEEE Computer Society. Scalable Coherent Interface (SCI). *ANSI/IEEE Std 1596-1992*, August 1993.

[112] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[113] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple

Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[114] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[115] Gurindar S. Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.

[116] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, VA, December 1991.

[117] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, VA, September 1995.

[118] Harold S. Stone. A Logic-in-Memory Computer. *IEEE Transactions on Computers*, pages 73–78, January 1970.

[119] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, May 1993.

[120] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 171–193, October 1994.

[121] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A New Page Table for 64-bit Address Spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 184–200, December 1995.

[122] Olivier Temam. Investigating Optimal Local Memory Performance. In *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 218–226, October 1998.

[123] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[124] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[125] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[126] Gary Tyson and Todd Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[127] Gary Tyson, Matthew Farrens, John Matthews, and Andrew Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 93–103, December 1995.

[128] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997.

[129] Shlomo Weiss and James E. Smith. *POWER and PowerPC*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1994.

[130] Loring Wirbel. NSA taps Cray Computer, National. *Electronic Engineering Times*, 1(816):39–40, September 1994.

[131] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[132] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):24, March 1995.

[133] Nobuyuki Yamashita, Tohru Kimura, Yoshihiro Fujita, Yoshiharu Aimoto, Takashi Manaba, Shin'ichiro Okazaki, Kazuyuki Nakamura, and Masakazu Yamashina. A 3.84GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2Mb SRAM. In *Proceedings of the 1994 International Solid-State Circuits Conference*, pages 260–261. NEC, February 1994.

[134] Tadaaki Yamauchi, Lance Hammond, and Kunle Olukotun. A Single Chip Multiprocessor Integrated with DRAM. In *Workshop on Mixing Logic and DRAM, held at the 24th International Symposium on Computer Architecture*, June 1997.

[135] J. H. Yoo et sal. A 32-bank 1Gb DRAM with 1 GB/s Bandwidth. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 378–379. Samsung Electronics Co., February 1996.

[136] Albert Yu. The Future of Microprocessors. *IEEE Micro*, pages 46–53, December 1996.

Appendix A

Quantifying Latency and Bandwidth Stalls

In this appendix, we quantify experimentally the effects of latency tolerance optimizations on the execution time breakdown. Our results show that as we incorporate techniques to tolerate memory latency more aggressively, the fraction of time spent stalling for bandwidth increases. Furthermore, while the latency tolerance techniques that we measure are successful at reducing raw latency stalls (f_L), they are ineffective at reducing f_B.

A.1 Experimental methodology

To measure f_P, f_L, f_B (derived in Section 1.2.2), we simulate three configurations per experiment (from which we obtain T_P, T_I, and T). Our simulations were based on the SimpleScalar target machine described in Chapter 2, with parameters described later in this section. To obtain T_P, we run a simulation with a perfect memory system, in which every load and store hits in the L1 cache (one cycle). We measure T_I by simulating a memory hierarchy assuming infinitely wide paths between adjacent levels of the hierarchy. (We define "infinitely wide" by assuming that any number of requests of any size can be transmitted across any bus in one cycle, and that there is no need for arbitration). Finally, we measure T by simulating the full memory system, including contention at all buses.

In this appendix, we present breakdowns for three separate sets of experiments, published in previous studies. We will denote the experiment sets as **E1**, **E2**, **E3**, respectively. In the first execution time breakdown that we measured (**E1**), we used the SPEC92 benchmarks, as we did not yet have access to SPEC95. In the second set (**E2**), we used a subset of the SPEC95 benchmarks. We published both sets of results in ISCA23 [13]. More recently, we were invited to publish a rewrite of the ISCA paper in IEEE Micro [14]. We reran a set of the

SPEC95 benchmarks with our more mature simulation environment, which we improved over the intervening year, and ran the experiments with updated parameters that were more current than those in **E1** and **E2**. We will refer to that most recent set of experiments as **E3**.

In Table A-1, we list the inputs used for the various benchmarks in **E1**-**E3**. At the time of these studies, we had not yet performed the analysis on the benchmark inputs and data set sizes presented in Chapter 2. Consequently, in many cases we used input sets that were significantly smaller than the **ref** data sets. Since smaller inputs and data set sizes tend to shift the results to be more processor-bound, however, these results are therefore conservative from a memory system perspective.

In Table A-2 we list the memory system parameters associated with each experiment set. Since we did not scale the data set sizes of the benchmarks for the newer experiments, the sizes of the various levels of the memory hierarchy remain the same (with one exception). At the time of the first study (**E1** and **E2**), we chose cache sizes that were typical of high-performance machines at the time (64 KB split level-one caches and an off-chip, 1MB level-two cache). When we moved to the newer study, we doubled the size of the L2 cache to compensate for the fact that SPEC95 has larger data sets than SPEC92, but we did not scale up the L2 cache to more than 2MB, and we left the L1 caches the same size. Since the data sets remained unchanged, our goal was to use cache sizes that were from a processor generation equivalent to the benchmark generation (circa 1995, when SPEC92 was still in wide use and SPEC95 was just released). We did scale the timing parameters to reflect more current values, however, assuming that the memory banks got faster (in particular, assuming a more aggressive 14ns for the L2 cache; 30ns was too slow for newer machines). We did not simulate bank contention at main memory, since the large L2 caches (coupled with the small data sets) kept the global L2 miss rates sufficiently low (a mean global miss ratio, measuring the data stream only, of 0.004 across all benchmarks for the 1MB cache, and lower for the 2MB cache) that memory bank contention would be a small factor. Like the small inputs, this assumption makes the results more conservative, since the absence of bank contention will only serve to increase processor utilization.

E1	**compress**	**eqntott**	**espresso**	**su2cor**	**swm**	**tomcatv**
	train	int_pri_3.eqn	mlp4 only	in.short	180x180, 50 it.	256x256, 10 it.
E2	**applu**	**hydro2d**	**li**	**su2cor**	**swim**	**vortex**
	33x33x33, 2 it.	test, 1 it.	test	test	test	test
E3	**compress**	**ijpeg**	**perl**	**su2cor**	**swim**	**vortex**
	train	train	test	test	test	train, 1it

Table A-1: Input files used for benchmarks in experiments **E1**-**E3**

Structure	E1 (SPEC92)	E2 (SPEC95)	E3 (SPEC95)
L1 cache	128KB unified	64KB I, 64 KB D	64KB I, 64 KB D
	Direct mapped	Direct mapped	Direct mapped
	On-chip, 1-cycle access	On-chip, 1-cycle access	On-chip, 1-cycle access
L1/L2 bus	128 bits wide	128 bits wide	128 bits wide
	bus/proc clock: 1/3	bus/proc clock: 1/4	bus/proc clock: 1/5
L2 cache	1MB	2MB	2MB
	4-way set assoc.	4-way set assoc.	4-way set assoc.
	Off-chip, 30 ns access	Off-chip, 30 ns access	Off-chip, 14 ns access
L2/memory bus	64 bits wide	64 bits wide	64 bits wide
	bus/proc clock: 1/3	bus/proc clock: 1/4	bus/proc clock: 1/5
Memory	90 ns access	90 ns access	80 ns access
	No bank conflicts	No bank conflicts	No bank conflicts

Table A-2: Memory system simulation parameters

In Table A-3 we list the processor parameters that we used for the experiments. For each experiment set, we ran 6 experiments, which we label **A**-**F**. In Table A-3, parameters that differ among **E1**, **E2**, and **E3** are listed for all three, separated by slashes, in the order **E1/E2/E3**. We ran six experiments per set to examine the effects of latency tolerance techniques upon the execution time breakdown. We used 4-wide issue superscalar processor cores for all experiments, each of which uses a two-level adaptive gshare branch predictor. Experiments **A**, **B**, and **C** all use statically scheduled (in-order issue) cores, while **D**, **E**, and **F** all use dynamically scheduled (out-of-order issue) cores, based on the RUU described in Chapter 2. **A** and **B** use blocking caches, while **C**, **D**, **E**, and **F** use non-blocking (lock-up free) caches [79]. To improve cache performance, **B** uses large cache lines (factor of two larger), while **E** and **F** use tagged prefetching [47]. **F** uses a more aggressive processor core than **A**-**E** for each of the

Experiment	A	B	C	D	E	F
Processor	in-order issue			out-of-order issue		
Clock speed	300/400/500 MHz					0.3/0.6/1 GHz
RUU slots	16/64/128					64/128/256
L/S Q entries	8/32/64					32/64/128
Branch predictor	8K/8K/16K					16K/16K/32K
Cache	Blocking		Lockup-free			
L1:L2 block sizes	32:64	64:128	32:64			
HW prefetch	no				yes	

Table A-3: Processor simulation parameters (**E1**/**E2**/**E3**)

experiment sets. We can isolate the effects of the individual latency tolerance mechanisms by comparing pairs of experiments: larger cache blocks (**B/A**), non-blocking caches (**C/A**), dynamic scheduling (**D/C**), tagged prefetching (**E/D**), and a more aggressive processor core (**F/E**).

Our implementation of blocking caches differs between **E1/E2** and **E3**. In **E1** and **E2**, we assume that a miss blocks the cache, but that hits may still occur while the memory system is servicing the miss (hit-under-miss). In **E3**, we implemented the blocking, hit-under-miss policy by restricting all caches to one miss status holding register (MSHR), which allows combining of up to 8 separate requests for the same cache block (MSHR hits). The cache may thereby service multiple misses simultaneously if they are to the same cache block.

Finally, we assume that multiplexed data/address lines are used only on the main memory bus (the on-chip and cache buses have separate address and data lines), that all channels are bidirectional, that all memories return the critical word first, and that we have an infinitely deep write buffer.

A.2 Simulation results

In Figure A-1, Figure A-2, and Figure A-3, we depict the execution time breakdowns for **E1**, **E2**, and **E3**, respectively. In all three figures, each bar represents the breakdown of execution time into f_P, f_L, and f_B (black, dark grey, and light grey bars, respectively) for one experiment.

The number atop each bar represents the value of f_B for that experiment. The execution times for each benchmark are normalized to the processing time (T_P) for experiment **A**.

A.2.1 E1 results

In this experiment set (Figure A-1), several of the benchmarks (eqntott and espresso in particular) do not spend much of their time stalled for memory; for these benchmarks, f_P is high (over 0.90 for all experiments). The small data sets typical of the SPEC92 benchmarks produce high hit rates in both the 64KB L1 caches and in the 1MB L2 cache, causing little time to be spent in the memory system. For experiments **A**-**C** with the other four benchmarks (compress, su2cor, swm, and tomcatv), the time spent stalled for memory ($f_L + f_B$) is more significant: roughly a quarter (su2cor) to a half (compress). The bulk of the memory stall time for experiments **A**-**C** is spent stalling for latency (f_L). Adding dynamically scheduled cores changes the breakdown substantially. For the experiments with dynamically scheduled cores (**D**-**F**), the processing time (f_P) is cut roughly in half, the latency stall time (f_L) is reduced (dramatically in some cases), and the bandwidth stall time (f_B) increases, both in relative and in absolute terms, becoming the dominant component of memory stall time in most cases.

Increasing the block size from 32 to 64 bytes in the L1 cache, and 64 to 128 bytes in the L2 cache, improved the performance of some applications but not others (compare experiments **B** and **A**). For the SPEC92 version of compress, the unified 128KB L1 cache has a high miss rate of 4.20% for 32 byte blocks. Increasing the L1 block size to 64 bytes causes a slight increase in the miss rate, to 4.53%. This increase causes a correspondingly small increase in f_L. f_B increases by a factor of four, however (0.03 to 0.13), since each L1 miss requires 6 extra cycles to fill the cache (2 additional bus cycles, since 32 extra bytes must be moved across the 16-byte bus, at 3 processor cycles per bus cycle), contributing to f_B for every miss. For su2cor, the larger block size reduces the L1 miss rate slightly (2.97% to 2.53%), causing a decrease in f_L, but the increase in f_B (0.02 to 0.08) overcomes the reduction in f_L, causing a net increase in execution time. For swm and tomcatv, the L1 miss rates are reduced substantially by the larger block size (1.27% to 0.82% and 2.82% to 1.49%, respectively), so f_L is reduced substantially, causing negligible increases in f_B, and resulting in a net improvement in execution time.

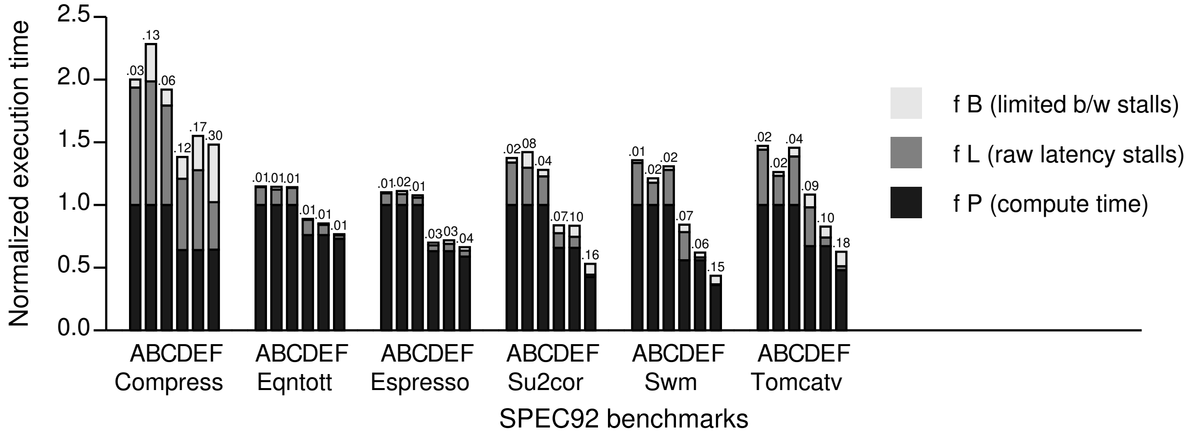


Figure A-1: Execution time breakdown for **E1** (SPEC92)

Adding non-blocking caches to the statically scheduled cores (compare experiments **C** and **A**) had a uniform effect on applications' performance: in each case, a fraction of the memory latency was hidden by overlapping misses, but contention was increased because of queueing. With non-blocking caches, two factors drive f_B in opposite directions: memory requests may become queued behind others for bus access, increasing latency and therefore f_B, but the data transmission portion of the cache miss latency (which contributes directly to f_B) may be tolerated by overlapping it with other requests, thus having a smaller impact on the processor and reducing f_B. For experiments **A** and **C**, the portion of a L1 cache fill attributable to finite bandwidth is six of nineteen cycles[1], which is sufficiently small that the overlapping of transmission time was outweighed by the contention introduced by multiple misses being serviced simultaneously. For every benchmark in this set, therefore, f_B increased slightly, but not as much as f_L was reduced, causing small net reductions in execution time. The total reductions in $f_L + f_B$ were small because—since the cores for experiment **C** were statically scheduled—the non-blocking caches had only small instruction windows (at most two fetch widths) from which to find memory requests that could be overlapped.

Using dynamically scheduled cores with non-blocking caches (compare experiments **D** and **C**) had three effects on execution time decomposition. First, the time required to perform the

1. The L2 lookup accounts for ten processor cycles, and one bus cycle (at three processor cycles each) accounts for each of critical word forwarding, bus arbitration, and three transmission of the rest of the cache line across the cache bus. Of those four latency components, only the last two count toward f_B.

actual computation (T_P) was reduced, on average by about a third. In the graphs, this effect corresponds to a reduction in f_P for experiments **D**-**F**, since we are normalizing all execution times to T_P for experiment **A**. Second, the effect of uncontested memory latencies is better tolerated by the dynamically scheduled core, resulting in 30% to 50% reductions in f_L. Third, the fraction of execution time resulting from memory contention increases in all cases, because of both *absolute* differences (the dynamically scheduled core allows more memory requests to be in the memory system simultaneously), and *relative* differences (execution time is reduced without changing the amount of contention).

The incorporation of tagged prefetching (compare **E** and **D**) causes mixed results. The prefetching increases the L1 miss rates for compress (4.2% to 4.7%) and espresso (0.4% to 0.5%), which results in both f_L and f_B increases, even though the L2 miss rates are improved slightly by the prefetching. For su2cor, the L1 and L2 miss rates are both reduced (3.0% to 2.2% and 3.5% to 0.3%, respectively), but the increases in f_B due to increased contention nullify the reduction in f_L, causing no net change in execution time. This example demonstrates that cache miss ratios can be inaccurate predictors of performance. For swm and tomcatv, however, the prefetching causes large reductions in the miss ratios (1.2% to 0.3% and 3.1% to 1.0% in the L1 caches, respectively), which reduces the f_L component to near-zero in both cases. (Both codes, particularly swm, contain sufficient ILP to tolerate almost all cache miss latencies if the miss rate is sufficiently low). f_B changes only slightly for both codes, as the reductions in misses counterbalance the relative increases in f_B due to decreased execution time.

Finally, a more aggressive processor core (compare **F** and **E**) serves to reduce f_P, reduce f_L, and increase f_B, in all cases. For experiment **F**, f_B is the dominant component of memory stall time (*i.e.*, f_B is larger than f_L) in every case. In Table A-4, we show how the composition of memory stall time shifts from f_L to f_B as we compare a simple, statically scheduled core (experiment **A**) to an aggressive, dynamically scheduled core (experiment **F**) that includes several latency tolerance mechanisms. The shaded cells represent those experiments for which memory stall time accounts for less than 10% of execution time (and are thus unimportant). For the other four benchmarks, significant shifts from f_L to f_B occur.

	Compress		Eqntott		Espresso		Su2cor		Swm		Tomcatv	
Exp.	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B
A	0.936	0.064	0.964	0.036	0.922	0.078	0.903	0.097	0.941	0.059	0.936	0.064
F	0.452	0.548	0.769	0.231	0.628	0.372	0.175	0.825	0.075	0.925	0.216	0.784

Table A-4: Shift from f_L to f_B for **E1**

A.2.2 E2 results

In Figure A-2 we show the execution time breakdown for **E2**. The most notable difference from the comparable results of **E1** is that the total memory stall time is (on average) larger. This effect is caused by three factors: the fact that the SPEC95 data sets are considerably larger than SPEC92 (resulting in higher miss ratios), the longer access times for the L2 cache and memory (twelve cycles versus ten for the L2 cache, and 36 versus 30 cycles per memory access), and the slower off-chip buses (we assume for **E1** and **E2** that the bus is clocked at 100MHz, except for experiment **F** in **E2**, in which the bus is clocked at 150MHz). Vortex has an extremely high L1 instruction cache miss ratio (between 2% and 4% for all experiments), which causes high values for both f_L and f_B, since our microarchitecture assumes that the fetch unit blocks completely on instruction cache misses.

The addition of non-blocking caches for **E2** has a different effect on the time breakdown than it does for **E1**. Like **E1**, execution time is reduced, but unlike **E1**, the non-blocking caches in **E2** cause f_B to be reduced instead of increased. This effect occurs because the non-blocking caches tend to cause higher L1 data cache miss rates, particularly for the more regular (floating-point) codes. For example, the L1 data cache in the Applu experiment has a miss rate of 2.2% for **A** and 4.9% for **C**. The extra misses overcome the most of the reductions in f_L due to the non-blocking cache (in two cases, Li and Su2cor, f_L is actually *increased* by the non-blocking cache). f_B is reduced in these cases because the transmission time is mostly hidden in the latency of the extra misses (the statically scheduled cores exacerbate this effect by preventing other instructions from issuing), and is thus not counted.

Using larger blocks (**B**) has similar effects in **E2** as in **E1**. In all six benchmarks, the larger L1 cache lines result in lower L1 data cache miss rates. As in E1, most of the benchmarks see

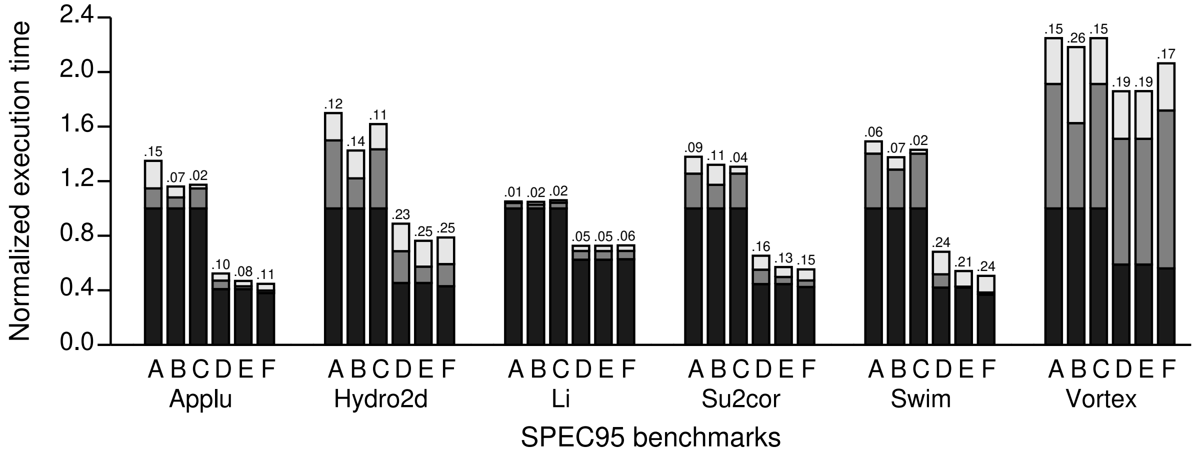


Figure A-2: Execution time breakdown for **E2** (SPEC95)

reduced f_L, with slightly increased f_B (because of greater bus contention, due to more data transfer per miss), resultant in a net decrease in execution time. The only exception is Applu, for which both f_L and f_B are reduced by larger blocks. We do not have a good explanation for why this aberration occurred in applu.

The addition of a dynamically scheduled core also has consistent effects across the six benchmarks in **E2**. In all cases, f_P is reduced, since the dynamically scheduled core can take arithmetic dependences off of the critical path. f_B also increases in every case (because T_B increases in every case, making the bandwidth increase both in absolute and relative terms). The bandwidth increases occur because the dynamically scheduled core allows more operations into the memory system, greatly increasing contention, which overwhelms the effects of the core tolerating the portions of memory delay due to finite bandwidth. T_L is reduced in four of the benchmarks (applu, hydro2D, su2cor, and swim), as the dynamically scheduled core better tolerates memory latencies. For these benchmarks, f_L sometimes increases and sometimes decreases, depending on the absolute change in T_L and the relative effects of the independent changes in f_P and f_B. T_L increases in one case and stays the same in another: in Li, the dynamically scheduled core causes an increase in the L1 data cache miss rate (0.5% to 0.7%). In Vortex, the dynamically scheduled core does not affect the prime component of T_L, the L1 instruction cache miss rate, so the absolute value of T_L remains the same, and the relative component of memory latency stalls (f_L) increases.

Tagged prefetching shows no effect in li or vortex. In li, the cache miss ratio is sufficiently low that the extra traffic caused by prefetches does not cause much additional contention. In Vortex, the memory stalling is due to I-cache misses. Since we only implemented prefetches on the L1 data and L2 caches, the stalls caused by I-cache misses are not affected by prefetching. In hydro2D, the prefetching causes f_B to increase slightly due to extra traffic, but works well enough to reduce f_L, resulting in a net win. In the other three benchmarks (applu, su2cor, and swim), the tagged prefetching is so effective—due to the programs' regularity—that the miss rate is reduced enough to overcome the effect of superfluous prefetches, resulting in reductions in both f_L *and* f_B.

In experiment **F**, we improved the processor core and sped up the processor clock (scaling the off-chip buses but not the memory access latencies). T_B remains unchanged for most of the benchmarks, but since f_P shrinks slightly, f_B increases for most of the benchmarks (applu, li, su2cor, and swim) because the relative size of T_B grows. The exceptions, vortex and hydro2D, are the only two that still have significant f_L components for the aggressive core (the other experiments manage to tolerate most of that latency), and the faster clock increases T_L, increasing f_L even more, and causing f_B to decrease slightly. This result corresponds with our intuition: if the processor clock scales faster than cache and memory bank access times, f_L will grow, and if the processor clock scales faster than bus clocks, f_B will grow. Since the latency tolerance mechanisms seem to almost eliminate f_L in most cases, it would seem that scaling bus clocks (as do Rambus interfaces [30]) is more important than providing fast memory banks.

In Table A-5 we present the relative contributions to memory stall time ($f_L + f_B$) for experiments **A** and **F** in set **E2**. Li is shaded out because its L1 cache miss rate is so low. Vortex shows little change in the distribution between f_B and f_L because its high instruction cache miss ratio is little affected by the latenc4444y tolerance mechanisms and aggressive processor core. The other four benchmarks (applu, hydro2D, su2cor, and swim) all show a significant shift from f_L to f_B, in which f_B is over 50% of memory stall time for all four of these benchmarks with experiment **F**.

	applu		hydro2d		li		su2cor		swim		vortex	
Exp	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B
A	0.421	0.579	0.714	0.286	0.789	0.211	0.674	0.326	0.817	0.183	0.731	0.269
F	0.270	0.730	0.454	0.546	0.600	0.400	0.372	0.628	0.113	0.887	0.770	0.230

Table A-5: Shift from f_L to f_B for **E2**

A.2.3 E3 results

In Figure A-3, we display the execution time breakdown for the updated SPEC95 runs, using a more mature simulator and more up-to-date parameters. Since many of these benchmarks were analyzed in the previous subsection, in this subsection we only describe salient differences in the results.

The most prominent difference between the results from **E2** and **E3** is that f_B is much higher across the board for almost all of the benchmarks in **E3**. This difference occurs for two reasons: (1) the **E3** experiments were run with a higher ratio of processor cycles to bus cycles (5:1 instead of the 4:1 ratio used for **E2**), and (2) we assumed a more aggressive memory hierarchy that had lower L2 cache access latencies (7 cycles instead of 12 for **E2**). The main memory access times were actually slightly larger for **E3** (40 versus 36 cycles), but that small difference is negligible considering the fairly low global L2 miss ratios.

Another effect that we see in **E3** is that the aggressive dynamically scheduled core (much more aggressive than **E1** or **E2**, see Table A-3) causes a larger drop in f_P than occurs in **E1** or **E2**. This larger drop has the effect of amplifying the relative size of the memory stall components, even though the absolute value of T_L is typically reduced by the use of a dynamically scheduled core.

Compress shows different behavior than any other application in any of the experiment sets: most of the memory stall time in each experiment—which is non-negligible—is caused by contention. This result is an artifact of the version of the simulator with which we performed these experiments. The compress input set we used for this experiment set was **train**. According to Table 2-5, the smaller input sets for compress have higher frequencies of stores than is usual (88% of memory operations for **test** were stores, and 45% for **train**, as opposed to 35%

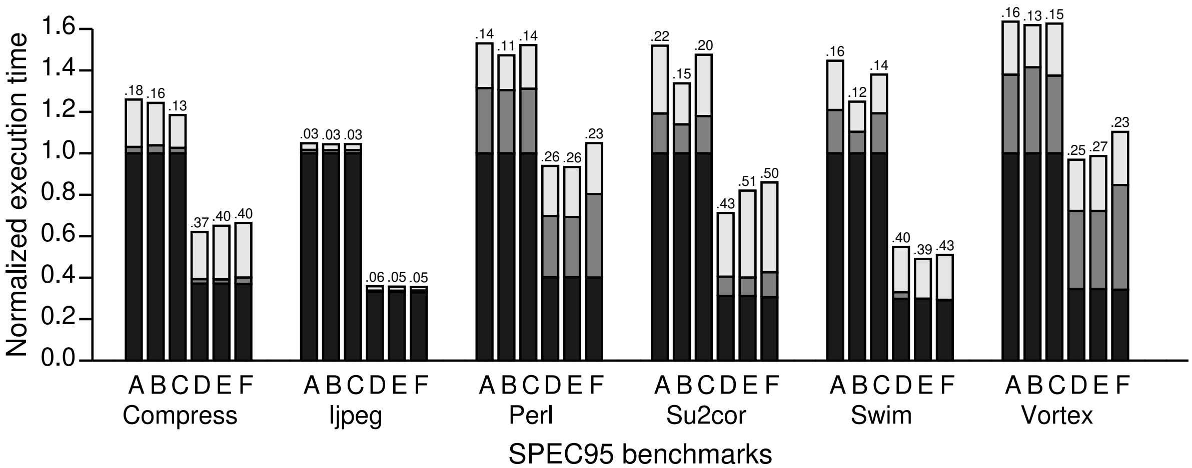


Figure A-3: Execution time breakdown for **E3** (SPEC95)

for **std** and **ref**). The older version of our simulator did not simulate a finite write buffer. Stores could therefore cause cache misses, causing contention that interfered with loads, but never directly stalled the commit stage of the pipeline. High frequencies of stores therefore added to T_B but not T_L. In the newer version of the memory system simulator, write misses will stall the commit stage, exerting back pressure on the execution stage, and eventually stalling it if the frequency or duration of the write misses are sufficiently high (as they are for compress with the **test** or **train** inputs). For these results, however, the older simulator measured an optimistically low T_L.

Perl displays an effect similar to vortex: high L1 instruction cache miss ratios (1.6% for **A**) cause a high memory stall component that is unmitigated by the latency tolerance optimizations that we implemented (except for the larger cache blocks, which reduced the I-cache miss ratio to 1.1%).

In Table A-6, we list the effects that going from **A** to **F** have on the memory stall time distribution for **E3**. Ijpeg is shaded because its cache miss rates are too low for the memory stall time distribution to be meaningful (since both f_L and f_B are negligible). Perl and vortex actually show a reduction in the fraction of memory stall time attributable to f_B when comparing **F** to **A**. This reduction occurs because both benchmarks have high f_L components due to high instruction cache miss rates. When the clock rate is increased for experiment **F**, T_B changes little, but T_L increases since L2 and memory instruction fetches become more expensive. This in

	compress		ijpeg		perl		su2cor		swim		vortex	
Exp	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B
A	0.120	0.880	0.348	0.652	0.594	0.406	0.371	0.629	0.468	0.532	0.598	0.402
F	0.106	0.894	0.350	0.650	0.620	0.380	0.218	0.782	0.008	0.992	0.663	0.337

Table A-6: Shift from f_L to f_B for **E3**

turn increases f_L, which decreases f_B. Su2cor and swim show significant increases in the f_B component of memory stall time. For experiment **F**, swim spends almost all of its memory stall time in bandwidth stalls. This aberration occurs because the large core can exploit enough ILP in swim to fully tolerate almost all memory latencies in the absence of memory contention, making T_L negligible. Because of contention, however, memory stall time mushrooms to 43% of execution time, nearly all of which results from finite bandwidth.

A.3 Summary

Our results show that limited bandwidth and contention in the memory system can cause serious performance degradation in processor performance. For smaller (SPEC92) benchmarks running on less aggressive processors, the fraction of time spent in bandwidth stalls averaged 14%. For slightly larger applications (even using their small data sets) running on highly aggressive processor, this fraction swelled to over 34%, on average.

Two factors contribute to these large bandwidth stalls. The success of processor cores and latency tolerance techniques at reducing computation time and raw memory latency stalls, respectively, increases the bandwidth stalls as a relative component of execution time. Also, the presence of so many memory operations existing in the memory hierarchy simultaneously—for the more aggressive processor models—causes contention to increase, further contributing to bandwidth stalls.

We see three classes of application behavior in these experiments. The first class is *processor-bound* applications: these are applications that have such low cache miss ratios that they are dominated by f_P. Eqntott and espresso in **E1**, li in **E2**, and ijpeg in **E3** are all examples of this class of applications. To improve performance for these applications, better processing cores and faster clocks are the only hardware solution. The second application class we call *instruction-bound* applications; these are applications for which high instruction cache rates are the performance bottleneck. Perl and vortex are examples of this class of applications. To improve performance for these applications, instruction cache performance must be improved, whether with larger instruction caches, trace caches [98], or instruction prefetching schemes [58]. The third class of applications is *bandwidth-bound* applications, into which all the other benchmarks we measured in these studies fall. ILP processor cores and sophisticated latency tolerance techniques make these programs progressively more bandwidth-bound as these techniques are pursued more aggressively. Many research efforts are underway to improve the performance of the first two classes of applications. It is on the third class that we focused in this dissertation.

Appendix B

Cache performance of SPEC95

B.1 Set associativity

benchmark	assoc.	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
099.go	1	28.007	21.403	9.971	5.468	3.035	1.681	1.481	0.001	0.000
	2	------	15.339	8.781	2.749	0.913	0.356	0.025	0.000	0.000
	4	------	------	5.955	2.409	0.532	0.066	0.008	0.000	0.000
	8	------	------	------	1.892	0.477	0.032	0.001	0.000	0.000
124.m88ksim	1	4.546	2.564	1.522	0.904	0.426	0.141	0.132	0.007	------
	2	------	0.653	0.297	0.165	0.061	0.025	0.008	0.007	------
	4	------	------	0.099	0.061	0.021	0.009	0.008	0.007	------
	8	------	------	------	0.049	0.012	0.009	0.008	0.007	------
126.gcc	1	7.951	5.146	3.265	1.975	1.043	0.619	0.359	0.128	0.064
	2	------	3.223	1.848	1.051	0.575	0.312	0.145	0.055	0.015
	4	------	------	1.435	0.818	0.469	0.283	0.129	0.040	0.013
	8	------	------	------	0.781	0.444	0.279	0.124	0.036	0.012
129.compress	1	5.617	5.519	5.466	5.427	5.380	5.162	1.113	0.369	------
	2	------	5.367	5.337	5.320	5.304	5.191	1.464	0.351	------
	4	------	------	5.333	5.315	5.301	5.216	2.063	0.351	------
	8	------	------	------	5.315	5.300	5.228	3.216	0.351	------
130.li	1	3.829	2.241	1.127	0.476	0.016	0.000	0.000	------	------
	2	------	1.083	0.555	0.192	0.012	0.000	0.000	------	------
	4	------	------	0.483	0.215	0.000	0.000	0.000	------	------
	8	------	------	------	0.234	0.000	0.000	0.000	------	------
132.ijpeg	1	9.607	3.577	1.843	0.826	0.552	0.360	0.278	0.233	0.217
	2	------	1.942	0.671	0.338	0.205	0.100	0.047	0.042	0.042
	4	------	------	0.492	0.265	0.199	0.098	0.044	0.042	0.042
	8	------	------	------	0.251	0.202	0.101	0.042	0.042	0.042
134.perl	1	5.688	3.145	2.150	1.679	0.801	0.495	0.257	0.205	0.165
	2	------	1.719	1.055	0.590	0.515	0.370	0.209	0.174	0.155
	4	------	------	0.569	0.458	0.423	0.376	0.214	0.175	0.155
	8	------	------	------	0.441	0.423	0.381	0.226	0.175	0.156
147.vortex	1	6.955	5.103	3.141	1.464	0.922	0.519	0.318	0.215	0.133
	2	------	2.674	1.805	1.009	0.570	0.308	0.194	0.129	0.086
	4	------	------	1.468	0.840	0.439	0.258	0.156	0.100	0.073
	8	------	------	------	0.730	0.402	0.228	0.149	0.095	0.071

Table B-1: Miss rates for varied associativities on the SPECINT95 data stream

benchmark	assoc.	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
101.tomcatv	1	8.955	7.561	4.275	1.933	1.175	1.157	1.145	1.137	1.126
	2	------	5.626	4.446	0.929	0.390	0.365	0.361	0.356	0.347
	4	------	------	3.647	1.075	0.363	0.362	0.361	0.358	0.353
	8	------	------	------	1.105	0.363	0.362	0.361	0.358	0.353
102.swim	1	49.698	39.780	21.024	6.658	2.015	1.989	1.976	1.968	1.960
	2	------	38.302	23.768	3.296	1.963	1.943	1.943	1.942	1.940
	4	------	------	24.958	3.391	1.956	1.676	1.675	1.674	1.671
	8	------	------	------	3.988	1.956	1.676	1.675	1.674	1.671
103.su2cor	1	10.110	8.058	7.279	6.693	2.350	1.883	1.372	0.640	0.286
	2	------	2.913	2.440	2.294	2.136	1.742	1.292	0.460	0.199
	4	------	------	2.107	1.977	1.883	1.761	1.329	0.443	0.180
	8	------	------	------	1.931	1.692	1.527	1.358	0.447	0.168
104.hydro2d	1	5.203	4.258	3.539	2.880	2.728	2.660	2.636	2.523	2.289
	2	------	3.250	3.001	2.662	2.594	2.587	2.583	2.562	2.332
	4	------	------	2.910	2.631	2.584	2.583	2.582	2.565	2.389
	8	------	------	------	2.644	2.584	2.584	2.582	2.567	2.400
107.mgrid	1	5.934	2.620	1.865	1.457	1.235	0.966	0.901	0.596	0.566
	2	------	1.224	1.001	0.967	0.933	0.775	0.602	0.572	0.551
	4	------	------	0.994	0.977	0.932	0.918	0.603	0.575	0.548
	8	------	------	------	0.975	0.932	0.904	0.601	0.581	0.545
110.applu	1	5.092	2.630	1.913	1.573	1.380	1.266	1.226	1.184	1.098
	2	------	1.560	1.280	1.234	1.222	1.217	1.200	1.156	1.085
	4	------	------	1.255	1.219	1.217	1.215	1.204	1.155	1.086
	8	------	------	------	1.218	1.217	1.215	1.207	1.141	1.098
125.turb3d	1	4.065	3.461	3.255	2.158	1.364	1.271	0.871	0.394	0.386
	2	------	2.584	2.306	2.072	1.234	1.166	0.883	0.379	0.377
	4	------	------	1.843	1.727	1.040	0.934	0.932	0.378	0.374
	8	------	------	------	1.190	0.578	0.394	0.394	0.378	0.374
141.apsi	1	6.995	5.911	5.646	4.450	2.943	1.673	0.816	0.056	0.001
	2	------	2.970	2.732	2.611	2.130	1.478	0.381	0.021	0.000
	4	------	------	2.074	2.021	1.677	0.394	0.223	0.008	0.000
	8	------	------	------	2.002	1.739	0.388	0.158	0.011	0.000
145.fpppp	1	5.638	4.334	3.726	2.986	2.921	2.823	0.000	------	------
	2	------	1.536	0.703	0.379	0.072	0.045	0.000	------	------
	4	------	------	0.242	0.065	0.014	0.000	0.000	------	------
	8	------	------	------	0.054	0.008	0.000	0.000	------	------
146.wave5	1	24.882	21.038	12.873	7.568	1.888	1.057	0.824	0.680	0.610
	2	------	20.266	13.995	6.446	1.234	0.700	0.438	0.315	0.249
	4	------	------	15.304	6.327	1.245	0.606	0.384	0.283	0.219
	8	------	------	------	6.448	1.293	0.613	0.361	0.285	0.216

Table B-2: Miss rates for varied associativities on the SPECFP95 data stream

B.2 Block size

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	16.270	10.366	6.159	3.605	2.064	1.064	0.835	0.002	0.001
32B	20.450	13.322	7.539	4.218	2.377	1.291	1.092	0.002	0.000
64B	28.007	21.403	9.971	5.468	3.035	1.681	1.481	0.001	0.000
128B	32.791	26.280	12.884	7.049	3.957	2.283	2.049	0.001	0.000
256B	37.808	30.704	16.867	9.378	5.274	3.098	2.804	0.002	0.000
512B	44.510	36.112	22.459	13.202	7.460	4.290	3.923	0.003	0.000
1024B	53.363	43.870	30.361	18.400	10.885	6.135	5.521	0.004	0.000
2048B	------	50.711	37.615	25.455	14.553	8.325	7.045	0.232	0.000
4096B	------	------	47.549	32.105	20.485	12.932	9.439	0.483	0.000
8192B	------	------	------	41.768	30.578	21.073	11.938	1.307	0.000
instruction request stream									
16B	21.629	18.027	13.931	8.220	2.821	0.740	0.176	0.001	0.000
32B	11.917	9.905	7.697	4.572	1.580	0.392	0.094	0.001	0.000
64B	6.912	5.758	4.490	2.673	0.924	0.215	0.051	0.000	0.000
128B	4.301	3.564	2.784	1.676	0.561	0.122	0.028	0.000	0.000
256B	2.838	2.337	1.816	1.131	0.368	0.073	0.017	0.000	0.000
512B	2.236	1.768	1.322	0.837	0.270	0.045	0.010	0.000	0.000
1024B	1.979	1.518	1.093	0.679	0.237	0.031	0.006	0.000	0.000
2048B	------	1.476	1.052	0.709	0.182	0.022	0.004	0.000	0.000
4096B	------	------	1.225	0.856	0.215	0.022	0.004	0.000	0.000
8192B	------	------	------	1.031	0.263	0.026	0.006	0.000	0.000
unified instruction and data stream									
16B	26.504	21.372	16.477	10.256	5.176	2.468	1.335	0.536	0.534
32B	18.964	14.976	11.260	7.177	3.925	1.923	1.167	0.439	0.437
64B	15.801	12.620	8.841	5.726	3.350	1.666	1.091	0.369	0.368
128B	14.549	11.599	7.800	5.063	3.087	1.562	1.071	0.306	0.305
256B	15.279	12.108	7.870	5.100	3.257	1.692	1.181	0.251	0.250
512B	17.902	13.670	9.233	5.792	3.795	1.948	1.387	0.216	0.215
1024B	22.485	16.488	11.506	7.206	4.822	2.552	1.783	0.204	0.203
2048B	------	22.456	15.569	10.007	6.872	3.788	2.333	0.312	0.260
4096B	------	------	------	------	------	------	------	------	------
8192B	------	------	------	20.390	15.210	9.215	4.838	1.332	1.034

Table B-3: Cache miss rates for 099.go, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	3.773	1.807	1.182	0.549	0.275	0.123	0.117	0.028	------
32B	4.189	2.157	1.387	0.713	0.337	0.120	0.116	0.014	------
64B	4.546	2.564	1.522	0.904	0.426	0.141	0.132	0.007	------
128B	6.394	3.934	1.800	0.976	0.496	0.203	0.183	0.004	------
256B	10.353	6.730	2.959	1.687	0.762	0.291	0.272	0.002	------
512B	14.294	8.890	3.786	2.055	1.087	0.488	0.469	0.001	------

Table B-4: Cache miss rates for 124.m88ksim, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
1024B	19.284	12.056	5.159	2.621	1.498	0.732	0.711	0.001	------
2048B	------	15.320	7.197	3.837	2.420	1.292	1.251	0.000	------
4096B	------	------	12.952	6.783	3.827	1.872	1.824	0.000	------
8192B	------	------	------	8.743	5.751	3.013	2.935	0.000	------
instruction request stream									
16B	30.053	23.435	15.011	8.854	4.063	0.003	0.002	0.002	------
32B	19.390	15.153	10.078	6.117	3.167	0.002	0.001	0.001	------
64B	13.431	10.782	7.472	4.655	2.208	0.001	0.001	0.000	------
128B	8.953	6.971	4.941	2.957	1.299	0.001	0.000	0.000	------
256B	6.243	5.124	3.468	2.137	1.075	0.000	0.000	0.000	------
512B	4.770	4.006	3.112	2.126	1.419	0.000	0.000	0.000	------
1024B	3.764	3.142	2.556	1.839	0.984	0.000	0.000	0.000	------
2048B	------	2.761	2.185	1.517	0.767	0.000	0.000	0.000	------
4096B	------	------	2.229	1.693	0.655	0.000	0.000	0.000	------
8192B	------	------	------	1.657	0.856	0.009	0.000	0.000	------
unified instruction and data stream									
16B	29.924	22.802	15.009	9.777	4.061	0.060	0.041	0.018	------
32B	20.396	15.456	10.528	7.190	3.130	0.057	0.040	0.015	------
64B	14.566	11.420	7.726	5.317	2.225	0.058	0.042	0.011	------
128B	11.137	8.493	5.939	3.746	1.530	0.076	0.054	0.010	------
256B	10.294	7.609	5.454	3.381	1.300	0.112	0.078	0.013	------
512B	11.352	8.477	5.926	4.024	1.743	0.228	0.127	0.014	------
1024B	16.410	10.468	6.303	3.994	1.647	0.318	0.189	0.016	------
2048B	------	15.047	9.594	6.771	4.179	2.287	0.359	0.057	------
4096B	------	------	16.938	13.503	10.881	8.532	3.688	3.250	------
8192B	------	------	------	16.075	12.377	9.603	4.409	3.707	------

Table B-4: Cache miss rates for 124.m88ksim, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	8.809	6.129	4.155	2.702	1.639	1.027	0.620	0.255	0.144
32B	7.880	5.208	3.349	2.100	1.242	0.767	0.451	0.173	0.093
64B	7.951	5.146	3.265	1.975	1.043	0.619	0.359	0.128	0.064
128B	8.930	5.696	3.560	1.974	0.982	0.560	0.322	0.109	0.052
256B	11.060	7.106	4.304	2.305	1.141	0.614	0.360	0.111	0.050
512B	14.911	9.588	5.913	3.231	1.508	0.772	0.453	0.135	0.060
1024B	------	------	------	------	------	------	------	------	------
2048B	------	------	------	------	------	------	------	------	------
4096B	------	------	------	------	------	------	------	------	------
8192B	------	------	------	------	------	------	------	------	------
instruction request stream									
16B	21.451	16.574	11.878	7.341	4.218	1.739	1.229	0.475	0.165
32B	12.812	9.994	7.303	4.595	2.651	1.094	0.779	0.295	0.107
64B	8.211	6.492	4.837	3.090	1.766	0.752	0.535	0.194	0.077
128B	5.716	4.561	3.482	2.291	1.319	0.572	0.418	0.149	0.061
256B	4.121	3.323	2.591	1.775	1.041	0.452	0.337	0.121	0.055
512B	3.292	2.584	2.043	1.466	0.927	0.435	0.321	0.106	0.050

Table B-5: Cache miss rates for 026.gcc, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
1024B	------	------	------	------	------	------	------	------	------
2048B	------	------	------	------	------	------	------	------	------
4096B	------	------	------	------	------	------	------	------	------
8192B	------	------	------	------	------	------	------	------	------
unified instruction and data stream									
16B	22.185	17.082	12.468	8.145	4.920	2.443	1.357	0.609	0.296
32B	15.075	11.486	8.382	5.520	3.344	1.695	0.919	0.404	0.203
64B	11.719	8.721	6.313	4.177	2.497	1.314	0.683	0.294	0.155
128B	10.712	7.627	5.429	3.552	2.132	1.130	0.576	0.244	0.130
256B	11.775	7.823	5.327	3.467	2.074	1.089	0.525	0.221	0.120
512B	15.170	9.531	6.148	3.923	2.410	1.267	0.598	0.254	0.146
1024B	------	------	------	------	------	------	------	------	------
2048B	------	------	------	------	------	------	------	------	------
4096B	------	------	------	------	------	------	------	------	------
8192B	------	------	------	------	------	------	------	------	------

Table B-5: Cache miss rates for 026.gcc, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	20.961	20.883	20.840	20.806	20.764	20.336	3.857	1.446	------
32B	10.698	10.606	10.564	10.535	10.492	10.223	2.033	0.728	------
64B	5.617	5.519	5.466	5.427	5.380	5.162	1.113	0.369	------
128B	3.347	3.223	3.010	2.963	2.903	2.625	0.639	0.190	------
256B	2.857	2.070	1.778	1.697	1.621	1.357	0.394	0.102	------
512B	2.797	1.875	1.454	1.166	0.985	0.727	0.263	0.074	------
1024B	16.410	10.468	6.303	3.994	1.647	0.318	0.189	0.016	------
2048B	------	15.047	9.594	6.771	4.179	2.287	0.359	0.057	------
4096B	------	------	16.938	13.503	10.881	8.532	3.688	3.250	------
8192B	------	------	------	16.075	12.377	9.603	4.409	3.707	------
instruction request stream									
16B	2.190	0.807	0.567	0.165	0.101	0.049	0.049	0.049	------
32B	1.632	0.515	0.354	0.097	0.061	0.028	0.028	0.028	------
64B	1.193	0.364	0.244	0.064	0.038	0.016	0.016	0.016	------
128B	0.871	0.267	0.177	0.046	0.025	0.009	0.009	0.009	------
256B	0.590	0.192	0.119	0.036	0.019	0.005	0.005	0.005	------
512B	0.516	0.145	0.092	0.030	0.015	0.003	0.003	0.003	------
1024B	16.410	10.468	6.303	3.994	1.647	0.318	0.189	0.016	------
2048B	------	15.047	9.594	6.771	4.179	2.287	0.359	0.057	------
4096B	------	------	16.938	13.503	10.881	8.532	3.688	3.250	------
8192B	------	------	------	16.075	12.377	9.603	4.409	3.707	------
unified instruction and data stream									
16B	10.994	9.409	8.805	8.544	8.459	8.248	2.144	0.682	------
32B	6.946	5.315	4.638	4.413	4.325	4.181	1.156	0.354	------
64B	5.202	3.432	2.666	2.402	2.286	2.154	0.659	0.190	------
128B	5.220	2.981	1.909	1.521	1.340	1.156	0.408	0.109	------
256B	6.693	3.553	1.858	1.222	0.925	0.693	0.289	0.068	------
512B	10.912	5.569	2.722	1.481	0.908	0.551	0.266	0.052	------

Table B-6: Cache miss rates for 129.compress, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
1024B	20.175	10.065	4.929	2.425	1.356	0.755	0.439	0.145	------
2048B	------	20.685	9.457	4.531	2.491	1.273	0.747	0.256	------
4096B	------	------	18.201	8.643	4.653	2.296	1.297	0.366	------
8192B	------	------	------	16.840	8.802	4.331	2.276	0.462	------

Table B-6: Cache miss rates for 129.compress, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	5.088	3.651	2.178	0.993	0.034	0.001	0.001	------	------
32B	4.080	2.685	1.446	0.640	0.022	0.001	0.001	------	------
64B	3.829	2.241	1.127	0.476	0.016	0.000	0.000	------	------
128B	4.072	2.212	1.014	0.405	0.014	0.000	0.000	------	------
256B	5.311	2.878	1.478	0.592	0.018	0.000	0.000	------	------
512B	7.583	4.145	2.228	0.712	0.020	0.000	0.000	------	------
1024B	14.435	8.824	3.884	0.984	0.049	0.000	0.000	------	------
2048B	------	19.512	14.353	2.941	1.408	0.000	0.000	------	------
4096B	------	------	22.624	6.141	3.766	0.000	0.000	------	------
8192B	------	------	------	10.858	6.089	0.000	0.000	------	------
instruction request stream									
16B	14.666	7.401	1.762	1.626	0.154	0.000	0.000	------	------
32B	9.483	4.867	1.214	1.120	0.124	0.000	0.000	------	------
64B	5.674	3.037	0.869	0.802	0.098	0.000	0.000	------	------
128B	3.814	2.265	0.694	0.625	0.073	0.000	0.000	------	------
256B	2.991	1.794	0.523	0.457	0.080	0.000	0.000	------	------
512B	2.982	2.059	0.817	0.717	0.379	0.000	0.000	------	------
1024B	2.820	1.925	0.783	0.682	0.379	0.000	0.000	------	------
2048B	------	2.145	1.140	1.026	0.539	0.000	0.000	------	------
4096B	------	------	1.321	1.175	0.586	0.000	0.000	------	------
8192B	------	------	------	1.159	0.472	0.000	0.000	------	------
unified instruction and data stream									
16B	16.799	9.233	4.252	2.591	1.010	0.064	0.063	------	------
32B	12.438	6.884	3.204	1.851	0.726	0.043	0.043	------	------
64B	9.103	5.279	2.722	1.448	0.589	0.032	0.031	------	------
128B	8.328	5.144	2.724	1.383	0.662	0.028	0.028	------	------
256B	9.590	6.588	3.469	1.582	0.872	0.020	0.020	------	------
512B	14.342	9.344	5.513	2.613	1.656	0.026	0.026	------	------
1024B	22.374	15.369	8.668	4.859	3.655	0.025	0.025	------	------
2048B	------	26.398	17.995	8.266	6.662	0.036	0.036	------	------
4096B	------	------	30.784	15.098	11.760	0.089	0.088	------	------
8192B	------	------	------	27.286	17.566	0.394	0.386	------	------

Table B-7: Cache miss rates for 130.li, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	9.868	4.837	2.821	1.600	1.122	0.705	0.517	0.426	0.387
32B	9.072	3.769	2.082	1.059	0.721	0.453	0.335	0.273	0.248
64B	9.607	3.577	1.843	0.826	0.552	0.360	0.278	0.233	0.217
128B	11.942	4.592	2.150	0.845	0.569	0.389	0.319	0.279	0.266
256B	16.031	7.063	3.138	1.115	0.759	0.532	0.452	0.406	0.392
512B	21.581	11.593	5.419	1.823	1.242	0.890	0.765	0.689	0.667
1024B	29.368	18.975	10.243	3.508	2.497	1.795	1.566	1.443	1.400
2048B	------	22.030	12.857	4.544	3.137	2.159	1.821	1.653	1.589
4096B	------	------	17.286	8.251	5.056	2.489	1.831	1.468	1.335
8192B	------	------	------	13.007	7.579	3.659	2.386	1.658	1.454
instruction request stream									
16B	1.629	1.170	0.776	0.328	0.129	0.067	0.004	0.001	0.001
32B	0.907	0.644	0.430	0.186	0.074	0.039	0.002	0.001	0.001
64B	0.535	0.373	0.247	0.107	0.046	0.025	0.002	0.000	0.000
128B	0.332	0.232	0.156	0.065	0.028	0.016	0.001	0.000	0.000
256B	0.230	0.154	0.106	0.041	0.018	0.011	0.001	0.000	0.000
512B	0.177	0.109	0.072	0.028	0.011	0.007	0.001	0.000	0.000
1024B	0.151	0.086	0.053	0.024	0.008	0.006	0.001	0.000	0.000
2048B	------	0.094	0.051	0.023	0.006	0.004	0.001	0.000	0.000
4096B	------	------	0.060	0.032	0.012	0.007	0.001	0.000	0.000
8192B	------	------	------	0.035	0.015	0.009	0.001	0.000	0.000
unified instruction and data stream									
16B	6.934	4.274	2.605	1.536	1.132	0.514	0.253	0.161	0.131
32B	5.510	3.003	1.789	1.033	0.741	0.341	0.163	0.104	0.082
64B	5.075	2.455	1.405	0.784	0.538	0.263	0.126	0.082	0.065
128B	5.737	2.585	1.407	0.757	0.494	0.262	0.123	0.085	0.070
256B	7.330	3.242	1.678	0.857	0.558	0.328	0.148	0.108	0.092
512B	11.069	5.489	2.522	1.220	0.776	0.500	0.230	0.170	0.149
1024B	17.266	9.534	4.944	2.473	1.643	1.166	0.460	0.355	0.314
2048B	------	14.343	7.231	3.822	2.480	1.780	0.633	0.470	0.410
4096B	------	------	11.778	6.333	4.106	2.303	0.732	0.498	0.384
8192B	------	------	------	11.038	6.345	3.586	1.318	0.638	0.456

Table B-8: Cache miss rates for 132.ijpeg, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	6.280	4.258	3.411	3.033	1.772	1.289	0.687	0.603	0.514
32B	5.501	3.453	2.554	2.180	1.101	0.749	0.397	0.338	0.283
64B	5.688	3.145	2.150	1.679	0.801	0.495	0.257	0.205	0.165
128B	6.940	3.701	2.242	1.588	0.679	0.391	0.198	0.144	0.108
256B	12.152	5.894	3.882	2.950	0.744	0.395	0.197	0.123	0.084
512B	14.912	7.997	5.285	3.714	1.090	0.489	0.238	0.139	0.085
1024B	18.885	11.358	7.769	5.677	2.273	0.770	0.349	0.199	0.114

Table B-9: Cache miss rates for 134.perl, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	18.032	12.544	9.290	4.588	2.673	0.566	0.303	0.175
4096B	------	------	16.775	12.864	7.737	5.159	0.968	0.483	0.281
8192B	------	------	------	23.074	11.925	7.846	1.685	0.869	0.481
instruction request stream									
16B	18.974	11.271	7.493	5.343	2.238	1.873	0.111	0.000	0.000
32B	12.231	7.579	5.007	3.669	1.622	1.324	0.061	0.000	0.000
64B	8.436	5.201	3.608	2.553	1.111	0.920	0.046	0.000	0.000
128B	6.140	4.411	3.302	2.598	1.056	0.897	0.060	0.000	0.000
256B	5.077	3.707	2.819	2.276	0.835	0.689	0.060	0.000	0.000
512B	4.272	3.240	2.455	2.105	0.807	0.609	0.052	0.000	0.000
1024B	3.965	3.021	2.197	1.759	0.662	0.516	0.066	0.000	0.000
2048B	------	2.735	2.072	1.687	0.649	0.386	0.082	0.000	0.000
4096B	------	------	2.555	1.814	0.908	0.652	0.362	0.000	0.000
8192B	------	------	------	2.091	1.293	0.917	0.497	0.000	0.000
unified instruction and data stream									
16B	19.935	12.558	8.602	6.469	3.638	3.010	1.358	0.286	0.222
32B	13.999	8.970	5.965	4.494	2.582	2.086	0.879	0.178	0.138
64B	11.489	7.192	4.585	3.361	1.913	1.534	0.650	0.126	0.096
128B	11.459	6.949	4.534	3.530	1.810	1.425	0.596	0.109	0.081
256B	15.128	8.184	5.119	4.010	1.870	1.459	0.764	0.102	0.069
512B	23.130	10.209	6.391	4.540	2.279	1.583	0.915	0.124	0.077
1024B	29.887	15.105	11.108	8.475	5.674	4.704	1.873	0.183	0.108
2048B	------	21.306	15.592	11.834	7.911	6.501	2.056	0.270	0.145
4096B	------	------	24.558	17.095	12.428	10.684	5.755	0.525	0.209
8192B	------	------	------	26.464	18.033	14.470	8.087	1.042	0.533

Table B-9: Cache miss rates for 134.perl, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	6.215	4.947	3.124	1.568	1.028	0.651	0.451	0.345	0.254
32B	6.433	5.043	3.125	1.479	0.938	0.566	0.363	0.260	0.177
64B	6.955	5.103	3.141	1.464	0.922	0.519	0.318	0.215	0.133
128B	11.992	5.646	3.606	1.880	1.244	0.621	0.374	0.241	0.119
256B	14.353	6.564	4.082	2.373	1.627	0.705	0.423	0.259	0.131
512B	18.936	10.572	5.787	3.358	2.441	0.850	0.501	0.293	0.147
1024B	28.702	16.055	8.631	5.377	3.638	1.745	0.604	0.365	0.184
2048B	------	21.186	11.778	7.207	4.676	2.304	0.837	0.493	0.239
4096B	------	------	14.684	9.367	6.154	3.401	1.291	0.781	0.351
8192B	------	------	------	15.966	12.122	5.028	2.141	1.389	0.812
instruction request stream									
16B	31.257	17.651	10.732	5.338	2.987	1.663	0.354	0.149	0.000
32B	18.821	10.508	6.445	3.195	1.780	0.970	0.238	0.092	0.000
64B	11.582	6.864	4.363	2.195	1.279	0.743	0.165	0.061	0.000
128B	8.211	4.926	3.099	1.618	0.996	0.605	0.120	0.040	0.000
256B	6.401	3.762	2.485	1.245	0.844	0.532	0.101	0.029	0.000
512B	4.832	3.026	2.023	1.065	0.713	0.439	0.085	0.027	0.000
1024B	4.096	2.779	1.883	1.029	0.750	0.436	0.088	0.025	0.000

Table B-10: Cache miss rates for 147.vortex, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	3.182	1.986	1.087	0.766	0.453	0.089	0.024	0.000
4096B	------	------	2.680	1.661	0.991	0.555	0.108	0.032	0.000
8192B	------	------	------	2.006	1.381	0.822	0.328	0.041	0.000
unified instruction and data stream									
16B	27.096	16.806	10.982	6.128	3.874	2.042	0.690	0.352	0.140
32B	18.165	11.595	7.808	4.357	2.736	1.396	0.537	0.264	0.106
64B	13.775	8.982	6.128	3.564	2.309	1.180	0.445	0.213	0.084
128B	14.247	8.312	5.690	3.533	2.432	1.175	0.430	0.202	0.075
256B	14.813	9.077	5.461	3.446	2.439	1.250	0.437	0.197	0.079
512B	17.524	10.997	6.298	4.036	2.928	1.359	0.500	0.239	0.088
1024B	27.743	16.685	9.314	6.784	5.139	2.021	0.635	0.282	0.116
2048B	------	25.753	17.032	12.335	10.373	2.882	1.004	0.399	0.151
4096B	------	------	25.289	18.088	14.821	3.899	1.722	0.735	0.219
8192B	------	------	------	28.328	23.080	7.109	4.608	1.523	0.437

Table B-10: Cache miss rates for 147.vortex, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	7.976	6.722	3.156	2.234	2.144	2.129	2.115	2.096	2.060
32B	8.361	7.072	3.324	1.690	1.495	1.479	1.468	1.457	1.438
64B	8.955	7.561	4.275	1.933	1.175	1.157	1.145	1.137	1.126
128B	12.101	10.301	5.059	2.122	1.034	1.004	0.989	0.979	0.971
256B	13.357	11.487	5.717	2.290	1.006	0.950	0.921	0.906	0.897
512B	16.897	14.215	6.034	2.432	1.048	0.977	0.887	0.869	0.859
1024B	18.262	14.930	6.751	2.884	1.208	1.074	0.899	0.865	0.848
2048B	------	21.066	9.900	5.708	3.837	3.582	0.951	0.886	0.853
4096B	------	------	15.929	6.599	4.368	3.873	1.068	0.943	0.881
8192B	------	------	------	9.906	6.037	5.194	1.780	1.568	1.461
instruction request stream									
16B	22.869	18.103	12.350	7.068	2.143	0.094	0.000	0.000	0.000
32B	13.489	11.040	7.765	4.639	1.344	0.075	0.000	0.000	0.000
64B	7.951	6.703	4.854	3.052	0.874	0.075	0.000	0.000	0.000
128B	5.097	4.337	3.248	2.197	0.621	0.094	0.000	0.000	0.000
256B	3.445	2.976	2.263	1.681	0.479	0.075	0.000	0.000	0.000
512B	2.300	1.981	1.456	1.089	0.385	0.075	0.000	0.000	0.000
1024B	2.159	1.727	1.333	0.948	0.376	0.094	0.000	0.000	0.000
2048B	------	1.821	1.155	0.779	0.282	0.056	0.000	0.000	0.000
4096B	------	------	1.117	0.732	0.300	0.038	0.000	0.000	0.000
8192B	------	------	------	0.788	0.394	0.056	0.000	0.000	0.000
unified instruction and data stream									
16B	22.155	17.770	12.438	7.540	3.295	0.949	0.570	0.558	0.546
32B	14.460	11.938	8.510	5.285	2.280	0.710	0.403	0.394	0.386
64B	10.245	8.562	6.112	3.959	1.667	0.600	0.320	0.312	0.306
128B	9.159	7.504	4.996	3.225	1.446	0.563	0.281	0.272	0.267
256B	9.228	6.973	4.594	2.977	1.300	0.567	0.266	0.254	0.248
512B	10.968	7.641	4.223	2.678	1.175	0.501	0.263	0.248	0.240
1024B	16.882	8.551	4.627	2.897	1.363	0.702	0.306	0.253	0.240

Table B-11: Cache miss rates for 101.tomcatv, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	15.825	7.081	4.487	2.854	1.399	0.367	0.296	0.274
4096B	------	------	12.987	6.775	4.376	1.723	0.613	0.370	0.332
8192B	------	------	------	13.159	9.039	6.050	4.750	0.557	0.494

Table B-11: Cache miss rates for 101.tomcatv, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	45.199	31.166	8.442	7.393	6.880	6.871	6.865	6.859	6.839
32B	47.764	35.429	13.544	3.900	3.624	3.610	3.603	3.596	3.585
64B	49.698	39.780	21.024	6.658	2.015	1.989	1.976	1.968	1.960
128B	50.823	42.673	25.778	8.596	1.246	1.196	1.171	1.158	1.149
256B	51.594	44.320	28.323	9.689	0.934	0.836	0.787	0.762	0.749
512B	52.356	45.481	29.900	10.467	0.915	0.724	0.629	0.581	0.557
1024B	54.204	46.792	31.325	11.334	1.187	0.809	0.620	0.526	0.478
2048B	------	49.780	34.317	12.875	1.912	1.157	0.781	0.592	0.498
4096B	------	------	44.000	23.329	12.693	11.429	10.797	10.482	10.324
8192B	------	------	------	42.222	35.214	33.084	32.382	32.031	31.855
instruction request stream									
16B	3.162	2.170	0.809	0.001	0.001	0.001	0.001	0.001	0.001
32B	1.674	1.178	0.436	0.001	0.001	0.000	0.000	0.000	0.000
64B	1.302	0.806	0.435	0.001	0.000	0.000	0.000	0.000	0.000
128B	0.868	0.620	0.372	0.000	0.000	0.000	0.000	0.000	0.000
256B	0.868	0.620	0.372	0.000	0.000	0.000	0.000	0.000	0.000
512B	0.743	0.496	0.372	0.000	0.000	0.000	0.000	0.000	0.000
1024B	0.867	0.620	0.248	0.000	0.000	0.000	0.000	0.000	0.000
2048B	------	0.867	0.557	0.000	0.000	0.000	0.000	0.000	0.000
4096B	------	------	0.743	0.000	0.000	0.000	0.000	0.000	0.000
8192B	------	------	------	0.000	0.000	0.000	0.000	0.000	0.000
unified instruction and data stream									
16B	14.965	9.976	3.400	2.237	1.792	1.710	1.668	1.646	1.631
32B	14.108	10.080	4.240	1.396	1.025	0.939	0.895	0.873	0.860
64B	14.151	10.784	5.969	2.059	0.663	0.564	0.515	0.490	0.476
128B	14.547	11.409	7.079	2.558	0.510	0.391	0.331	0.301	0.286
256B	16.155	12.116	7.866	2.932	0.500	0.337	0.256	0.215	0.195
512B	18.250	13.050	8.718	3.418	0.636	0.381	0.254	0.190	0.158
1024B	22.350	15.998	10.833	5.146	2.044	0.555	0.329	0.215	0.159
2048B	------	20.082	13.623	6.645	2.853	0.999	0.547	0.322	0.209
4096B	------	------	17.738	10.026	5.922	3.780	3.095	2.752	2.580
8192B	------	------	------	21.737	12.381	9.477	8.492	7.999	7.753

Table B-12: Cache miss rates for 102.swim, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	10.739	9.185	8.783	8.492	8.106	6.752	4.993	2.347	0.977
32B	7.238	5.481	4.899	4.531	4.202	3.470	2.554	1.202	0.511
64B	10.110	8.058	7.279	6.693	2.350	1.883	1.372	0.640	0.286
128B	23.005	20.680	19.791	19.191	4.552	1.186	0.838	0.383	0.195
256B	30.311	25.402	24.296	23.511	15.463	1.951	0.673	0.296	0.186
512B	34.351	28.759	27.091	26.020	20.827	6.089	1.296	0.287	0.203
1024B	38.765	33.631	29.745	28.399	24.157	9.415	3.971	0.499	0.248
2048B	------	38.735	33.012	30.771	26.572	12.439	6.391	1.512	0.403
4096B	------	------	38.035	35.276	29.068	15.567	8.662	2.685	0.896
8192B	------	------	------	38.438	30.873	18.934	11.316	4.608	2.185
instruction request stream									
16B	9.640	7.180	4.047	1.706	0.850	0.002	0.001	0.001	0.001
32B	5.772	4.460	2.616	1.103	0.550	0.001	0.001	0.001	0.001
64B	3.631	2.897	1.838	0.701	0.325	0.000	0.000	0.000	0.000
128B	2.224	1.831	1.212	0.451	0.250	0.000	0.000	0.000	0.000
256B	1.556	1.331	0.974	0.375	0.200	0.000	0.000	0.000	0.000
512B	1.004	0.883	0.662	0.225	0.100	0.000	0.000	0.000	0.000
1024B	0.788	0.672	0.549	0.275	0.125	0.000	0.000	0.000	0.000
2048B	------	0.610	0.511	0.200	0.100	0.000	0.000	0.000	0.000
4096B	------	------	0.526	0.187	0.125	0.050	0.000	0.000	0.000
8192B	------	------	------	0.259	0.161	0.123	0.000	0.000	0.000
unified instruction and data stream									
16B	13.690	10.184	6.955	4.969	3.865	2.664	1.769	0.853	0.376
32B	9.742	6.993	4.642	3.211	2.307	1.520	0.925	0.447	0.199
64B	9.090	6.624	4.819	3.645	1.519	0.977	0.517	0.248	0.114
128B	12.499	9.969	8.534	7.671	2.251	0.799	0.355	0.177	0.097
256B	15.388	11.706	10.125	9.139	5.855	1.072	0.311	0.150	0.093
512B	17.748	13.136	11.140	10.003	7.665	2.496	0.549	0.162	0.105
1024B	22.906	16.128	12.506	11.245	8.910	3.672	1.482	0.246	0.128
2048B	------	20.726	15.174	12.990	10.189	4.881	2.386	0.624	0.200
4096B	------	------	19.804	16.255	11.572	6.426	3.318	1.097	0.388
8192B	------	------	------	23.451	14.983	8.428	4.806	2.195	1.234

Table B-13: Cache miss rates for 103.su2cor, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	13.484	12.734	11.755	10.882	10.599	10.445	10.392	9.982	9.056
32B	7.462	6.730	6.020	5.508	5.333	5.244	5.212	5.005	4.540
64B	5.203	4.258	3.539	2.880	2.728	2.660	2.636	2.523	2.289
128B	5.073	3.539	2.583	1.652	1.479	1.399	1.371	1.301	1.178
256B	8.228	5.113	3.582	1.625	1.324	1.205	1.159	1.103	1.033
512B	15.168	9.177	6.147	2.687	1.471	1.278	1.191	1.128	1.078
1024B	24.432	15.510	9.939	4.552	2.630	1.899	1.740	1.642	1.511

Table B-14: Cache miss rates for 104.hydro2d, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	24.327	14.881	7.160	4.127	2.695	2.408	2.254	2.023
4096B	------	------	22.034	11.827	7.672	3.740	3.250	2.989	2.355
8192B	------	------	------	18.172	12.422	6.023	5.237	4.844	2.362
instruction request stream									
16B	7.047	5.541	3.649	2.194	0.864	0.482	0.001	0.001	0.001
32B	4.223	3.322	2.257	1.329	0.519	0.273	0.000	0.000	0.000
64B	2.642	2.105	1.493	0.908	0.344	0.164	0.000	0.000	0.000
128B	1.659	1.235	0.891	0.543	0.214	0.097	0.000	0.000	0.000
256B	1.177	0.799	0.609	0.391	0.173	0.062	0.000	0.000	0.000
512B	0.945	0.572	0.442	0.310	0.159	0.035	0.000	0.000	0.000
1024B	0.805	0.417	0.331	0.257	0.149	0.028	0.000	0.000	0.000
2048B	------	0.384	0.299	0.255	0.177	0.021	0.000	0.000	0.000
4096B	------	------	0.269	0.226	0.163	0.034	0.000	0.000	0.000
8192B	------	------	------	0.220	0.159	0.034	0.000	0.000	0.000
unified instruction and data stream									
16B	11.223	9.172	7.203	5.529	4.140	3.682	3.264	3.127	2.834
32B	7.341	5.662	4.338	3.163	2.247	1.906	1.647	1.573	1.424
64B	5.714	4.075	3.031	2.032	1.337	1.026	0.846	0.800	0.721
128B	5.443	3.446	2.383	1.408	0.883	0.588	0.462	0.421	0.376
256B	7.006	4.296	2.832	1.526	1.011	0.524	0.406	0.364	0.333
512B	10.533	6.281	3.971	2.081	1.213	0.632	0.442	0.385	0.353
1024B	15.732	9.834	6.101	3.190	1.980	0.963	0.655	0.566	0.499
2048B	------	15.453	9.426	5.117	3.167	1.369	0.940	0.795	0.678
4096B	------	------	15.242	8.499	5.319	2.081	1.426	1.167	0.888
8192B	------	------	------	14.129	8.975	3.562	2.389	1.935	1.011

Table B-14: Cache miss rates for 104.hydro2d, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	8.083	4.908	4.381	3.991	3.822	3.053	2.963	2.233	2.153
32B	5.178	3.052	2.510	2.175	2.020	1.600	1.536	1.138	1.093
64B	5.934	2.620	1.865	1.457	1.235	0.966	0.901	0.596	0.566
128B	13.171	2.920	1.801	1.228	0.908	0.681	0.600	0.333	0.307
256B	18.171	4.007	2.243	1.357	0.873	0.610	0.493	0.216	0.184
512B	25.055	7.388	3.964	2.048	1.163	0.718	0.502	0.199	0.144
1024B	30.573	12.685	6.377	2.746	1.440	0.832	0.533	0.196	0.121
2048B	------	19.776	12.142	7.764	2.277	1.234	0.720	0.282	0.155
4096B	------	------	19.499	13.398	4.990	2.125	1.126	0.486	0.254
8192B	------	------	------	23.060	13.088	3.872	1.970	0.905	0.471
instruction request stream									
16B	0.020	0.018	0.015	0.009	0.004	0.002	0.000	0.000	0.000
32B	0.011	0.010	0.009	0.006	0.003	0.001	0.000	0.000	0.000
64B	0.007	0.006	0.005	0.003	0.002	0.001	0.000	0.000	0.000
128B	0.004	0.004	0.003	0.002	0.001	0.001	0.000	0.000	0.000
256B	0.003	0.002	0.002	0.001	0.001	0.000	0.000	0.000	0.000
512B	0.002	0.002	0.001	0.001	0.001	0.000	0.000	0.000	0.000
1024B	0.002	0.002	0.001	0.001	0.000	0.000	0.000	0.000	0.000

Table B-15: Cache miss rates for 107.mgrid, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	0.002	0.001	0.001	0.000	0.000	0.000	0.000	0.000
4096B	------	------	0.001	0.000	0.000	0.000	0.000	0.000	0.000
8192B	------	------	------	0.000	0.000	0.000	0.000	0.000	0.000
unified instruction and data stream									
16B	12.481	8.974	2.906	2.174	1.812	1.367	1.254	0.930	0.880
32B	8.707	6.215	1.927	1.331	1.035	0.757	0.670	0.485	0.453
64B	7.588	4.893	1.602	1.011	0.706	0.494	0.413	0.266	0.241
128B	9.387	4.229	1.667	0.964	0.598	0.391	0.298	0.164	0.139
256B	13.314	6.003	2.109	1.148	0.650	0.396	0.272	0.125	0.094
512B	16.806	7.760	3.196	1.629	0.867	0.490	0.300	0.131	0.085
1024B	23.072	13.340	4.958	2.340	1.188	0.641	0.364	0.155	0.088
2048B	------	18.971	8.506	4.934	1.884	1.002	0.557	0.268	0.160
4096B	------	------	14.608	8.795	3.881	1.832	0.955	0.466	0.259
8192B	------	------	------	15.667	8.742	3.574	1.863	0.982	0.585

Table B-15: Cache miss rates for 107.mgrid, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	7.216	5.826	5.325	5.061	4.902	4.801	4.723	4.595	4.275
32B	5.234	3.512	2.969	2.693	2.535	2.439	2.389	2.320	2.157
64B	5.092	2.630	1.913	1.573	1.380	1.266	1.226	1.184	1.098
128B	6.456	2.852	1.721	1.171	0.873	0.702	0.655	0.622	0.572
256B	9.837	4.124	2.234	1.268	0.753	0.462	0.390	0.352	0.316
512B	15.537	7.355	3.831	2.104	1.145	0.610	0.301	0.240	0.200
1024B	23.102	11.671	6.226	3.449	1.850	0.940	0.363	0.213	0.154
2048B	------	20.776	10.474	5.712	2.996	1.356	0.574	0.307	0.173
4096B	------	------	17.402	9.713	5.127	2.178	0.942	0.559	0.267
8192B	------	------	------	18.624	9.896	4.657	1.804	1.018	0.517
instruction request stream									
16B	12.205	4.805	0.003	0.002	0.000	0.000	0.000	0.000	0.000
32B	6.114	2.422	0.002	0.001	0.000	0.000	0.000	0.000	0.000
64B	3.078	1.248	0.001	0.001	0.000	0.000	0.000	0.000	0.000
128B	1.545	0.634	0.000	0.000	0.000	0.000	0.000	0.000	0.000
256B	0.788	0.346	0.000	0.000	0.000	0.000	0.000	0.000	0.000
512B	0.406	0.193	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1024B	0.223	0.134	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2048B	------	0.096	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4096B	------	------	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8192B	------	------	------	0.000	0.000	0.000	0.000	0.000	0.000
unified instruction and data stream									
16B	13.841	7.226	3.108	2.320	1.832	1.671	1.587	1.516	1.398
32B	8.522	4.317	1.942	1.351	1.010	0.872	0.814	0.771	0.707
64B	6.295	2.922	1.371	0.886	0.597	0.472	0.427	0.398	0.362
128B	5.656	2.554	1.267	0.718	0.420	0.281	0.238	0.214	0.191
256B	6.956	3.047	1.546	0.791	0.403	0.211	0.156	0.128	0.108
512B	9.310	4.548	2.338	1.190	0.574	0.276	0.135	0.096	0.072
1024B	15.274	6.927	3.603	1.947	0.914	0.418	0.173	0.096	0.061

Table B-16: Cache miss rates for 110.applu, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	11.767	6.252	3.393	1.487	0.636	0.282	0.145	0.075
4096B	------	------	13.161	6.023	2.629	1.105	0.515	0.294	0.122
8192B	------	------	------	11.168	5.265	2.590	1.335	0.906	0.239

Table B-16: Cache miss rates for 110.applu, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	4.683	3.920	3.550	2.688	2.268	2.174	1.864	1.505	1.489
32B	4.121	3.433	3.178	2.313	1.663	1.572	1.204	0.767	0.755
64B	4.065	3.461	3.255	2.158	1.364	1.271	0.871	0.394	0.386
128B	4.644	3.868	3.640	2.119	1.235	1.131	0.710	0.211	0.201
256B	5.738	4.604	4.262	2.149	1.201	1.076	0.637	0.123	0.111
512B	7.478	5.718	5.246	2.256	1.242	1.078	0.617	0.087	0.070
1024B	8.775	6.474	5.678	2.390	1.323	1.109	0.620	0.075	0.051
2048B	------	8.759	7.309	2.960	2.506	1.751	1.098	0.541	0.504
4096B	------	------	8.740	3.791	3.024	2.387	1.389	0.791	0.730
8192B	------	------	------	7.260	6.202	2.819	1.770	1.002	0.887
instruction request stream									
16B	2.388	1.404	0.923	0.128	0.037	0.001	0.000	0.000	0.000
32B	1.344	0.781	0.532	0.096	0.025	0.001	0.000	0.000	0.000
64B	0.836	0.535	0.382	0.100	0.015	0.000	0.000	0.000	0.000
128B	0.522	0.350	0.276	0.079	0.011	0.000	0.000	0.000	0.000
256B	0.433	0.305	0.259	0.042	0.007	0.000	0.000	0.000	0.000
512B	0.402	0.209	0.189	0.041	0.007	0.000	0.000	0.000	0.000
1024B	0.408	0.207	0.170	0.041	0.007	0.000	0.000	0.000	0.000
2048B	------	0.202	0.166	0.038	0.005	0.000	0.000	0.000	0.000
4096B	------	------	0.325	0.043	0.009	0.000	0.000	0.000	0.000
8192B	------	------	------	0.045	0.011	0.000	0.000	0.000	0.000
unified instruction and data stream									
16B	7.405	4.438	2.023	1.124	0.868	0.684	0.557	0.441	0.430
32B	5.588	3.170	1.526	0.918	0.636	0.497	0.361	0.228	0.219
64B	4.915	2.714	1.375	0.827	0.505	0.400	0.263	0.120	0.114
128B	4.773	2.602	1.400	0.802	0.471	0.358	0.217	0.068	0.061
256B	5.477	3.017	1.590	0.807	0.477	0.347	0.198	0.044	0.036
512B	7.615	4.026	2.080	0.885	0.516	0.363	0.197	0.036	0.026
1024B	9.953	5.664	2.669	1.301	0.899	0.383	0.204	0.036	0.022
2048B	------	8.488	3.530	1.617	1.324	0.590	0.354	0.177	0.156
4096B	------	------	5.013	2.489	1.676	0.840	0.471	0.266	0.231
8192B	------	------	------	13.694	11.889	1.096	0.660	0.380	0.315

Table B-17: Cache miss rates for 125.turb3d, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	8.533	7.232	6.902	5.217	3.927	2.491	1.278	0.062	0.001
32B	7.269	6.218	5.985	4.694	3.262	1.939	0.965	0.055	0.001
64B	6.995	5.911	5.646	4.450	2.943	1.673	0.816	0.056	0.001
128B	7.729	6.256	5.954	4.369	2.817	1.566	0.761	0.073	0.000
256B	8.978	6.714	6.299	4.466	2.858	1.566	0.767	0.090	0.000
512B	11.845	7.833	7.158	5.060	3.257	1.606	0.789	0.098	0.000
1024B	17.041	10.488	9.395	5.750	3.748	1.712	0.846	0.113	0.000
2048B	------	13.644	12.018	6.220	3.827	1.699	0.793	0.138	0.002
4096B	------	------	14.810	7.438	4.801	1.994	0.984	0.326	0.002
8192B	------	------	------	10.094	6.457	2.329	1.196	0.460	0.024
instruction request stream									
16B	4.893	2.957	1.728	0.434	0.169	0.035	0.003	0.000	0.000
32B	2.785	1.682	0.952	0.236	0.090	0.019	0.002	0.000	0.000
64B	1.625	0.965	0.526	0.130	0.051	0.010	0.001	0.000	0.000
128B	0.993	0.570	0.289	0.077	0.033	0.007	0.001	0.000	0.000
256B	0.755	0.405	0.207	0.047	0.022	0.005	0.000	0.000	0.000
512B	0.528	0.302	0.140	0.035	0.017	0.003	0.000	0.000	0.000
1024B	0.500	0.302	0.169	0.028	0.013	0.003	0.000	0.000	0.000
2048B	------	0.273	0.174	0.028	0.015	0.004	0.000	0.000	0.000
4096B	------	------	0.191	0.031	0.017	0.004	0.000	0.000	0.000
8192B	------	------	------	0.043	0.024	0.004	0.000	0.000	0.000
unified instruction and data stream									
16B	11.311	7.846	5.458	3.431	1.824	1.206	0.593	0.133	0.099
32B	8.230	5.686	4.072	2.723	1.430	0.902	0.428	0.081	0.056
64B	6.767	4.575	3.388	2.295	1.236	0.747	0.353	0.057	0.034
128B	6.827	4.386	3.340	2.163	1.164	0.691	0.323	0.052	0.025
256B	8.255	5.097	3.849	2.165	1.180	0.689	0.325	0.055	0.022
512B	11.281	6.398	4.722	2.566	1.350	0.708	0.333	0.058	0.024
1024B	17.684	10.031	6.897	3.531	1.586	0.803	0.407	0.112	0.032
2048B	------	15.133	10.336	4.018	1.742	0.839	0.397	0.125	0.038
4096B	------	------	14.652	6.493	2.596	1.183	0.493	0.208	0.057
8192B	------	------	------	10.815	3.970	1.829	0.769	0.408	0.108

Table B-18: Cache miss rates for 141.apsi, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	9.008	6.488	5.295	4.391	4.316	4.182	0.000	------	------
32B	6.943	5.304	4.524	3.737	3.670	3.555	0.000	------	------
64B	5.638	4.334	3.726	2.986	2.921	2.823	0.000	------	------
128B	5.370	4.151	3.477	2.595	2.533	2.427	0.000	------	------
256B	6.615	4.966	4.225	3.228	3.161	2.998	0.000	------	------
512B	9.609	6.752	5.751	4.421	4.336	4.156	0.000	------	------
1024B	13.655	9.275	7.526	5.916	5.819	5.638	0.000	------	------

Table B-19: Cache miss rates for 145.fpppp, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	16.834	14.678	11.478	11.361	11.157	0.000	------	------
4096B	------	------	22.152	16.211	16.112	15.431	0.000	------	------
8192B	------	------	------	16.722	16.664	15.859	0.000	------	------
instruction request stream									
16B	47.068	45.903	34.737	27.516	16.627	0.159	0.001	------	------
32B	23.760	23.139	17.572	13.858	8.381	0.101	0.000	------	------
64B	12.076	11.733	8.948	6.997	4.230	0.061	0.000	------	------
128B	6.208	6.011	4.618	3.553	2.147	0.037	0.000	------	------
256B	3.247	3.124	2.453	1.824	1.111	0.022	0.000	------	------
512B	1.747	1.668	1.340	0.963	0.592	0.019	0.000	------	------
1024B	0.992	0.927	0.759	0.530	0.320	0.017	0.000	------	------
2048B	------	0.554	0.466	0.318	0.178	0.012	0.000	------	------
4096B	------	------	0.329	0.217	0.123	0.022	0.000	------	------
8192B	------	------	------	0.156	0.090	0.022	0.000	------	------
unified instruction and data stream									
16B	33.305	30.919	24.997	18.429	12.602	3.664	0.165	------	------
32B	18.963	17.142	13.874	10.260	7.216	2.655	0.140	------	------
64B	11.821	10.179	8.178	5.929	4.284	1.922	0.117	------	------
128B	8.752	6.915	5.449	3.752	2.808	1.539	0.103	------	------
256B	9.025	6.505	4.832	3.184	2.539	1.752	0.098	------	------
512B	12.259	8.074	5.615	3.475	2.921	2.332	0.091	------	------
1024B	19.111	11.465	7.185	4.220	3.612	3.094	0.083	------	------
2048B	------	20.217	14.029	7.533	6.560	5.961	0.086	------	------
4096B	------	------	21.394	11.199	9.548	8.225	0.078	------	------
8192B	------	------	------	17.482	11.318	9.102	0.094	------	------

Table B-19: Cache miss rates for 145.fpppp, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
data reference stream									
16B	27.813	24.393	17.336	12.137	5.585	3.784	3.137	2.631	2.384
32B	25.932	22.256	14.367	9.032	3.108	1.951	1.593	1.332	1.203
64B	24.882	21.038	12.873	7.568	1.888	1.057	0.824	0.680	0.610
128B	24.516	20.449	12.503	6.950	1.368	0.665	0.455	0.361	0.318
256B	25.500	20.966	12.951	7.326	1.302	0.604	0.309	0.220	0.180
512B	28.648	23.647	14.017	7.993	1.696	0.902	0.382	0.259	0.206
1024B	32.300	26.494	16.864	9.650	2.663	1.679	0.616	0.406	0.325
2048B	------	30.507	20.901	12.817	4.411	3.047	1.317	0.813	0.667
4096B	------	------	28.614	18.424	9.197	6.787	4.604	3.717	3.417
8192B	------	------	------	22.907	11.998	8.579	5.055	3.913	3.460
instruction request stream									
16B	4.198	2.943	1.281	1.258	0.005	0.002	0.000	0.000	0.000
32B	2.514	1.803	0.764	0.751	0.004	0.001	0.000	0.000	0.000
64B	1.568	1.145	0.472	0.464	0.003	0.001	0.000	0.000	0.000
128B	0.962	0.702	0.271	0.266	0.003	0.001	0.000	0.000	0.000
256B	0.620	0.448	0.160	0.157	0.002	0.000	0.000	0.000	0.000
512B	0.517	0.360	0.117	0.113	0.002	0.000	0.000	0.000	0.000
1024B	0.483	0.372	0.075	0.069	0.002	0.000	0.000	0.000	0.000

Table B-20: Cache miss rates for 146.wave5, test input set, direct-mapped caches

blk. size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
2048B	------	0.473	0.053	0.048	0.002	0.000	0.000	0.000	0.000
4096B	------	------	0.140	0.135	0.002	0.000	0.000	0.000	0.000
8192B	------	------	------	0.202	0.002	0.000	0.000	0.000	0.000
unified instruction and data stream									
16B	15.149	11.907	7.975	5.769	2.519	1.565	1.210	1.002	0.901
32B	13.104	10.300	6.491	4.261	1.553	0.842	0.636	0.525	0.471
64B	12.082	9.452	5.769	3.540	1.088	0.490	0.351	0.286	0.255
128B	11.767	9.101	5.565	3.245	0.907	0.332	0.202	0.157	0.136
256B	12.545	9.479	5.865	3.420	0.952	0.328	0.166	0.121	0.101
512B	14.713	11.007	6.683	3.897	1.274	0.456	0.199	0.139	0.112
1024B	19.544	13.653	8.510	4.975	1.990	0.821	0.313	0.214	0.173
2048B	------	19.258	12.449	7.139	3.291	1.395	0.594	0.377	0.306
4096B	------	------	18.736	10.792	5.992	3.158	1.860	1.475	1.331
8192B	------	------	------	15.624	9.143	4.281	2.240	1.659	1.419

Table B-20: Cache miss rates for 146.wave5, test input set, direct-mapped caches

B.3 Validating cache simulation

		direct-mapped				4-way set associative			
benchmark	**indexing**	**16KB**	**64KB**	**256KB**	**1MB**	**16KB**	**64KB**	**256KB**	**1MB**
099.go	cheetah/VIVT	9.971	3.035	1.481	0.000	5.955	0.532	0.008	0.000
	sim-cache/VIVT	9.970	3.030	1.480	0.000	5.950	0.530	0.010	0.000
	sim-cache/PIPT	10.980	4.550	0.970	0.000	5.950	0.630	0.010	0.000
124.m88ksim	cheetah/VIVT	1.111	0.528	0.334	0.326	0.376	0.337	0.325	0.324
	sim-cache/VIVT	1.120	0.540	0.350	0.340	0.380	0.350	0.340	0.340
	sim-cache/PIPT	1.170	0.410	0.350	0.340	0.380	0.350	0.340	0.340
126.gcc	cheetah/VIVT	3.428	1.126	0.465	0.109	1.558	0.532	0.252	0.052
	sim-cache/VIVT	3.440	1.170	0.550	0.250	1.570	0.580	0.340	0.200
	sim-cache/PIPT	3.440	1.380	0.580	0.260	1.570	0.620	0.340	0.200
129.compress	cheetah/VIVT	4.912	2.643	0.920	------	3.608	1.988	0.989	0.068
	sim-cache/VIVT	4.910	2.640	0.940	0.150	3.610	1.990	1.000	0.110
	sim-cache/PIPT	4.310	2.500	0.770	0.110	3.610	1.980	0.980	0.110
130.li	cheetah/VIVT	2.178	0.810	0.004	------	1.378	0.628	0.004	0.004
	sim-cache/VIVT	2.260	1.620	1.550	1.550	1.610	1.550	1.550	1.550
	sim-cache/PIPT	2.130	0.820	0.020	0.020	1.380	0.640	0.020	0.020
132.ijpeg	cheetah/VIVT	1.837	0.795	0.515	0.449	0.609	0.229	0.130	0.032
	sim-cache/VIVT	1.840	0.800	0.610	0.580	0.610	0.240	0.210	0.190
	sim-cache/PIPT	2.120	0.570	0.270	0.190	0.610	0.240	0.200	0.190
134.perl	cheetah/VIVT	2.150	0.801	0.257	0.165	0.569	0.423	0.214	0.155
	sim-cache/VIVT	2.170	0.840	0.310	0.240	0.590	0.460	0.270	0.230
	sim-cache/PIPT	2.280	1.540	0.330	0.250	0.590	0.470	0.270	0.230
147.vortex	cheetah/VIVT	4.263	1.738	0.364	0.143	1.402	0.440	0.113	0.051
	sim-cache/VIVT	4.300	1.770	0.400	0.190	1.430	0.470	0.150	0.100
	sim-cache/PIPT	3.570	1.120	0.440	0.160	1.430	0.460	0.160	0.100

Table B-21: Cache performance varying simulator and indexing for SPECINT95

		direct-mapped				4-way set associative			
benchmark	**indexing**	**16KB**	**64KB**	**256KB**	**1MB**	**16KB**	**64KB**	**256KB**	**1MB**
101.tomcatv	cheetah/VIVT	12.469	3.513	3.436	3.381	10.985	1.078	1.071	1.047
	sim-cache/VIVT	12.510	3.530	3.450	3.400	11.020	1.090	1.080	1.060
	sim-cache/PIPT	13.020	2.700	1.470	1.150	11.020	1.090	1.080	1.060
102.swim	cheetah/VIVT	28.150	2.310	2.258	2.244	33.457	2.232	2.158	2.153
	sim-cache/VIVT	28.150	2.310	2.260	2.250	33.460	2.230	2.160	2.150
	sim-cache/PIPT	27.870	8.460	3.750	2.580	33.460	2.240	2.160	2.150
103.su2cor	cheetah/VIVT	7.495	2.295	1.774	0.722	2.202	2.140	1.787	0.685
	sim-cache/VIVT	7.510	2.300	1.780	0.730	2.210	2.150	1.790	0.690
	sim-cache/PIPT	4.490	2.350	1.720	0.740	2.210	2.150	1.870	0.710
104.hydro2d	cheetah/VIVT	3.855	3.029	2.925	2.544	3.240	2.866	2.864	2.662
	sim-cache/VIVT	3.860	3.040	2.930	2.550	3.250	2.870	2.870	2.670
	sim-cache/PIPT	3.900	3.060	2.920	2.650	3.250	2.870	2.870	2.670
107.mgrid	cheetah/VIVT	1.903	1.282	0.954	0.638	1.051	0.997	0.671	0.621
	sim-cache/VIVT	1.900	1.280	0.960	0.650	1.050	1.000	0.670	0.630

Table B-22: Cache performance varying simulator and indexing for SPECFP95

		direct-mapped				4-way set associative			
benchmark	**indexing**	**16KB**	**64KB**	**256KB**	**1MB**	**16KB**	**64KB**	**256KB**	**1MB**
	sim-cache/PIPT	1.910	1.190	0.720	0.650	1.050	1.000	0.670	0.630
110.applu	cheetah/VIVT	1.839	1.319	1.179	1.094	1.191	1.138	1.132	1.083
	sim-cache/VIVT	1.840	1.320	1.180	1.100	1.190	1.140	1.130	1.090
	sim-cache/PIPT	1.980	1.330	1.180	1.110	1.190	1.140	1.130	1.080
125.turb3d	cheetah/VIVT	3.202	1.426	0.909	0.398	1.792	1.095	0.982	0.372
	sim-cache/VIVT	3.200	1.430	0.910	0.400	1.790	1.090	0.980	0.370
	sim-cache/PIPT	2.740	1.590	1.020	0.900	1.790	0.730	0.560	0.480
141.apsi	cheetah/VIVT	5.078	4.699	2.984	0.787	2.411	2.069	1.618	0.213
	sim-cache/VIVT	5.080	4.700	2.980	0.790	2.410	2.070	1.620	0.210
	sim-cache/PIPT	3.990	2.470	1.550	0.370	2.410	2.050	1.590	0.170
145.fpppp	cheetah/VIVT	3.798	2.988	0.001	------	0.251	0.013	0.001	0.001
	sim-cache/VIVT	3.800	2.990	0.000	0.000	0.250	0.010	0.000	0.000
	sim-cache/PIPT	1.490	0.480	0.000	0.000	0.250	0.010	0.000	0.000
146.wave5	cheetah/VIVT	13.825	2.018	0.933	0.680	16.539	1.385	0.437	0.231
	sim-cache/VIVT	13.830	2.020	0.930	0.680	16.540	1.390	0.440	0.230
	sim-cache/PIPT	15.190	2.580	0.630	0.320	16.540	1.950	0.440	0.240

Table B-22: Cache performance varying simulator and indexing for SPECFP95