Chapter 1

Introduction

The memory system is a critical component of any high-performance computer system. Memory speed is often a major component of the perceived execution speed of the computer since the processor can only execute as fast as the memory system provides data. Computer users would prefer a memory that is both infinitely large and infinitely fast, or at least one that is large enough and fast enough. The challenge is to design a cost-effective memory system that meets these goals.

Rapid technology change always alters the memory system design problem, making previous designs an insufficient solution, and new research essential. Memory densities are dramatically improving: main memories of hundreds of megabytes will be common in the future. Processors are getting faster: processors will execute hundreds of millions of instructions in a single second. With more processing and memory capabilities, users solve different and larger problems than previously envisioned. These technology and usage changes reshape the problems presented to the memory system designer. As memory systems evolve, design decisions must be constantly reevaluated.

This dissertation analyzes a memory system that is motivated by increasing processing speeds and expanding main memories. While main memories are getting much larger, they are not getting much faster. This creates a speed gap between the processor and main memory. The processor can perform hundreds of operations in the time it takes to service a single main memory request, so memory accesses must be rare. With this large speed gap, it becomes particularly difficult to meet the expectations of the user: a fast and large memory. While a given memory may be large enough, it is probably not fast enough.

The problem is that larger memories are inherently slower than smaller memories because of physical limitations. Given this problem, it may at first seem impossible to build a memory system that has both a large capacity and a high speed. Fortunately, there is an elegant design solution that overcomes these physical limitations. A smaller (faster) and a larger (slower) memory are configured

hierarchically, yet they appear as a single memory because the faster memory transparently caches the frequently-referenced contents of the slower memory. The average speed of the hierarchy is nearly the speed of the fastest memory because it services most references (locality of reference [DENN68]). The largest memory decides the capacity of the hierarchy because it services all the other references that the smaller one(s) cannot. At no burden to the programmer, hierarchical memory systems bridge the large speed gap between processing and memory by simultaneously using the speed and capacity advantages of small and large memories.

Figure 1.1 shows the evolution (from left to right) of hierarchical memory systems to fill the processor-memory speed gap. Technology changes push designs to the far right of the figure: to the memory system examined in this study. *Virtual memory* (on the far left) is the earliest transparent hierarchical structuring of mechanical (disk or drum) and electronic main memories [FOTH61, KILB62, KIEL82]. Main memory could successfully bridge the speed gap between processors and mechanical storage when it could service references at the speed of the processor. Faster processors create another speed gap since large main memories cannot equal processor speeds. Another hierarchy level bridges this gap: *cache* memory. Caches are essential for the processor-main-memory performance gap just like virtual memory is essential for the main memory and disk gap. Caches are faster (but smaller) than main memories, so they can satisfy processor references more quickly. Caches greatly improve memory system performance, at no burden to the programmer.

With even faster processors, a single cache becomes an insufficient buffer between the processor and main memory. A two-level cache configuration is necessary. Again, an addition of another level to the hierarchy allows the capacity advantages of the large cache to be combined with the speed advantages of the smaller cache. The large cache reduces the frequency of main memory accesses, which is essential because they take so long, but a large cache is too slow to directly service the processor. This leads to the configuration on the far right of Figure 1.1, one with smaller primary and larger secondary CPU caches. This two-level cache configuration gracefully tolerates a processor-memory speed gap of even a hundred or more. The primary caches service most processor references, and the secondary cache services most of the references that escape the primary caches. Split primary caches provide high bandwidth to the processor; separate instruction and data caches allow instruction and data references to be serviced simultaneously.

The technology changes that drive hierarchical memory systems to two cache levels are evident in Figure 1.1. Microprocessor speeds have improved by a factor of one hundred over the last decade while main memory speeds have improved by only 50% in the same period. CPU caches fill this speed gap by pushing the processor up in the hierarchy. Memory sizes have grown by several orders of magnitude. Caches also grow with each evolutionary generation. These large caches can satisfy most of the references from even the largest workloads.

1.1. Dissertation Overview and Contributions

This study focuses on the evaluation and design of the evolutionary two-level cache configuration depicted on the far right in Figure 1.1. In particular, this dissertation concentrates on large secondary caches. Most previous cache analyses focus on smaller primary caches because only recent technology advances and increasing workload sizes necessitate the larger caches as a part of a two-level cache configuration. Evolutionary trends suggest that these caches will be multiple megabytes, an order of magnitude larger than the previous generation of caches. The scope of this study is restricted to

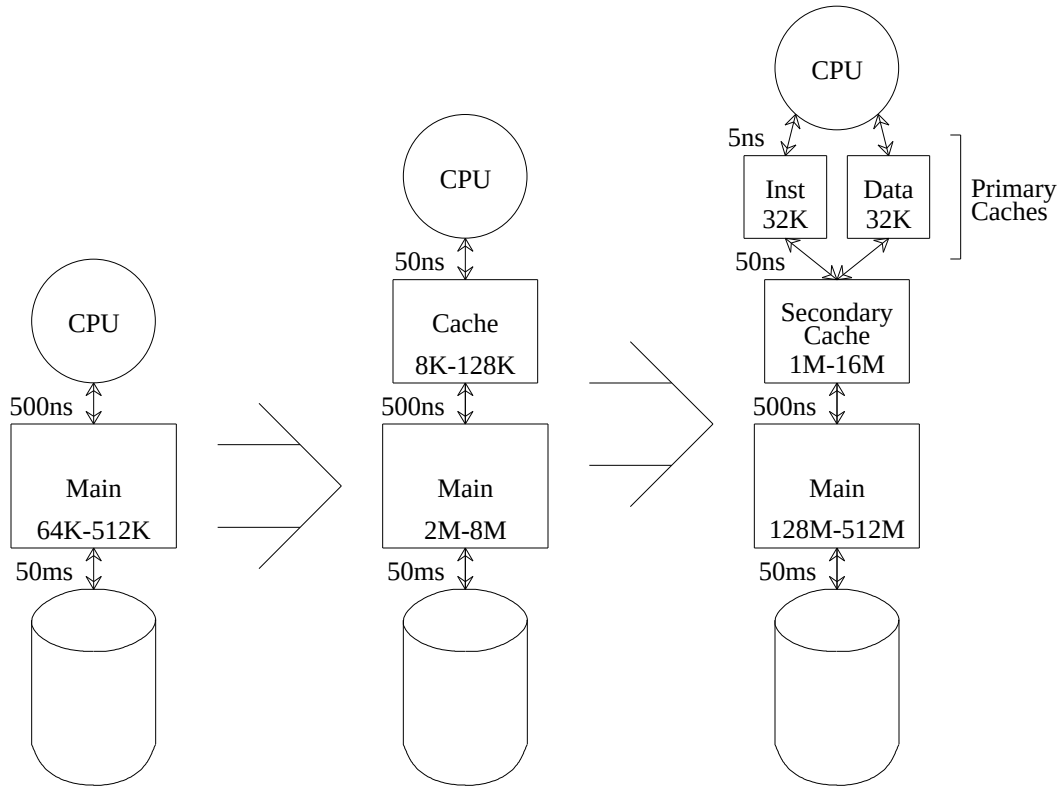


Figure 1.1. Hierarchical Memory System Evolution.

This figure shows the evolution of hierarchical memory systems (from left to right) to fill the gap between processor and memory speeds. At the top of the hierarchy is the processor (CPU), and at the bottom is non-volatile (mechanical) storage. In the middle, there are several levels of memory, both cache and main, with sizes given. The latency required for a transfer between levels is also given. The latencies and sizes depicted are only examples. This dissertation studies secondary caches in a configuration like the one at the far right.

uniprocessors because they continue to push the limits of low-cost computational performance, though multi-megabyte secondary caches are equally or more useful for multiprocessors.

While primary cache and main memory design considerations are much the same as in the previous generation, the new secondary caches are fundamentally different. Secondary caches are different from primary ones for two reasons. First, secondary caches will be as large as main memories once were, much larger than previous (or future) primary caches. Second, primary caches directly service processor references, while secondary caches service only the references not satisfied by the primary caches. Most previous research has focused on primary cache design. This dissertation studies the secondary cache, which sees memory references with very different characteristics.

Although they are of similar size, multi-megabyte secondary caches are fundamentally different from previous main memories. One difference is the time to access the next hierarchy level: disk accesses take many orders of magnitude longer than main memory accesses. This latency difference makes secondary cache design very different. Secondary caches can use the main memory much more frequently than previous main memories could access the slow disk.

The differences in multi-megabyte secondary cache design motivate the research in this dissertation. The first component of this research involved gathering tools appropriate for analyzing multi-

megabyte secondary caches. As memory systems evolve, the tools used to evaluate them also must evolve. This dissertation describes the implementation and use of new cache performance analysis tools. By evaluating multi-megabyte secondary caches in the proper framework, new insights into their behavior and design are evident.

Chapter 2 defines the basic concepts related to virtual memory and CPU caches, and gives the default parameters that are assumed throughout the rest of this dissertation. It also surveys some important previous cache performance analysis studies, and shows trace-driven simulation to be an effective performance analysis tool for cache memories. It introduces the *MPI* and *SCPI* cache performance metrics, and describes the statistical techniques used throughout this dissertation. It then describes a simulation environment that allows trace-driven simulation results to be gathered quickly. Chapter 2 establishes the basis for the rest of the research included in this dissertation.

Chapter 3 then describes the collection of superior traces for the analysis of multi-megabyte caches. These traces predict the workloads that will execute on future high performance processors that have main memories of hundreds of megabytes. The traces are one hundred times longer than previous traces, and are taken from workloads that are ten times larger than previously traced. Chapter 3 further shows the advantages of very long traces for multi-megabyte cache analysis.

The problem with long traces is the enormous computing resources needed to simulate with them. Chapter 4 discusses trace-sampling techniques that use only a portion of the long trace references, and consequently reduce the required simulation time, to get accurate cache performance results. Sampling can reduce simulation time by more than an order of magnitude while introducing only small errors.

Chapter 5 gives more detailed motivation for the two-level hierarchical cache configuration shown in Figure 1.1. It shows that multi-megabyte caches are needed as the processor-memory speed gap reaches a factor of one hundred. It then discusses some key design considerations of multi-megabyte secondary caches: block size, associativity, and multi-level inclusion.

Chapter 6 examines the interaction of virtual memory and set-associative CPU caches. Because of their interaction, the virtual to real page mapping affects the placement of data in the cache. Chapter 6 introduces the problem caused by this interaction: page conflicts. It also introduces a simple model to measure the quality of naive page placement in the cache. This model suggests that 30% of pages may be poorly placed when they are naively mapped to page frames. Chapter 6 introduces and examines practical page mapping techniques that improve the placement of data in the cache and, consequently, improve cache and memory system performance. Trace-driven simulation shows that careful page mapping eliminates 10%-20% of direct-mapped real-indexed cache misses. Thus, careful page mapping can cause a cache to act much larger, at no hardware cost.

Finally, Chapter 7 discusses the conclusions of this research, and areas for future research. This dissertation presents the analysis required to understand the performance and design issues involved with multi-megabyte secondary caches.

Chapter 2

Background and Cache Performance Analysis

2.1. Introduction

This chapter gives the memory system background needed for the analysis of the subsequent chapters. In Section 2.1 and Section 2.2, this chapter discusses definitions and default parameters of the virtual memory and cache configurations examined throughout this dissertation. Section 2.3 surveys the most important previous cache performance studies, and motivates the use of trace-driven simulation. Section 2.4 describes the cache performance metrics used throughout this dissertation. Section 2.5 summarizes the statistical techniques used in Chapters 4 and 6. Section 2.6 then describes a simulation environment that takes advantage of levels of homogeneity and parallelism. This chapter discusses the tools used in the subsequent research in Chapters 3, 4, 5, and 6.

2.2. Virtual Memory Concepts

This section summarizes key concepts of virtual memory that are used throughout this dissertation. Definition 2.1 gives key definitions and default parameters related to virtual memory.

Definition 2.1. Definitions for Virtual Memory.

- **Page** - A fixed-size, aligned, and contiguous portion of virtual memory that is managed as a unit. The default page size is 16 kilobytes.
- **Page Frame** - Physical memory that can hold a page. Typically, this refers to main memory. The number of page frames times the page size is the size of the main memory. The default main memory size is 128 megabytes.

- **Page Fault** - An exception that occurs when a referenced page is not held in the main memory. Usually, the operating system stops the user-level program that faulted, loads the page from disk into main memory, and restarts the program.
- **Page Mapping (Placement)** - The relationship between virtual memory pages and page frames. The page mapping specifies where a given page is stored.
- **Page Mapping Function (Policy)** - The algorithm that determines the page mapping.
- **Page Replacement** - The removal of a page from main memory to make room for a new page. This may require writing the replaced page back to disk.
- **Page Replacement Policy** - The algorithm that chooses the page(s) that will be removed from main memory. Replacement policies tend to choose pages that were not recently referenced. The default is to replace the least-recently-used page.

The goal of virtual memory is to provide the user with a unified view of a large and fast memory, although the physical memory is really structured hierarchically. Two separate address spaces provide this view. The user references *virtual addresses* while *real addresses* index into the physically available memory. The virtual memory system manages the memory in pages. At any given time, a virtual memory page may either reside in a (physical) main memory page frame or it may be on disk. That is, each page maps to page frames in the main memory or disk memory. When there are no page faults, the system translates each virtual (or page) address to a real (or page frame) address and the main memory is directly accessed. This address translation is depicted in Figure 2.1. Because pages (frames) are properly aligned, the translation does not modify the offset of an address within a page (i.e. the bottom address bits are unchanged). Translation does modify the upper address bits. It determines the page frame corresponding to a virtual page by consulting the page mapping, often a *page table*. Occasionally, the page table shows that a virtual page is not contained in the main memory. Then, a page fault awakens the system software, and a page rearrangement resolves the fault. Pages may need to be written-back to the disk because of the page fault.

Virtual address translation is typically a fully associative mapping of pages to page frames. This means that any page can be stored in any page frame in main memory. This mapping flexibility allows the system software to choose many different maps of pages to page frames. It also allows considerable flexibility in deciding which pages will reside on the disk and which will be placed in main memory. Typically, recently used pages are held in the main memory, and unreferenced pages reside on the disk. With this policy, the page fault frequency will be low if there is enough locality of reference. The page fault frequency is a key metric of the performance of a virtual memory system. The more often faults occur, the more often the disk must be accessed. Since a disk access is so slow (millions of instructions can be executed during a disk access), page faults must be rare for good performance.

2.3. Set-Associative CPU Cache Concepts

This section summarizes key concepts of caches that are used throughout this dissertation. Definition 2.2 gives key definitions and default parameters related to caches.

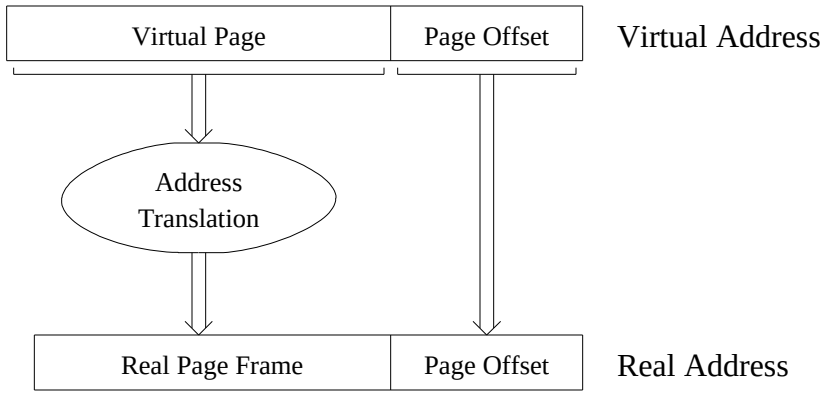


Figure 2.1. Virtual to Real Address Translation.

This figure shows the translation from virtual addresses to real addresses. The virtual page number translates into a real page frame number.

Definition 2.2. Definitions for Caches.

- **Block** - A fixed-size, aligned, and contiguous portion of memory that is managed as a unit. A block is similar in function, but is typically much smaller than, a virtual memory page. In this dissertation, the primary cache block size is 32 bytes and the secondary cache default block size is 128 bytes.
- **Block Frame** - A cache memory location that can store a block. The size of the cache is the number of block frames times the size of a block. Each primary cache contains 32-kilobytes worth of block frames. The secondary cache size varies from 128-kilobytes to 16-megabytes.
- **Set** - The block frame(s) that can store a block. Each block indexes to a particular set, and only the block frames within that set can hold the block.
- **Cache Miss** - A hardware-handled exception that occurs when the processor references a block that is not in the cache. The cache replaces another block from the same set, and loads the referenced block into the cache to service the reference.
- **Associativity** - The number of block frames in a set, or the number of block frames where any given block can be stored. The primary caches are direct-mapped (associativity of one) and the associativity of the secondary caches range from one to four.
- **Indexing (Placement)** - The selection of a set using index bits from the address of a reference. If the index bits come from the virtual (real) address, the cache is virtual-indexed (real-indexed). Except for Chapter 6, virtual-indexing is used throughout this dissertation. The virtually-indexed secondary caches use process-identifier (PID) hashing so that common virtual addresses do not index to the same set[1]. Virtual-indexed PID-hashing approximates real-indexing. Note that all the virtually-indexed caches used in this dissertation are tagged with PIDs, and that the cache entries are not flushed at process switches.

1. For each address space (or process) an 8-bit PID is exclusive-ored with the upper (virtual) index bits to choose the set.

- **Block Replacement** - Removing a block from the cache to make room for a new block. In a write-back cache, this may require writing the replaced block into main memory.
- **Block Replacement Policy** - The algorithm that chooses the block that will be replaced when space must be allocated. The default is a random block from the set.
- **Write Policy** - Decides when writes propagate to the next level in the hierarchy. Write-through implies that updates immediately propagate when any write occurs. Write-back implies that updates occur at block replacement. Write-back is assumed in this dissertation[2].
- **Multi-Level Inclusion** - The property that the contents of a cache closer to the processor must always be contained in a cache further from the processor [BAEW88]. It is particularly useful when maintaining coherency in a shared memory multiprocessor or in the presence of I/O devices. Inclusion is not maintained between the primary and secondary caches in this dissertation, except in Section 5.8. Inclusion between the CPU caches and the main memory is maintained in Chapter 6.

The function of a CPU cache is similar to virtual memory: the cache holds a dynamic portion of the blocks that service most memory references. The design of caches is considerably different from the design of virtual memory, however. Hardware manages caches while software manages virtual memory misses. The major reason for this is, as suggested in Figure 1.1, that the time for a CPU cache miss is much less than the time for a page fault. Software intervention is useful and affordable when a page fault costs millions of instructions. Software intervention is too slow for cache misses, so hardware must service them as quickly as possible. A cache is set-associative rather than fully associative. This means that each cache block indexes to a set, and the block can only be stored in the few block frames in that set. Hardware set-associativity implementations allow very fast associative access to blocks stored in the cache.

Figure 2.2 shows the operation of a 2-way set-associative cache. The bottom bits of the address are the offset within a block, much like the page offset. Set-associativity, unlike full page associativity, extracts index bits from the address to choose the set where the block will reside. The index bits are typically in the middle of the address. The cache compares the upper tag bits of the address against other tags in the set. When the cache holds a block, its tag is stored for later comparison. When the tag bits of the address match a stored tag, the block is in the cache (a hit). When a referenced block is not found in the cache (a miss), it replaces another block in its set so that later references will find the block (later references will hit).

Figure 2.2 depicts a two-way set-associative cache because there are two cache memory banks. This dissertation also examines associativities of one and four. With 1-way, or direct-mapped, there is only a single bank, meaning that each block can only reside in a single cache block frame. Direct-mapped is the opposite end of the associativity spectrum from the full associativity of virtual memory. A 2-way or 4-way set-associative cache has more mapping flexibility than a direct-mapped cache

2. Additionally, the cache is write-allocate, meaning that on write misses, the new block is loaded and then immediately updated in the cache.

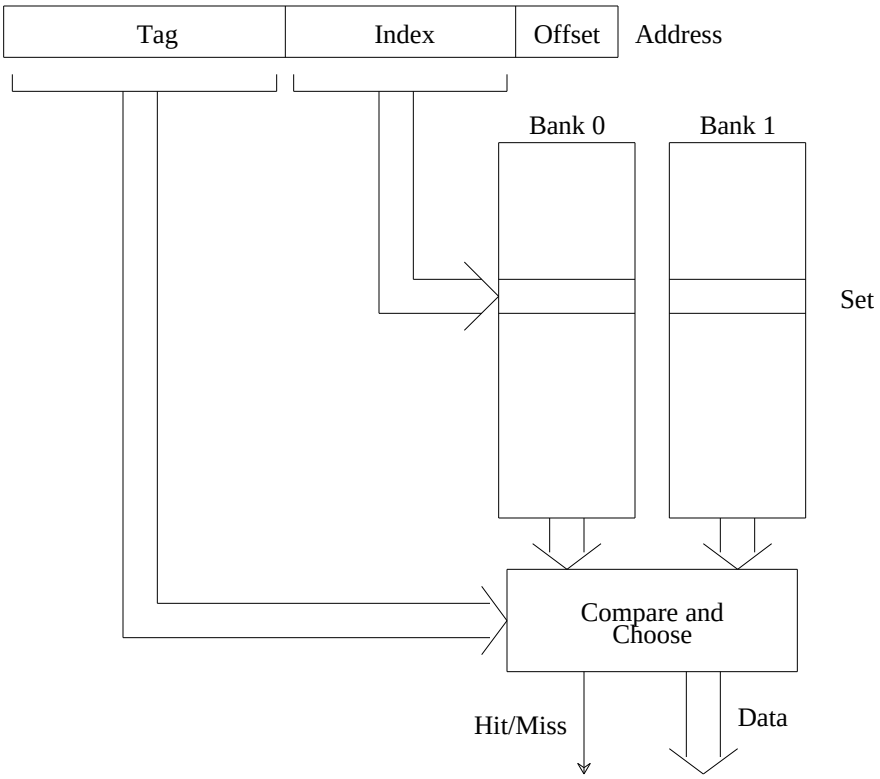


Figure 2.2. Set-Associative CPU Cache Indexing.

This figure shows the operation of a 2-way set-associative cache. The index bits choose the set. The associative matching hardware decides if the cache contains the block referenced by the address.

because there are more frames where a given block can reside. On the other hand, a direct-mapped cache can be faster than caches of higher associativity since its design is simpler.

As with virtual memory page faults, cache misses tend to occur infrequently because of locality of reference. The more often there are misses, the more often the next slower memory in the hierarchy must be accessed. Misses must be sufficiently rare so that their performance degradation is not too high. The cache miss frequency is a key cache performance metric that is estimated throughout this dissertation.

2.4. Previous Cache Analysis

This section surveys many previous cache performance studies. It discusses hardware monitoring and analytical models, and shows that trace-driven simulation is a powerful tool for the performance analysis of cache memories.

2.4.1. Hardware Monitoring

The most precise way to measure cache performance is to take direct measurements by monitoring the operation of a cache. This often involves measurement hardware designed specifically for a given system. For example, Clark, et al., discuss microcode histogram hardware for the VAX 11/780 and 8800 [CLAR83, CLAE85, CLBK88]. The histogram counts determine many cache performance metrics, including cache miss frequencies, misses per instruction, and cycle time effects on the average

instruction. Wood describes the embedded SPUR monitoring hardware, which also measures many cache performance metrics, but is an internal part of the SPUR cache design [WOOD90], rather than a plug-in measurement device.

Hardware monitors give the most accurate cache performance results, but they are costly and can only measure existing cache configurations. It is extremely useful to test a range of cache configurations since trends and fundamental cache performance factors can be determined. Hardware monitoring may not be a cost-effective alternative for the analysis of a wide range of configurations.

2.4.2. Analytical Modeling

Analytical modeling is much less costly than hardware performance monitoring, and is flexible enough to estimate performance over a wide range of configurations. Strecker shows that simple equations can predict cache miss ratios with different process switching [STRE83]. Haikala estimates cache performance with process switching using a Markov Chain model, given the LRU stack distribution [HAIK84]. Smith and Goodman use a loop reference model to predict instruction cache performance and show that set-associativity can be better that full associativity [SMIG85]. Thiebaut and Stone develop analytical techniques to estimate the cache footprint left by an executing process for use in predicting cache performance with process switching [THIS87]. Voldman, et al., [VOMH83] and Thiebaut [THIE88, THIE89] show that fractal analysis can predict cache performance. Agarwal, et al., develop a model that counts start-up, non-stationary, and interference effects on cache performance estimates [AGHH89]. Singh, et al., use empirical curve-fitting to predict cache performance [SIST88]. Higbie develops a cache performance model based on rules-of-thumb [HIGB90].

The problem with these models is that it is difficult to establish confidence in their results without direct comparison to more accurate results. Analytical models are most useful when they increase understanding and intuition. For example, simple rules-of-thumb such as ‘‘The miss rate of a direct-mapped cache of size X is about the same as a 2-way set-associative cache of size X/2’’ [HENP90] or ‘‘Doubling the cache size decreases the miss rate by 25%’’ [HIGB90] are good first-cut approximations, even if they are not always correct. Analytical models may not be appropriate when accuracy is required, but they are useful when more accurate workload models are not available or when simulation time is not affordable.

2.4.3. Trace-Driven Simulation

This dissertation uses trace-driven simulation for cache analysis because simulation overcomes the limitations of hardware monitors and analytical models. Trace-driven simulation uses memory reference traces to model cache behavior. Unlike hardware monitors, a wide range of cache configurations can be cost-effectively examined with trace-driven simulation. Unlike analytical models, simulation produces accurate results, at least to the limits of the available traces. Hundreds of previous cache performance studies have used trace-driven simulation, including Smith’s excellent survey [SMIT82]. In the process, it has been established as powerful tool for cache performance evaluation.

Figure 2.3 depicts the production of trace-driven simulation cache performance results. The idea is to capture the processor-memory references under realistic conditions, then feed the references into a cache simulator. The first, and most important, phase in the process is gathering the trace data from a given workload. This is done by the trace-gathering mechanism, a component of the traced system.

The trace data describes the memory reference behavior of the processor, and may be stored for later use. The simulators of the second phase recreate the memory references using the trace data.

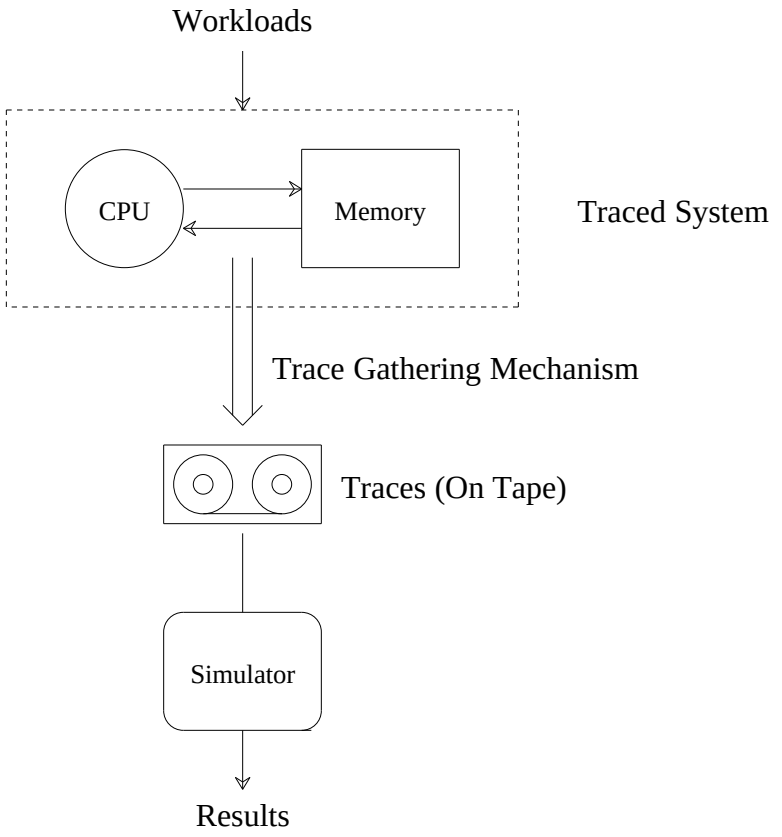


Figure 2.3. Trace-Driven Simulation.

This figure shows the major components of any trace-driven simulation study.

The second phase of trace-driven simulation is the generation of simulation results using the previously captured trace data. Note that while the trace data is gathered only once, it can be used often. The simulator interprets the trace data and gives cache performance results for desired cache configurations. It extracts each reference from the trace and submits them to the simulated cache, just as the real references are submitted to the real cache. The simulator steps through the actions that the cache the cache would execute, so the cache performance estimate of the simulator is an accurate indicator of cache performance for the traced workload. By varying the simulated cache configuration, the simulator can capture the performance of many caches.

The next two subsections describe previous work related to the trace gathering and usage phases of trace-driven simulation.

2.4.3.1. Gathering Address Traces

It is unfortunate that the most important component of any trace-driven simulation study, the traces themself, receive so little attention in the literature. Any trace-driven simulation is only as good as the traces; it is important to gather traces that properly characterize the workloads of the simulated machine. Although all sequences of memory references tend to exhibit locality of reference, the exact memory reference behavior contained in a trace has a large effect on cache performance. Clark, et. al.,

show that trace-driven simulation performance estimates can be optimistic because complete memory reference behavior is not available in the traces [CLAE85]. Smith warns that trace-driven simulation results vary greatly for different architectures and workloads [SMIT85]. It is useful to obtain traces from a variety of workloads to determine their differences and cache performance effects.

Many traces have been gathered by either simulating architectural behavior or stepping through the execution of a program using operating system facilities to control the execution of a program. Entries are written into the trace at each execution step, corresponding to program execution. The simplicity of these techniques is useful, but it is time-consuming to gather these traces, and may be difficult to include operating system or multiprogramming references. Agarwal, et al., argue that hardware microcode modifications can be an effective mechanism to gather traces that include all references, user and operating system [AGSH86, AGAR87, AGHH88]. Altered microcode writes out trace entries for each executed instruction. Since the hardware mechanism requires little software support, operating system references can be traced. Agarwal, et al., show the importance of operating system references to correctly predict cache performance: operating system references are significantly different from user references; they may increase the cache misses in trace-driven simulation by 50% [AGHH88]. While this is an important achievement, the difficulty is that the microcode modifications can be expensive, inflexible, and are only useful for microcoded machines.

Borg, et al., developed an alternative tracing technique that uses code modification to trace programs [BOKL89, BOKW90]. The application code is modified to output trace data at specific points. The main advantage of code modification is that it can be implemented entirely in software, no hardware changes are required. This mechanism gathered the multi-billion-instruction traces described in Chapter 3. Properly interleaved multi-process trace data can be collected by code modification with operating system and compiler support. With the code of the operating system also modified, operating system references can even be gathered. Using similar techniques, Stunkel and Fuchs collect and analyze multicomputer traces [STUF89, STJF91]. Eggers, et al., [EGKK90] describe a tracing scheme that (using compiler data flow analysis) requires fewer trace entries than the two previous schemes, similar to the Abstract Execution of Larus [LARU90]. PFC-Sim also eliminates many entries [CAKP91]. The advantage of fewer trace entries is lower trace storage requirements and less execution time distortion. The disadvantage is that trace data interpretation is more complex.

Much trace data storage is required to collect the long traces needed to analyze multi-megabyte caches. Samples introduced an effective compression technique to reduce trace data storage requirements without information loss [SAMP89]. It combines Ziv-Lempel [WELC84] compression with differencing of the trace data, giving dramatic storage space reductions for traces. A variant of this technique, described in Chapter 3, compresses the traces used in this dissertation.

Techniques with information loss can also reduce trace data requirements. Smith's *stack deletion* and *snapshot method* [SMIT77] eliminate much trace information with only small effects on fully-associative simulation results. Stack-deletion eliminates many references to the most-recently-used items, while the snapshot method eliminates many time-contiguous references. The idea is that many most-recently-used or time-contiguous items will hit in the cache anyway, so there is no need to simulate them; all that is needed is a proper estimate of the number of misses in a trace. Puzak's *trace (tape) stripping* [PUZA85] is stack deletion (restricted to one most-recently-used item) for set-associative caches. References are stripped away by simulating the full trace on a small direct-mapped cache and recording only the misses. Puzak showed that, provided some restrictions are upheld, the stripped trace

produces the same number of set-associative cache misses as the full trace. Wang and Baer subsequently extended trace-stripping for write-backs and multiple block sizes [WANG89, WANB90]. Agarwal and Huffman introduce a technique that eliminates references by exploiting spatial locality [AGAH90].

Sampling techniques also reduce trace data requirements. Their cost is also an information loss. Puzak introduced *set sampling* (congruence class sampling), which eliminates all full trace references except those to the sampled sets. The sampled sets estimate the performance of all cache sets. Laha, et al. [LAPI88] advocate *time sampling*, using many short time-samples of a full trace to estimate cache performance. Individual time-samples can be statistically combined to produce a picture of the full trace. Chapter 4 analyzes both set sampling and time sampling in detail.

2.4.3.2. Cache Analysis Using Address Traces

Many previous studies of uniprocessor cache performance used address traces, far too many to reference them all in this document. For one example, the first paper describing the details of a CPU cache, the one describing the IBM 360 Model 85 cache [LIPT68], included trace-driven simulation results to validate their design choices. In another example study, Kaplan and Winder show the performance tradeoffs of different cache configurations [KAPW73]. A milestone in CPU cache research was reached with the survey of Smith [SMIT82]. This survey examines many basic aspects of cache design, including replacement policy, write policy, block size, and virtual versus real indexing. The block size analysis was later expanded [SMIT87]. Smith also presents an extensive bibliography [SMIT86] of studies on the topic of CPU caches, many of these papers are described in his survey.

Easton and Fagin point out a danger of trace-driven simulation: the cold-start problem [EASF78]. When traces are short, cache initialization effects may dominate performance, particularly with large caches. Since the multi-megabyte caches considered in this dissertation exacerbate the cold-start problem over that of smaller caches, Chapters 3 and 4 pay particular attention to cold start. Except in Section 4.3, all miss frequency results given in this dissertation are cold start, that is, they assume the cache starts from the empty state at the beginning of a simulation. Cold-start biases the results by at most a few percent because the traces that are used are very long.

Recent studies by Hill [HILL87, HILL88] and Przybylski, et al. [PRZY88, PRHH88] characterize the performance of various cache configurations. They point out that higher associativity does not necessarily mean better performance since more associativity may increase the time it takes to retrieve data from the cache. Hill and Smith [HILS89] further analyze associativity by classifying misses as one of three types: conflict, capacity, or compulsory. Conflict misses could be eliminated with higher associativity and serve as a measure of the potential usefulness of associativity. Capacity misses are due to limited cache size, and compulsory misses are due to the first references to cache blocks. Przybylski has also examined the impact of different block sizes and fetch policies [PRZY90].

Przybylski, et al. further point out that there is a limit on the average speed (hits and misses) that can be achieved with a single level cache [PRHH88]. While larger caches have fewer misses, they are slower. Beyond some point, multi-level caching is required to improve average access time.

Short and Levy present simulation results of multi-level cache configurations [SHOL88]. The results show that multiple levels caches give good performance and that write-back caches may give the highest performance. Baer and Wang [BAEW88] present the necessary requirements to implement a very restrictive form of *inclusion* in a cache hierarchy. The study of multi-level cache configurations by

Przybylski, et al., [PRHH89] points out that the frequency of cache misses in a secondary cache is largely independent of the lower level caches provided the cache is sufficiently large compared to the size of the upstream caches (about eight times). They also mention that secondary caches need not be as fast as primary caches (since they are not accessed on all memory references). Higher associativity and larger cache size is more useful in secondary caches since slower data withdrawal can be tolerated to reduce the frequency of misses. Slow implementations of associativity may even be useful in secondary caches, as pointed out by Kessler, et al., [KEJL89]. Wang, et al., [WABL89] present organizational details and simulation results of a multi-level virtual-real cache hierarchy that maintains inclusion. Borg, et al., present simulation results of multi-level cache configurations with multi-megabyte caches, and show that long traces are useful for the analysis of large caches [BOKL89, BOKW90]. Mogul and Borg further consider the costs of context switching in large caches [MOGB91]. Bugge, et al., also analyze design considerations of large caches [BUKB90]. Mudge, et al., also discuss some cache design considerations of a high-performance uniprocessor [MUBB91, OLMB91]. Chapter 5 discusses many of these design considerations for multi-megabyte caches.

2.5. Cache Performance Metrics

With a multi-level cache hierarchy, an analysis of the performance of the configuration can be difficult because there are many components. Each component of the memory system has an impact on overall performance; thus, a performance metric that can isolate the effects of each is needed. Optimization of the performance metric also should correspond to minimization of the total execution time of the workload.

Figure 2.4 depicts the relationship between four different cache performance metrics. Miss ratio, the cache misses divided by the number of times the cache is referenced, is the most commonly used one. Miss ratio is a simple and intuitive measure of the way a cache is performing. Unfortunately, miss ratio is often inadequate for optimizing cache designs; minimal miss ratio does not imply maximal performance because miss ratio does not factor in cache timing parameters (hit and miss times) and usage frequency (how often the cache is referenced).

Effective access time is a measure of the average amount of time required to access a memory location using a cache. It adds timing information to the miss ratio estimate. Effective access time is a superior metric to compare different cache configurations when the timing changes from one configuration to the next one: a lower effective access time implies better performance.

Though effective access time cures one problem with miss ratio, it also may be inadequate, particularly for cache configurations with multiple caches like the multi-level cache hierarchy examined in this dissertation. The problem with effective access time is that it does not consider cache access frequency. Consequently, it may not gauge the effect of a cache on overall system performance. For example, a secondary cache may have a considerably higher effective access time than a primary cache. This does not mean that the secondary caches are performing more poorly than the primary caches. On the contrary, secondary cache misses may still have a smaller effect on the execution speed of the processor because the primary caches are referenced much more frequently.

This dissertation uses the performance metrics shown on the right side of Figure 2.4, *misses per instruction MPI* and *stall cycles per instruction SCPI*, because they correctly factor in the frequency of the cache accesses. This is essential when considering the multiple caches of a multi-level hierarchy. *SCPI* measures the average impact of cache misses on the execution of each instruction. *SCPI* is one

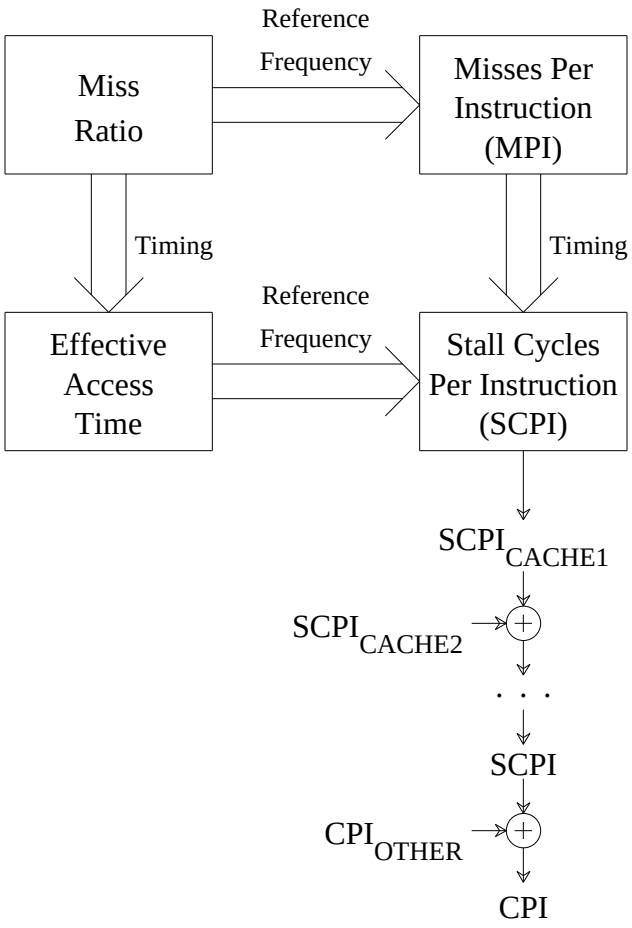


Figure 2.4. Cache Performance Metrics.

This figure shows the relationship between four alternative cache performance metrics.

component needed to estimate *CPI*, or the average cycles required to execute each instruction. (Hennessy and Patterson use *SCPI* to calculate *CPI* in this way [HENP90].) It puts each of the instruction, data, and secondary caches in their proper perspective in the hierarchy as a whole; it allows the performance effects each cache to be compared with the other caches, and with alternative implementations of the same cache.

Chapters 3 and 4 focus on techniques to estimate *MPI*. *MPI* is important because it, along with timing information, is needed to calculate *SCPI*. Chapter 5 adds in timing to the *MPI* estimates, producing *SCPI*, so that alternative cache designs can be compared. This dissertation then reverses itself (from a metrics standpoint), and Chapter 6 focuses again on *MPI*. Chapter 6 considers only software techniques to improve cache performance; the timing analysis is not required because software cannot change the hardware timing.

This dissertation uses a simple *SCPI* calculation. Each CPU cache in the hierarchy adds a component to the total SCPI:

$$SCPI_i = MPI_i \times Tmiss_i. \tag{2.1}$$

MPI_i is the misses per instruction from cache i and $Tmiss_i$ is the average time taken to service miss (in

cycles) in cache *i*, the *miss penalty*[3]. The total *SCPI* for a hierarchical cache configuration is:

$$SCPI = \sum_{i \in HIERARCHY} SCPI_i \tag{2.2}$$

with *i* ranging over the caches in the hierarchy. For the two level configuration in this dissertation (Figure 1.1 on the right):

$$SCPI = SCPI_{ICACHE} + SCPI_{DCACHE} + SCPI_{SCACHE}.$$

It measures the average cycles lost because of cache misses, an important component of the execution time of a traced workload. *SCPI* is closely related to the *CMC* of [BUKB90]. Ideally, *SCPI* should be small relative to the average cycles taken to execute each instruction, so the processor speed determines the execution speed. With a high *SCPI*, system speed would be dominated by the memory system performance, rather than the processor performance.

Note that the calculation of $SCPI_i$ in Equation 2.1 assumes a constant miss penalty. This is an approximation of the true effects of each miss. In particular, the costs of write-backs are not modeled. A *write-buffer* that performs the writes ‘‘in the background’’ is assumed; it allows the processor to continue while the write is being completed. Without a write buffer, the performance loss from dirty misses would be higher since the dirty block needs to be removed before the new one is fetched. With a write-buffer, the processor need only stall when the write-buffer is not successful in hiding the latency of the write. In the *SCPI* calculation, a perfect write-buffer is assumed. This assumption is realistic for large write-buffers [PRHH89].

The weakness with this calculation of *SCPI* is that it does not count some important memory system performance effects. For example, I/O latencies to support virtual memory are neglected to restrict the scope to CPU cache performance. *SCPI* may include miss latencies that could be eliminated by overlapping with other operations. This dissertation does not consider lockup-free caches, which allow miss latencies to be overlapped with other cache hits [KROF81, SOHF91]. It also does not consider that on a cache miss the requested data could be returned first, satisfying a reference early and allowing the processor to continue rather than waiting for an entire cache block to be fetched, which also allows miss latencies to be overlapped. This overlapping could make *Tmiss* a pessimistic estimate of the actual time for a miss; consequently, *SCPI* would be pessimistic. On the other hand, *SCPI* also may be an optimistic estimate of CPU cache performance since contention (because of dynamic memory refresh, or I/O requirements) for memory resources is not factored into Equation 2.1. That is, queueing delay for main memory is not considered. Contention could increase the time for a cache miss, making both *Tmiss* and *SCPI* optimistic.

Despite its shortfalls, SCPI is a powerful CPU cache performance analysis tool for multi-level cache hierarchies. First, it is simple, and can be calculated with only the *MPI* and *Tmiss* of each cache in the configuration. *SCPI* abstracts many implementation details of a real system. The results are not dependent on the issue rate of the processor, for example, so they are equally valid for a superscalar processor as one that can only issue a single instruction per cycle. Despite its abstraction, *SCPI* is a reasonable estimator of the overall performance of a given CPU cache configuration. Another advantage is that breaking *SCPI* into its constituents isolates the importance of each component to the system performance as a whole. The relative magnitude of each component of *SCPI* emphasizes the importance of

3. The miss penalty of the primary caches is the time for a secondary cache hit.

the corresponding cache misses to overall system performance and can be used to direct efforts toward performance improvement. By abstracting details, *SCPI* allows attention to be narrowed to the most fundamental factors in cache performance.

In this study, $Tmiss_{ICACHE} = Tmiss_{DCACHE} = 10$ cycles. $Tmiss_{SCACHE}$ will range from 30 to 200 cycles for different cache configurations, with a default of 100. The default parameters, for example, could correspond to a processor with a 5-nanosecond (ns) cycle time, where 50-ns is required to transfer a 32-byte block from the secondary to the primary caches, and 500-ns is required to transfer a main memory block of 128-bytes to the secondary cache, as depicted in Figure 1.1.

2.6. Statistical Techniques

Statistical techniques are used for two purposes in this dissertation: (1) to measure sampling errors (in Chapter 4); and (2) to establish the level of confidence in results (Chapter 4 and Appendix B).

Chapter 4 uses several sampling error measurement techniques, depending on the situation. When there is only a single estimate that is to be compared to a true value, a good error measure is the relative error: $\frac{(MPI_{estimate} - MPI_{true})}{MPI_{true}}$, where MPI_{true} is the true value and $MPI_{estimate}$ is the estimate. When there are multiple estimates, a better measure is the coefficient of variation of the estimates:

$$CV = \frac{\sqrt{\frac{1}{n}\sum_{j=1}^{n}(MPI_j - MPI_{true})^2}}{MPI_{true}}, \tag{2.3}$$

where MPI_j is the j-th of n estimates. This coefficient of variation is used to measure the average sampling errors in this dissertation when all $n = N$ out of N possible samples are available, or when $\frac{1}{n}\sum_{j=1}^{n} MPI_j = MPI_{true}$. The coefficient of variation is the standard deviation (or the root-mean-squared error) divided by the mean.

When sampling is used in practice, MPI_{true} is estimated with $MPI_{mean} = \sum_{j=1}^{n} MPI_j$, where n is the number of estimates gathered ($n \le N$). To establish confidence that MPI_{mean} is near MPI_{true} without knowing the value of MPI_{true}, it is useful to calculate a 90% confidence interval. Chapter 4 and Appendix B use simple statistical techniques to calculate 90% confidence intervals for MPI_{true}, assuming that the MPI_j estimates are normally distributed and independent[4]. To calculate the confidence intervals, the coefficient of variation of the mean of a sample of size n from a (possibly) finite population of size N is first estimated:

$$CVEST_{mean} = \frac{\sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(MPI_j - MPI_{mean})^2}}{MPI_{mean}\sqrt{n}} \sqrt{\frac{N-n}{N-1}}. \tag{2.4}$$

This equation calculates $CVEST_{mean}$ (the estimated coefficient of variation of MPI_{mean}) similar to the way Equation 2.3 calculates CV, but it reduces it by a $\frac{1}{\sqrt{n}}$ factor because MPI_{mean} is the mean of n

4. The central-limit theorem suggests that the normal assumption is accurate when the sample size (n) is large, or when MPI_j is the sum of many random variables [MILF77].

MPI_j's. $\sqrt{\frac{N-n}{N-1}}$ is a finite population correction factor [MILF77], which is only needed when n is close to N, because in other cases it is close to one. Note that this correction implies that $CVEST_{mean}$ approaches zero as n approaches N because $MPI_{mean} = MPI_{true}$ when $n = N$. (This correction is not needed in Appendix B because N is very large.) If $CVEST_{mean}$ were exact, the normal distribution would be directly used to estimate confidence intervals, but since $CVEST_{mean}$ is an estimate, the student-t distribution is used. Using the student-t, MPI_{true} is bounded (with 90% confidence) below by $MPI_{mean}(1 - t_{90\%,n-1}CVEST_{mean})$ and above by $MPI_{mean}(1 + t_{90\%,n-1}CVEST_{mean})$. $t_{90\%,n-1}$ is the value of the student-t statistic that has a tail of 5% (on each end) for $n-1$ degrees of freedom. For example, $t_{90\%,n-1} = 1.833$ for $n = 10$. Note that the confidence interval is centered around MPI_{mean}. If the MPI_j estimates are nearly normally distributed, or if n is large, MPI_{true} will be within this interval 90% of the time.

2.7. The Trace-Driven Simulation Environment

Trace-driven simulation provides accurate results for a variety of cache configurations, but its cost is large simulation times, particularly for the trace lengths needed with multi-megabyte caches. One alternative for more efficient simulation would be to use an algorithm that produces many results with a single pass through the trace data. This algorithm, called *stack simulation*, was pioneered by Mattson, et al., [MAGS70]. Thompson and Smith extended stack-simulation so that write-back traffic could be calculated, in addition to the miss ratio [THOS89]. Concurrently, Hill and Smith also extended stack simulation in several ways [HILL87, HILS89]: *forest simulation* gathers results for several direct-mapped caches concurrently, and *all-associativity simulation* simulates multiple set-associative caches concurrently. Wang and Baer combined these extensions, introducing an all-associativity simulation algorithm that produces write-traffic results in addition to miss ratios [WANG89, WANB90].

These simulation techniques can reduce simulation times, but they have a cost. One disadvantage is their increased complexity. Also, though stack simulation has been extended for write-back traffic in addition to miss ratios, it still restricts the performance metrics that can be gathered during a simulation. Another disadvantage is their large memory requirements. Memory requirements are particularly important when simulating multi-megabyte caches; they can even dominate other concerns.

An alternative technique for fast simulation takes advantage of the homogeneity inherent in multiple simulations of hierarchical cache configurations. Since the primary cache configuration is constant throughout this dissertation, and its behavior is often independent of the secondary cache configuration, multiple secondary caches can usually be simulated while simulating the primary cache only once. Figure 2.5 depicts this idea. Since the primary caches satisfy most of the references, most of the simulation time is spent on the primary caches. Eliminating these simulations is a big savings. Many secondary simulations can be completed with a single primary simulation. A concern with this implementation is that the simulator requires a large amount of memory. This concern was alleviated by minimizing the storage required to simulate each cache block frame. Usually, only a single word (4-bytes) is required per block, so even a 16-megabyte secondary cache simulation (with 128-byte blocks) needs only 512-kilobytes.

The homogeneity in Figure 2.5 saves much simulation time at no accuracy loss. The same effect could be extracted by saving primary cache misses (and write-backs!) when simulating only the primary caches, and later simulating the secondary caches using the saved references. This extraction is

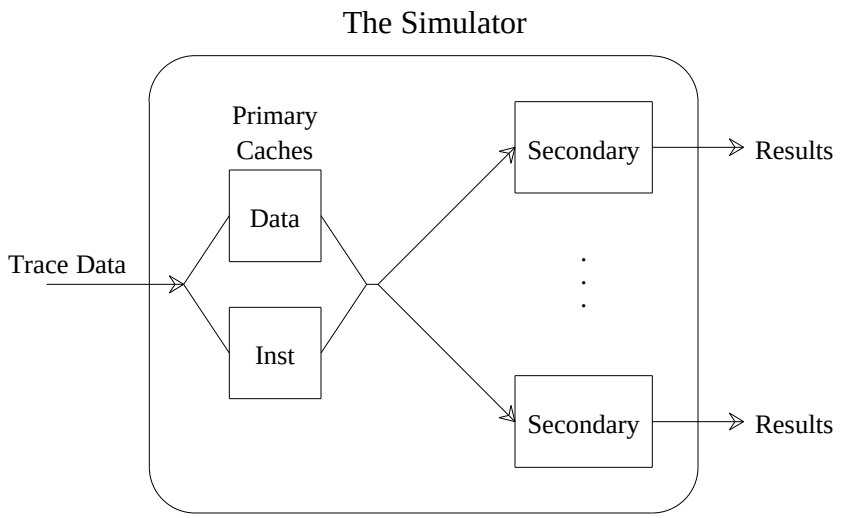


Figure 2.5. Simulation of Homogeneous Hierarchical Configurations.

This figure shows the structure of the simulator. A single pair of primary caches is simulated, rather than one for each secondary configuration. The secondary cache references (from primary cache misses) are submitted independently to each secondary cache, and results are obtained for each.

cumbersome when there are many intermediate traces, as is true for the simulations in Chapter 6, where each trace is modified by many different virtual to real address translations, producing many different traces. Another problem with saving the intermediate traces is that primary cache simulations are not always the same for all hierarchical configurations. For example, when enforcing inclusion between the primary and secondary cache levels (see Chapter 5), the primary cache behavior will be different for each secondary cache, and a brute-force simulation of each two-level configuration is required for precise results. The simulator in Figure 2.5 can adapt to the inclusion case by simulating only a single secondary cache configuration with each primary cache simulation, while an intermediate trace is completely inadequate for this case. Alternative techniques need to be used to speed up simulations when there is no primary cache homogeneity for different secondary caches.

Parallelism across simulations further reduces the simulation time beyond the homogeneity of primary caches. To use this effectively, multiple processors are required. Fortunately, the University of Wisconsin has a powerful tool that allows idle workstations to be used, called Condor [LiLM88]. Condor monitors all the workstations in the Computer Science Department. Users submit jobs to Condor, and the jobs are executed when an idle workstation becomes available. Except during peak hours of the day, Condor was consistently able to provide access to more processors than was needed to complete the simulations needed for this dissertation. The only interruptions in service occur when users return to idle workstations, in which case Condor retreats to a previous checkpoint and restarts the job on another idle workstation.

Figure 2.6 depicts the parallel condor configuration. The arrows show the trace data flow. A monitor process on the single home workstation (Ham-And-Cheese) continuously reads trace data off tape and stages (caches) it in a disk. While executing on idle workstations, the simulators read the trace data off the staging disk and across the network to do the simulations. The monitor process manages the synchronization so that the correct trace data is staged always.

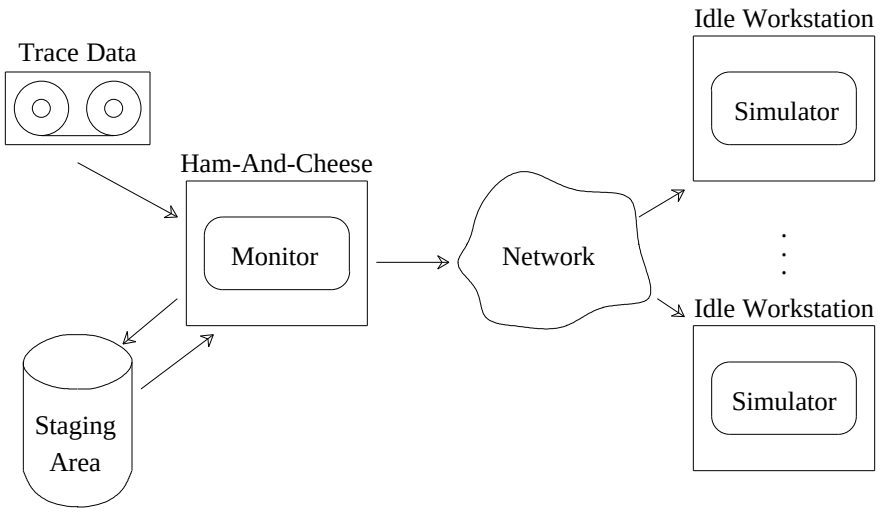


Figure 2.6. Parallel Simulation with Condor.

This figure shows the architecture of the parallel trace-driven simulation environment with Condor. The arrows show the trace data flow.

As many as twenty idle workstations can be used with the Condor environment shown in Figure 2.6. Using the primary cache homogeneity optimization with the parallelism of Condor, results for hundreds of hierarchical configurations could be gathered in nearly the same amount of time needed for a single naive simulation. A large bottleneck in this parallel design is the single processor that is the source of the trace data. Reading the data off the staging disk and transferring it over the network requires much computation. Most often, however, this did not limit the simulation speed. The biggest bottleneck was the single tape drive. It was not feasible to use manual intervention to switch tapes, so each tape would monopolize the tape drive for extended periods. Since only a single trace was stored on each tape, all simulations running at the same time had to use the same trace data. Often, there was not enough parallelism within a single trace to utilize the parallel Condor system fully. Simulations using one tape had to be completed before the next tape was inserted to start a new batch of simulations.

This parallel simulation environment was extremely useful. Without it, many results in this dissertation would not be available. Several years of simulation were condensed to a few months with Condor.

2.8. Conclusions

This chapter first defines and describes key definitions of virtual memory and caches used throughout this dissertation, together with the assumed default parameters. Then, it surveys the state-of-the-art research regarding cache analysis and design. Since memory system performance has such a dominant effect on overall system performance, it has been examined by many previous researchers. The importance of trace-driven simulation has been established. This dissertation uses trace-driven simulation extensively; it overcomes the limitations of previous trace-driven simulation studies. In particular, Chapter 3 discusses traces that remove the length and size limitations of previous traces, and Chapter 4 examines trace-sampling techniques that can reduce trace-driven simulation resource requirements.

This chapter describes the cache performance metrics used throughout this dissertation. In particular, it introduces *MPI* and *SCPI* as metrics that take into account the frequency that a cache is accessed. This is important for comparison and performance optimization with multi-level cache hierarchies. *SCPI* further considers the timing information needed for comparison of alternative cache configurations. This chapter also discusses the statistical techniques used in Chapters 4 and 6. Finally, it describes a parallel trace-driven simulation environment that allows for rapid accumulation of trace-driven simulation results. The environment allows hundreds of simulations to be completed in the same time that only a single simulation could otherwise be completed. Chapters 5 and 6 could not be completed without the simulation capabilities described in this chapter.

Chapter 3

Long Traces For Cache Analysis

3.1. Introduction

Many current (1991) traces are too short. Traces containing one million memory references or less have been used to analyze systems with caches of 128-kilobytes or more, stretching the ability of the traces to provide accurate results because large cache initialization takes too long. This chapter studies the creation and usage of traces capturing billions of instructions. These traces overcome the length limitations of many previous traces and are useful in analyzing the performance of multi-megabyte caches.

This chapter builds on the trace-gathering mechanism developed at The Western Research Laboratory of Digital Equipment Corporation [BOKL89, BOKW90]. This mechanism uses new software-based techniques to gather massive amounts of trace data. Trace compression is added to the trace gathering mechanism so that even billions of instructions can be efficiently stored on tape. The traces come from memory intensive (by the standards of today) workloads, with a working set of 40-megabytes to 100-megabytes. This chapter shows that the traces are suitable for the analysis of caches up to 16-megabytes because they overcome cache initialization and capture many program execution phases.

Section 3.2 of this chapter discusses the trace gathering mechanism and issues related to it, with the trace compression techniques outlined in detail. Section 3.3 presents the traced workloads. Section 3.4 shows general characteristics of the traces, including miss frequencies of multi-megabyte caches. Section 3.5 shows that long traces capture many phases of execution that introduce substantial variations in cache performance, even over execution intervals of 100 million instructions or more. By capturing these phases, long traces accurately characterize the entire execution of a workload, rather than just a portion of the execution. The same section also shows that a trace of a billion or more

instructions may be needed to initialize multi-megabyte cache configurations fully.

3.2. The Trace Gathering and Storage Mechanism

The first part of this section discusses the mechanism used to collect the traces used in this dissertation. The section then describes the data compression techniques used to minimize the storage space required by the traces. It also gives the format of the stored trace data.

3.2.1. Gathering Long Traces

When gathering memory address traces, the end goal is to extract an accurate representation of the memory references of an appropriate workload efficiently. A large part of the difficulty of this process is to capture the trace without distorting the true characteristics of the workload. One would like to include all references in the trace, user and operating system. This gives the most realistic characterization of system memory reference behavior.

It is difficult to gather traces that include all references. One way is to simulate the system. Given the source code for the operating system and the capability to simulate the hardware, the entire execution of the system could be simulated. This simulator could then be modified to write trace data capturing the memory reference behavior of the simulated processor. While it is possible to build such a simulator, it would be complex, and it would run much slower than the real machine. Properly simulating time-dependent events, such as interrupts, could be difficult.

Hardware tracing schemes can gather traces with less execution time distortion. A good example is the ATUM (Address Tracing Using Microcode) scheme [AGSH86]. ATUM modified the microcode of a DEC VAX[5] processor so that entries are written into a *trace buffer* as each memory reference executes. These hardware modifications distort the execution speed of the processor by only a factor of ten when traces are gathered. While ATUM can quickly gather traces that closely represent the true memory references of the processor, a serious restriction of the ATUM approach is the finite trace buffer, which limits the length of the traces that can be gathered. This technique gathered traces of only 500,000 memory references [AGSH86]. ATUM is appropriate with microcoded processors when these lengths are adequate, but it is inadequate for the trace lengths required with multi-megabyte caches. The most critical deficiency of ATUM, however, is that it is completely unacceptable for an unmicrocoded processor, as in this study.

Anita Borg and David Wall at DEC Western Research Laboratory (WRL) developed a software technique (more completely described by Borg, et. al [BOKL89]) to gather traces from the WRL Titan [NIEL86], a ‘‘RISC’’ machine. The Titan runs a modified version of the Unix[6] operating system. The mechanism used to gather the traces was code modification. Changes were made to the system software of the Titan, namely the operating system and the compiler, to gather the traces. The compiler creates traceable versions of applications with a special flag. This special compilation modifies the code so that special procedures, *trace procedures*, execute at appropriate points. These procedures write information in a trace buffer (of 32-megabytes) so that the memory reference behavior of the given program can be reconstructed. Figure 3.1 shows how the expanded code writes trace entries at the beginning of each basic block of instructions[7] and before each load and store. Since loads, stores, and instruction fetches

5. Digital Equipment Corporation Trademark.

6. AT&T Bell Laboratories Trademark.

are the only memory references made by the Titan, each memory reference is accounted for, though the exact positioning of loads and stores within basic blocks is not specified by the trace data[8].

```
Original Code:
     loc1:          Load R3,12(R5)
                    Sub R5,R5,R2
                    Add R4,R4,R3
                    Store 8(R5),R4
                    Branch R5,loc3
     loc2:          Sub R6,R6,R2
                    ...

Expanded Code (Symbolic):
     new_loc1:      Call Trace_Inst(5,loc1)
                    Call Trace_Load((R5)+12)
                    Load R3,12(R5)
                    Sub R5,R5,R2
                    Add R4,R4,R3
                    Call Trace_Store((R5)+8)
                    Store 8(R5),R4
                    Branch R5,new_loc3
     new_loc2:      Call Trace_Inst(8,loc2)
                    Sub R6,R6,R2
                    ...
```

Figure 3.1. Code Expansion For Tracing.

This figure shows an example of the code modifications of a hypothetical machine that would occur when tracing an application by code modification. The original code is at the top and the traceable code is below. Traceable code on the Titan is about a factor of two larger than the original code and executes about ten times slower when tracing.

Tracing by code modification retains many advantages of the ATUM tracing technique, including a modest execution time distortion of only a factor of 10, since code modification on the Titan is similar to microcode modification on the VAX. Both tracing mechanisms write information into a fixed size trace buffer to reconstruct the memory reference behavior of the processor. All traced user processes write in the same trace buffer; the trace is ordered in the sequence that the memory references execute. The difference of this technique from ATUM is that modified code is more flexible and can do many more things than the modified microcode. Carefully crafted trace procedures, with the modified Titan kernel, manage the synchronization among different processes writing in the trace buffer, ensuring that the trace reflects the true user memory reference pattern of the processor under real system workloads. This mechanism gathered traces including the proper execution interleaving of multi-process references for this dissertation. A later version of the mechanism could trace operating system references and user references by code-modifying the operating system to write in the same trace buffer. Unfortunately, this feature was not available at the time these traces were collected. The traces used in this dissertation include only user references.

7. A basic block of instructions is a contiguous block of instructions that are always executed in sequence. Each block has a single entry point and a single exit point.

8. The positioning of loads and stores within a basic block could be determined by analysis of the source code, but no attempt was made to do so.

The ability of the Titan compiler to do instruction scheduling and register allocation [WALP87] allows traceable code to be generated via a flag to the modified compiler. The compiler automatically inserts branches to trace code at the proper points to write the trace. Although the code of a traced program includes the code needed to write to the trace buffer as well as the original code, the basic block trace entries do not include any effect of the instructions inserted for tracing. That is, the code addresses in the trace buffer are those that would have been executed by the untraced program, though the code size increases by a factor of two. This modified-code tracing mechanism is similar to that used by Stunkel and Fuchs [STUF89], Eggers, et al. [EGKK90], and Larus [LARU90]. Since Titan instructions can only be one word, basic block trace entries need only contain the beginning address and the number of instructions in the block. Loads and stores require only a single word in the trace buffer to store the address of the memory access.

Similar to the ATUM tracing technique, the size of the trace buffer limits the length of a contiguous trace. Since a single trace buffer cannot hold long enough traces, it is necessary to concatenate many short traces to produce a long trace, as pictured in Figure 3.2. As the trace buffer becomes full, the operating system stops the traced programs and starts an *analysis program* to read the trace data out of the buffer. Once the analysis program completes, the traced programs continue executing until the trace buffer is again filled. The difficulty in this concatenation of traces is to maintain the fluidity (or seamlessness) of the trace. That is, the concatenated trace should look as if it were a trace extracted from a single (infinitely large) trace buffer. The Titan operating system modifications maintain fluidity because traced user programs do not execute while the analysis program reads data from the trace buffer. Once the trace buffer empties, the traced programs continue tracing from the point where they stopped when the trace buffer became full, so the concatenation of consecutive trace buffers gives a continuous trace that can be billions of instructions or more.

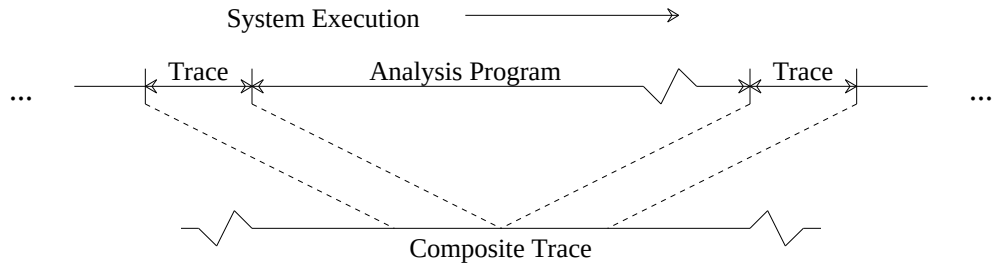


Figure 3.2. Concatenating Multiple Trace Buffers.

This figure shows a single multi-billion-instruction trace resulting from the concatenation of many trace buffers worth of data. System execution consists of periods of trace data gathering interleaved with executions of the analysis program. The analysis program concatenates consecutive trace buffers into a single contiguous trace. Operating System support minimized the distortions introduced by the time-gaps between the tracing.

3.2.2. Storing Long Traces

A version of the analysis program wrote the trace data to tape storage for later use. To minimize the amount of storage required by the trace data, the data was compressed before it was stored. The compression used techniques similar to those used by Samples [SAMP89]. At no information loss, this method reduced trace storage requirements by an order of magnitude. The space required to store the compressed traces varied for different workloads. At the minimum, an average of 0.38 bits was

required to store each memory reference of the traced workloads. Most of the workloads required from two to three bits for each memory reference. This allowed many billions of memory references to be stored on a single 2-gigabyte cartridge tape.

Figure 3.3 shows that the compression scheme separates into two phases, the preprocessing and Unix `compress` phases. The goal of the preprocessing phase is two-fold: (1) to compress most basic blocks into single word trace entries, and (2) to prepare the trace for the `compress` phase by *differencing* current addresses with previous addresses. Rather than storing an entire address in the trace data, a difference can be stored. The preprocessing phase inserts in the trace for the *i*th address, $A(i)$, the difference between address i and address $i-1$:

$$T(i) = A(i) - A(i-1)$$

where $T(i)$ is the trace difference for address i. The decompression algorithm reconstructs the original addresses from the differences using the equation:

$$A(i) = A(0) + \sum_{n=1}^{n=i} T(n),$$

which can easily be calculated by maintaining an accumulator of past values of $T(n)$, as in the following C code fragment:

```
accumulator = accumulator + current_trace_difference;
current_address = accumulator;
```

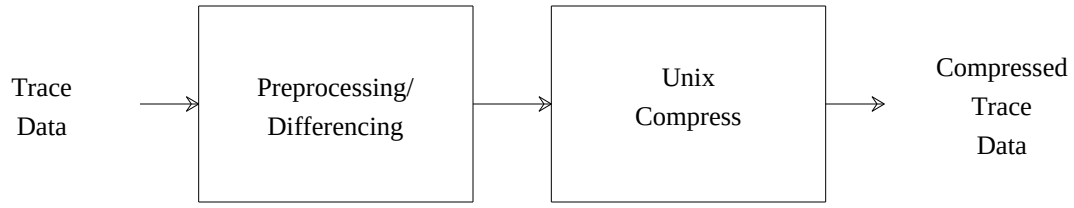


Figure 3.3. The Phases of Trace Compression.

This figure shows the execution phases of the trace compression program. The preprocessing phase combines double word basic block trace entries into single words and does differencing. The output of this phase is then compressed using Unix `compress`, which is extremely effective in reducing the amount of trace data and results in no information loss.

The address differences coming from the preprocessing phase can be regular because of the locality (both spatial and temporal) of memory references. For example, a trace of the differences of memory references is extremely regular when array elements are accessed in sequential order; addresses i, $i+1$, $i+2$, ... would be differenced as 1, 1, 1, ... The preprocessing phase encodes all address as differences: regularity is the key goal of preprocessing.

Unix `compress` is a version of Lempel-Ziv [WELC84, ZIVL76, ZIVL78] lossless compression. The basic idea of `compress` is that it compresses variable length input strings into single output code words. When `compress` recognizes an input string (by comparing it with previous input strings), the associated code word is output, rather than the entire input string. `Compress` is a single pass algorithm that executes in O(n), where n is its input length. The compression ratio of `compress` depends on the length of the input string that can be associated with an output code word: the longer each string is, the more efficient the coding. As shown by Samples, `compress` is very effective with an input of

differenced trace data. Differencing gives a regular input for `compress`, so it is easier to associate long input strings with small output code words.

The preprocessed trace data contains basic blocks, loads, and stores. In the original (unprocessed) trace data, a basic block takes two words. The preprocessor will often compress these <*address, instructions*> pairs into single words. the C code in Figure 3.4 shows the algorithm used to do this. Each <*address, instructions*> pair gets an index in a fixed size, circularly managed, previous basic block array. When a block is found in the array, the full two-word transfer is avoided, and only the single word index is output. In effect, this technique (done in the preprocessing phase) is similar to those used in `compress` to associate input strings with output code words. `Compress` is more general and can encode variable length input strings, whereas, this scheme does fixed-length encoding making use of application-specific knowledge.

```
struct arraystruct save_bblocks[NUMBER_IN_ARRAY];
int saveindex = 0;

Output_Basic_Block_Entry(address, instructions)
int address;
int instructions;
{
    int index;

    index = Associative_Lookup(address, instructions);
    if(index == NOT_SEEN)
    {
        index = saveindex;
        saveindex = (saveindex + 1) % NUMBER_IN_ARRAY;
        Remove_Table(index);
        save_bblocks[index].address = address;
        save_bblocks[index].instructions = instructions;
        Insert_Table(index);
        Long_Basic_Block(address, instructions);
    }
    else
        Short_Basic_Block(index);
}
```

Figure 3.4. Basic Block Compression Code.

This figure shows C code in the compression preprocessing phase to compress most double-word basic block trace entries into single words. Figure 3.5 shows the corresponding decompression code. `Associative_Lookup()`, `Insert_Table()`, and `Remove_Table()` functions (procedures) manage a means to match <*address, instructions*> pairs to an index of the previous basic block array holding the corresponding pair associatively, through hash table [AHU85] or other means. `Long_Basic_Block()` and `Short_Basic_Block()` pass their arguments as trace data to the next phase of compression.

Figure 3.5 shows the decompression code that corresponds to the basic block compression code in Figure 3.4. It builds the array of previously seen <*address, instructions*> basic blocks as they appear. The previous basic block array built during decompression precisely mirrors the array built during compression since it is constructed in exactly the same manner, with exactly the same replacement policy. So, the indices into the basic block array are guaranteed to indicate the same <*address, instructions*> pair in both the compression and decompression phases, so decompression can reconstruct the full trace data.

```
struct arraystruct previous_bblocks[NUMBER_IN_ARRAY];
int previousindex = 0;

Long_Basic_Block(address, instructions)
int address;
int instructions;
{
    int index;

    index = previousindex;
    previousindex = (previousindex + 1) % NUMBER_IN_ARRAY;
    previous_bblocks[index].address = address;
    previous_bblocks[index].instructions = instructions;
    simulate_basic_block(address, instructions);
}

Short_Basic_Block(index)
int index;
{
    simulate_basic_block(previous_bblocks[index].address,
                 previous_bblocks[index].instructions);
}
```

Figure 3.5. Basic Block Decompression Code.

This figure shows equivalent C procedures in the decompression algorithm for the trace data to extract the <*address, instructions*> pairs from the instruction trace data. Figure 3.4 shows the corresponding compression code. These procedures are called when a corresponding short or long instruction entry is seen in the trace data. They call procedures that simulate the instruction memory references.

Table 3.1 shows the eight types of entries included in the preprocessed trace data. Each 32 bit word in the trace data includes a three bit type identifier and the rest of the 29 bits hold different information depending on the entry type. Along with the short and long basic block types, each of which is differenced, there are two types each for loads and stores. The preprocessor partitions loads and stores depending on whether they access the stack or the data (non-stack) area so that the address differencing can extract locality within each. Differencing should produce a more regular output string if it takes the address differences from the same memory segment. Whereas interleaved accesses to both the data and stack areas could result in large, more varying differences that jumped from one area to another, the differences between consecutive references to the stack (and data) would likely be more closely spaced and regular. This splitting of data and stack references required the decompression phase to maintain multiple accumulators, one for the stack and one for data.

The preprocessing phase also includes differenced entries that denote a change between user and kernel mode, or a change to the TLB (Translation Look-aside Buffer). The processor is in user mode during execution of user programs and kernel mode while the operating system is executing. The trace gathering mechanism inserts trace entries each time the processor switches between user and kernel modes. Using these change mode entries, the interpreter of the trace can learn the address space that each reference should be interpreted in (equivalently, which process is executing), and whether the references are from user programs or the operating system. Software manages the Titan TLB, so the operating system is aware each time it changes a slot in the TLB, and it can place entries in the trace buffer to denote the change. The kernel never TLB faults since it uses real addresses; TLB changes only modify the virtual address space of the user processes. A change TLB entry in the trace data

Preprocessed Trace Entry Types	
Type	Description
Long Basic Block	A two word entry specifying the <*address, instructions*> pair describing the traced basic block. The first word in the trace specifies the number of instructions in the block, the second specifies the word address of the block.
Short Basic Block	A single word entry specifying an index into an array of previous long basic block entries. The index extracts the <*address, instructions*> pair that describes a basic block by maintaining previous long basic block entries.
Load Stack	A single word entry specifying the address of a load to the stack area.
Store Stack	A single word entry specifying the address of a store to the stack area.
Load Data	A single word entry specifying the address of a load to a non-stack area.
Store Data	A single word entry specifying the address of a store to a non-stack area.
Change Mode	A single word entry specifying that the processor changed from user to kernel mode or vice-versa. If the processor is changing to user mode, an 8-bit process identifier (PID) is included to determine the execution environment that the following references should be simulated in.
TLB Change	A two-word entry specifying a change in a TLB (translation look-aside buffer) entry. The first word includes the PID and virtual page of the TLB change. The second word denotes whether the entry concerns the data or instruction TLB, whether the entry is to validate or invalidate the TLB entry, where in the TLB the entry should be placed, and what is the real page corresponding to the given virtual page.

Table 3.1. The Types of Preprocessed Trace Entries.

This table shows the eight different types of preprocessed trace entries. They include loads, stores, and basic block, and entries to update the simulated TLB (Translation Look-aside Buffer) and change between kernel and user modes. Addresses are virtual while in user mode, but real addresses are used in kernel mode.

includes the PID (process identifier) and virtual page number to identify a unique page in a given virtual address space. When the operating system validates a TLB slot, the corresponding real page frame number is also included. All TLB slot changes are recorded in the trace data once tracing starts. This allows the interpreter of the trace data to maintain the TLB contents corresponding to each traced process.

Unfortunately, the contents of the TLB while tracing is not enough to know the real address that would have corresponded to virtual addresses when not tracing. Since the trace data includes those addresses that would have been referenced by the untraced code, and the processor executes the traced code, the instruction addresses in the trace are different from those executed by the processor. In other words, the extra code inserted for trace-gathering distorts the page mapping. The mapping for the modified code is in the TLB, but the mapping for the unmodified code may not be, so the real addresses corresponding to the instruction virtual addresses in the trace may not be in the TLB. This makes it difficult, and perhaps impossible, to use the TLB information successfully for virtual to real translation. System behavior, including the virtual to real page mapping, would be different when running the smaller code sizes of the untraced code rather than the traceable code.

The long trace gathering and storage mechanism described in this section gives the ability to trace multi-process user-only references with little distortion, and allows the trace data to be efficiently stored for later use. It allowed traces of several billions of instructions to be gathered and used for this study.

The next section discusses the use of this trace gathering mechanism to extract traces suitable to analyze multi-megabyte caches.

3.3. A Description of the Workloads

In choosing which applications to trace, it is important to keep in mind that the traces will influence the configuration of future CPU caches. It is likely that future workloads will be much like current workloads, although future workloads will surely fully utilize both the large main memories and the faster processors suggested in Chapter 1. Future workloads will execute many more instructions and use much more memory to solve bigger problems. This section discusses traces taken from workloads that predict the larger workloads of future systems.

A suite of applications that are large consumers of main memory (by 1990 standards) form the base of the workloads. All the programs solve important problems. They require large amounts of memory because the problem they are solving is large, not because they are poorly written programs. As subsequently shown, these memory-intensive applications can exercise the multi-megabyte caches examined in this dissertation. A cache simulation study would be uninteresting if the workloads fit comfortably in the caches.

Table 3.2 describes the programs that were traced. Scientific, Computer-Aided Design (CAD), and Compiled Scheme (A LISP dialect) programs form the bulk of the large programs. These are rounded out with some often used Unix utilities, including all phases of C compilation on the Titan. This collection of programs reflects those used in an engineering and research environment, heavy users of computer resources. Each of these programs were compiled into traceable code, with full optimizations enabled (including register allocation), except where noted.

Table 3.3 shows the combinations of the traced programs making up the different traced workloads. The larger programs were uniprogrammed workloads while the smaller programs were grouped into multiprogrammed workloads. The selected workloads range from a scientific program (Sor), to a Scheme (LISP-dialect) program (Tree), and a multiprogrammed workload (Mult1).

3.4. Simulation Results from the Traces

A suite of traces were collected from the workloads shown in Table 3.3, one from each uniprogrammed workload and two from each multiprogrammed workload, for a total of eight traces. The two traces from each multiprogrammed workload are the result of different process switch intervals, as will be discussed in section 3.4.2.

3.4.1. Instruction Mix and Memory Usage

Table 3.4 shows some general instruction statistics on the traces. The lengths of the traces range from three billion to six billion instructions, much longer than previous traces. They represent substantial portions of the execution of the workloads.

The fraction of instructions that are loads is constant at 30% for the traces other than Lin, while the fraction of stores varies considerably among the workloads in Table 3.4. Excluding the Lin trace, the load fraction is more stable than the store fraction for the different workloads. This observation is consistent with data for various ‘‘CISC’’ workloads and architectures [SMIT85]. Though Lin may invalidate the observation, it seems to hold often.

A Description of the Traced Programs	
Program	Description
Make	A Unix program to maintain, update, and regenerate groups of programs. The make program generated a sequence of compiles and loads. It made calls to cc, rm, and cat. Code size: 148 kilobytes.
Cc	The C compiler front end. Its main purpose is to start the C pre-processor, C compiler, Mahler Compiler, and loader. Code size: 45 kilobytes.
Cpp	The C language preprocessor. Code size: 62 kilobytes.
Ccom	The first phase of C compilation on the Titan. Code size: 397 kilobytes.
Mc	The Titan Mahler [WALP87] Intermediate Language Compiler. Code size: 877 kilobytes.
Xld	The Titan loader. Code size: 758 kilobytes.
Cat	Unix utility to concatenate files. Code size: 37 kilobytes.
Cp	Unix utility to copy files. Code size: 37 kilobytes.
Vi/Ex	Unix text editor. Code size: 361 kilobytes.
Ps	Unix utility to read process status. Code size: 127 kilobytes.
Ls	Unix utility to list directory contents. Code size: 127 kilobytes.
Rm	Unix utility to remove a file. Code size: 37 kilobytes.
Tcsh	Unix shell program. Code size: 348 kilobytes.
Magic	A VLSI layout editor [OUHM85]. Magic includes many features not found in other editors including design rule checking, routing, and plowing. Code size: 2.1 megabytes.
Grr	A Printed Circuit Board Router built by Jeremy Dion [DION88]. Code size: 483 kilobytes.
Tv	A Circuit Timing Verifier built by Norm Jouppi [JOUP87]. Code size: 291 kilobytes.
Sor	A Successive Overrelaxation algorithm contributed by Renato De Leone [DEMA88] that uses sparse representations of extremely large matrices. It is written in Fortran, and was compiled without optimizations since the Fortran compiler available was experimental. Code size: 98 kilobytes.
Linear	A program to analyze the power supply of circuits by solving linear systems of equations via sparse matrices [STAH89]. Code size: 102 kilobytes.
Tree	A compiled Scheme [BART89] (a LISP dialect) program contributed by Joel Bartlett that builds a tree data structure and searches for the largest element in the tree. The garbage collection method used by the Scheme compiler is important to the behavior of Tree. The data area is split into two halves, only one of which is used at any given time. Garbage collection occurs whenever one half of the data space has been spent, causing the (clean) data to be transferred to the other half of the data space. Code size: 406 kilobytes.

Table 3.2. A Description of the Traced Programs.

This table contains a description of the programs used to collect the trace data. They consist of CAD, scientific (Fortran), and Scheme (Lisp-like) programs with some common Unix utilities (most notably the collection of programs that are the C compilation environment). The code size listed is that of the untraced code.

A large portion of the variability in the store fraction can be attributed to the overhead of saving registers for procedure calls. Whereas Sor performs few procedure calls, Tree frequently makes recursive procedure calls, so Tree has nearly twice as many stores. Recursion can cause extra register state saving on a machine like the Titan that does static register allocation at compile time. The value of a register may need to be saved each time it is reused by a different procedure instance. The load fraction may not vary as much as the store fraction because register restores are a smaller portion of the loads.

The basic block size is an important metric of the traces since the tracing mechanism only recognizes a basic block rather than each instruction. The larger the basic block, the more inaccurate is the ordering of the instruction references with the loads and stores, though the ordering doesn't matter as

A Description of the Workloads	
Workload	Description
Mult1	A multiprogram workload consisting of: (1) Make C compiling portions of the Magic source code, (2) Grr routing the DECstation 3100 Printed Circuit Board (16 megabytes), (3) Magic Design Rule Checking the MultiTitan CPU chip (20 megabytes), (4) Tree given 10 megabytes of working space solving the same problem as the Tree workload, (5) another Make that largely consists of a call to Xld to load the Magic object code (20 megabytes), and (6) an infinite loop shell of interactive Unix commands (cp, cat, ex, rm, ps -aux, ls -l /*). The trace skipped about the first billion instructions so the larger programs, Grr, Magic, Tree, and Xld, were able to initialize their large data structures and start using them.
Mult2	The Mult1 workload excluding the Xld (Make) run (5) and the Tree program (4). Mult2 has a lower degree of multiprogramming and is smaller than Mult1.
Tv	A uniprogram workload of Tv analyzing the timing of the MultiTitan CPU chip [JODB87, JOTD89]. Tv required 12.5 billion instructions to complete the timing analysis. About the first 10 billion instructions build a very large linked data structure. The final 2-3 billion instructions traverse the structure. The end of the execution of Tv was captured on tape.
Sor	A uniprogram workload of the Sor program doing matrix manipulations on a 800,000 by 200,000 sparse matrix with approximately 4 million (0.0025%) of the matrix entries being non-zero. About the first billion instructions create the large matrices. The rest of the program is the matrix operations. The trace captures a portion of the matrix operations, excluding initialization.
Tree	A uniprogram workload consisting of the Tree program. Tree has two major phases that were traced. About the first half of the instructions build a large tree structure that represents a Unix-like hierarchical directory structure. The rest of the instructions search this tree to find the largest member.
Lin	A uniprogram workload of Linear analyzing the power supply of a register file. Normally, the program tries to minimize the amount of work it must do by combining circuit structures. The trace was collected by disabling some of these combining operations to produce a bigger problem, possibly reflecting the larger problems of the future.

Table 3.3. A Description of the Studied Workloads.

This table consists of a description of the user-only (no kernel references) workloads used in this study. Four workloads are uniprogrammed and two are multiprogrammed workloads. The uniprogrammed workloads consist of the largest programs. Several smaller programs were grouped with some standard Unix programs to produce the multiprogrammed workloads.

much because this dissertation uses split instruction and data primary caches. The average basic block sizes shown in Table 3.4 represent the dynamic average size, that is, it is the average basic block size of instruction entries in the trace data. They are stable for the traces other than Sor. Tree and Mult2 have the smallest average basic block size while Sor has the largest at over 2.3 times the minimum. The Sor results are not too surprising since scientific programs are generally considered to have larger basic blocks than other programs.

Table 3.5 lists the active and referenced memory for each trace. The active memory is an estimate of the average amount of memory that was active (mapped to any traced process) during the trace. The referenced memory is the amount of unique memory (in 128-byte blocks) referenced by the trace. The active memory of the traces ranges from 40-megabytes to 100-megabytes. This is large for 1990, but may be common for the large memories soon to be available.

The total memory referenced by the traces ranges from 7-megabytes to 72-megabytes. For the uniprogrammed traces, the referenced memory is always less than the memory that was actively

Trace	Instructions			Basic Block Size
	Billions	Loads	Stores	
Mult1	3.0	32%	16%	6.5
Mult1.2	3.9	32%	16%	6.5
Mult2	3.6	30%	12%	6.1
Mult2.2	3.7	30%	12%	6.1
Tv	6.0	28%	11%	6.9
Sor	3.8	29%	8%	14.3
Tree	4.0	31%	19%	6.1
Lin	3.6	40%	13%	7.7

Table 3.4. Instruction Information on the Traces.

This table shows the length of the traces used in this study (in billions of instructions), the fraction of instructions that are loads and stores, and the average basic block size (in instructions).

Trace	Memory (Megabytes)	
	Active	Referenced
Mult1	75	56
Mult1.2	75	69
Mult2	40	58
Mult2.2	40	59
Tv	96	72
Sor	62	58
Tree	64	62
Lin	57	7

Table 3.5. Memory Requirements of Traces.

This table shows the active and total memory requirements of the traces (in millions of bytes). The active memory was estimated by examining the status of the traced processes while being traced. The referenced memory is the amount of unique memory (in 128-byte blocks) referenced by the trace.

available for the processes. Note, however, that the active memory of the Mult2 traces is considerably smaller than the referenced memory. Since these traces represent the complete execution of many processes, the total referenced memory can be larger than the amount of memory that was active at any moment since active processes die and new ones restart.

3.4.2. The Multiprogrammed Traces

An important statistic of each multiprogrammed trace is the process switch interval, or the number of instructions that are executed by a process before another process uses the processor. Table 3.6 shows the switch intervals for the multiprogrammed traces. The target process switch interval is the maximum number of instructions that could be executed between process switches. There are several reasons why the actual process switch interval (measured in instructions executed) for these traces is less than the target: (1) The Titan may not be able to execute at its peak rate because of cache misses, data dependencies, etc.; (2) The operating system execution time (when there is no tracing) subtracts from the interval; and (3) Processes wait for I/O or other external events, relinquishing control of the processor before the full interval is used. The result is that the measured process switch interval in each trace is about half the target.

Trace	Switch Interval (instrs.) Target	Actual	Processes Active
Mult1	200,000	138,000	6
Mult1.2	400,000	196,000	6
Mult2	200,000	134,000	4
Mult2.2	400,000	214,000	4

Table 3.6. Process Statistics of Multiprogrammed Traces.

This table shows the target and actual process switch interval and an estimate of the number of active processes at any moment in the multiprogrammed traces. The target process switch interval was the maximum number of instructions that could be executed between each switch. It was measured using a simple infinite loop program designed to execute instructions rapidly.

The process switch intervals of the multiprogrammed traces, hundreds of thousands of instructions, are high compared to some previous traces. For instance, the ATUM traces average around 10 to 20 thousand memory references between each switch [AGHH88]. The increase in process switch interval is appropriate because of increasing processor speeds, since the time between process switches tends to be constant across different systems. Thus, as the speed of the processor increases, the number of instructions executed between each switch will increase. For example, Clark, et al., show that the context switch headway for the VAX 8800 (19000) is about three times that of the VAX 780 (6000), a nearly linear increase with processor performance [CLBK88]. The switch intervals for the traces were adjusted to values that should be appropriate for future high-performance processors. Mogul and Borg used similar switch intervals [MOGB91].

Table 3.6 shows that the multiprogrammed traces represent the concurrent execution of many processes. Though only about four or six processes are active at any given moment, there are references from many more included in the trace. For example, Mult2.2 has only four processes estimated active at any moment, yet the trace contains the execution of 176 different processes. This occurs since many processes complete and new ones start in their place. Furthermore, though only four or six processes are active at any given moment, some are suspended waiting for other processes to complete. For example, the Cc program does nothing other than start and wait for the Cpp, Ccom, and Mc processes in succession.

To understand more fully the processes included in these traces, this section examines the Mult2.2 trace in more detail. Figure 3.6 partitions the Mult2.2 processes by the number of instructions they execute. The left graph shows that most of the processes execute for about 100,000, 10 million, or 100 million instructions. By far the largest share of the instructions in the trace are from processes that execute for ten million instructions or more. Only two processes (probably Magic and Grr) execute for more than 100 million instructions during the trace. The right graph shows that these two processes contribute the most instructions to the trace. Though small in number, the long-running processes have a dominant effect on the trace.

A similar scenario exists when the processes are partitioned by the amount of memory they reference, as in Figure 3.7. The left graph shows that most of the processes reference from 100-kilobytes to 1-megabyte. The right graph shows that these 1-megabyte processes contribute the largest portion of the instructions in the trace, but not by much. The larger processes contribute a substantial portion, though there are few of them. For example, only a single process references more than 10-megabytes of memory, yet it contributes over 25% of the instructions.

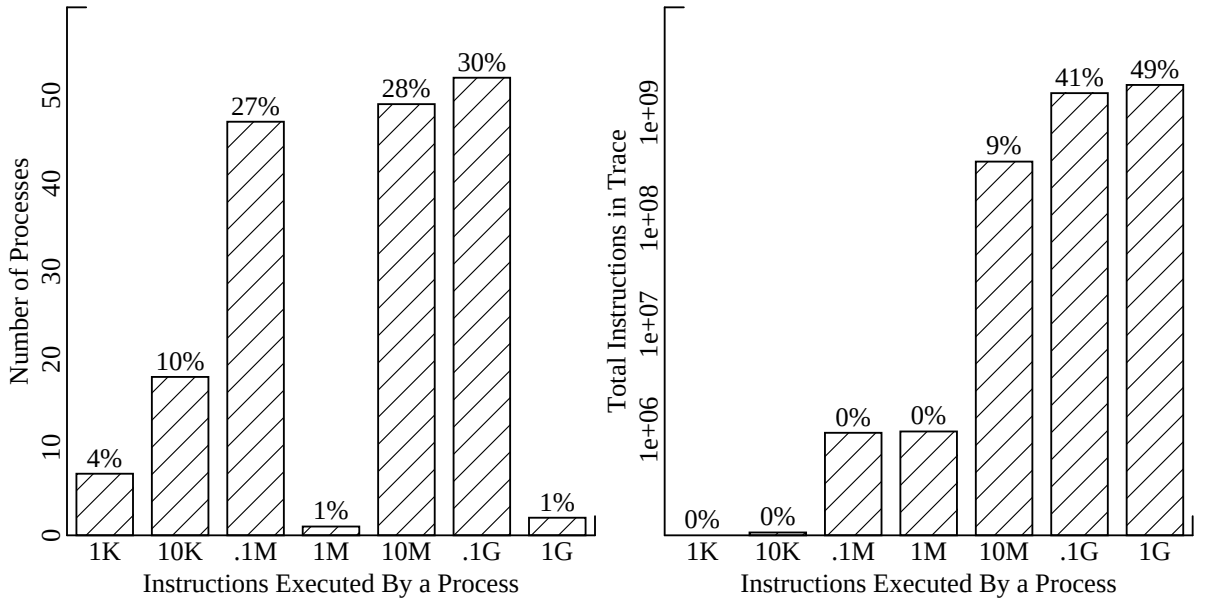


Figure 3.6. Mult2.2 Processes Classified by Execution Length.

This figure partitions the processes in the Mult2.2 trace by their execution length (in instructions). On the left, the number of processes in each class is shown. On the right, the number of instructions executed by the processes in each class is shown. The partitions are by factors of ten. For example, the bar labeled ''1M'' has statistics for processes that have 100,000 to one million instructions included in the trace. Note that some scales are logarithmic.

3.4.3. Miss Frequencies in Multi-Level Cache Configurations

This section expresses shows *MPI*'s (and relative *MPI* changes) for the base two-level configuration that is shown in Figure 1.1 and further described in Section 2.3, with default parameters also given in Section 2.3. By showing the cache performance effects of the traces, this section exposes the trace characteristics that effect cache design. Chapter 5 delves further into multi-megabyte cache design issues.

Table 3.7 shows the miss frequencies of the primary instruction and data caches for the different traces. With this primary cache configuration, the data cache has a higher *MPI* than the instruction cache.

The *MPI* results in Table 3.7 provide some insight into the traced workloads. For the instruction cache, the multiprogrammed traces have higher *MPI* than the uniprogrammed traces. The opposite is true for the data cache. The good instruction locality and poorer data locality of the uniprogrammed traces is likely a result of the scientific nature of several of the applications. Tight loops ranging over large amounts of data will tend to give lower instruction cache and higher data cache miss frequencies. This is exactly the behavior of the Sor and Lin traces, which have virtually no instruction cache misses.

A comparison of the results from the multiprogrammed traces with different process switch intervals shows a slight tendency toward lower *MPI*'s with the longer switch intervals. This is expected since each process switch results in a reloading of the primary caches; the data of the other processes corrupted the primary caches since the last time the process ran [SMIT82]. Although hundreds of thousands of instructions execute between each process switch, switching-induced cache reloading is

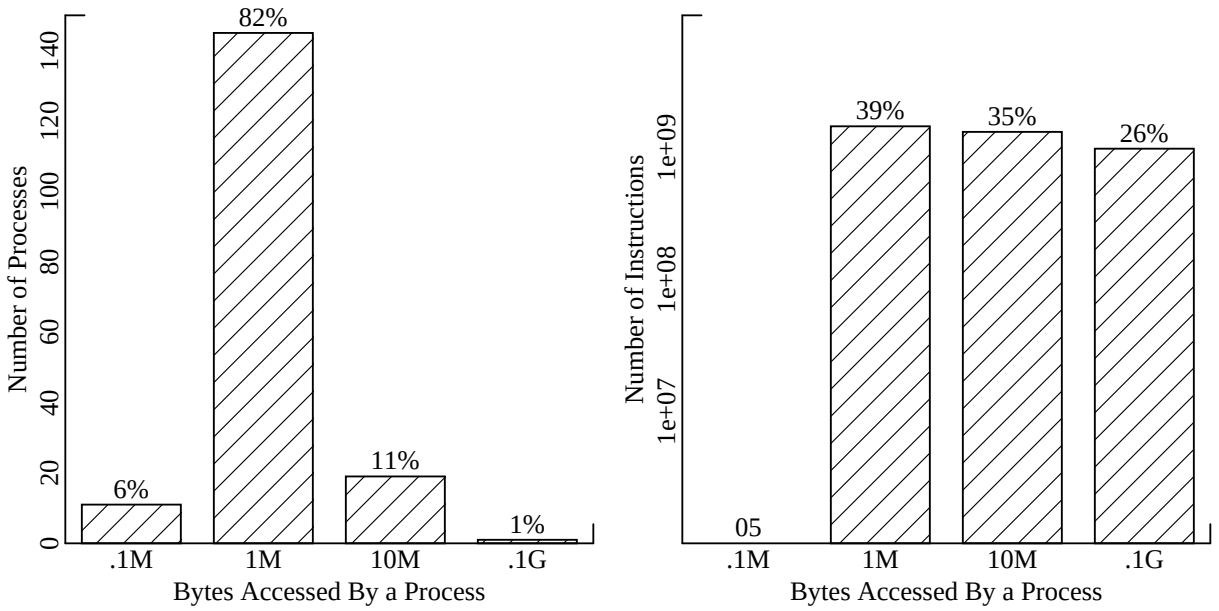


Figure 3.7. Mult2.2 Processes Classified by Memory Usage.

This figure partitions the processes in the Mult2.2 trace by the memory they use. On the left, the number of processes in each class is shown. On the right, the number of instructions executed by the processes in each class is shown. For example, the bar labeled ‘‘1M’’ has statistics for processes that access 100,000 to one million bytes. Note that some scales are logarithmic.

MPI×1000 **of Primary Caches**		
Trace	Instruction	Data
Mult1	5.5	9.7 (37%)
Mult1.2	5.8	8.1 (33%)
Mult2	6.1	8.2 (37%)
Mult2.2	5.5	7.4 (36%)
Tv	2.6	17.4 (8%)
Sor	0.0	22.9 (60%)
Tree	4.9	12.4 (21%)
Lin	0.0	3.7 (3%)

Table 3.7. Primary Cache MPI.

This table shows the miss frequency, expressed as misses per one thousand instructions (*MPI*×1000) for the split primary instruction and data caches for each trace. For the data cache, the fraction of misses that cause the write-back of a dirty cache block is also given in parentheses. As mentioned in Section 2.3, the instruction and data caches are each 32-kilobytes with 32-byte blocks.

still a factor in the performance of the caches. The Mult1.2 trace does have a slightly higher instruction cache *MPI* than Mult1, although Mult1.2 has a longer switch interval. This anomaly is likely a result of the different execution phases of the Mult1 workload covered by the two traces. The Mult1.2 trace captures a early portion of the workload that Mult1 does not.

Table 3.7 also shows the fraction of data cache misses that require writing back a dirty cache block. With write-back caches, writes are not immediately propagated to the secondary cache. Instead, the cache saves the value of the updated cache block until it must be replaced (as the result of a cache miss). Only then is the updated value of the cache block transferred to the secondary cache. The

fraction of data cache misses that cause these write-backs varies substantially across the traces, from a low of only 3% with the Lin trace up to 60% for the Sor trace. The multiprogrammed and Tree workloads display moderate behavior, with about 30% of the cache misses requiring a write-back. The Lin and Tv workloads expend a large portion of their references reading data, and only update small portions of it. The Sor trace modifies the largest portion of the cache blocks in the data cache, though Table 3.4 shows that it has the lowest fraction of instructions that are stores. This means that the locality of writes is much lower with the Sor trace; it modifies small portions of many cache blocks, increasing the write-back frequency of the cache.

Table 3.8 shows the *MPI* of a variety of secondary caches, with sizes ranging from 256-kilobytes to 16-megabytes and associativities ranging from direct-mapped to 4-way associative. The results vary substantially across workloads, just as with the primary caches. This validates the results shown by Smith [SMIT85]. Many workloads like those in this study will give the best conclusions, much better than a single workload. Had resources allowed it, even more workloads would have been used in this study.

Sor has higher *MPI*'s than the other traces. This is consistent with the observation that scientific workloads often have poor cache performance because of poor locality of reference. Sor makes frequent traversals through extremely large array structures that represent sparse matrices. Its locality is poor since the traversals through the array purge cache entries before they can be reused. Associativity does not help Sor's *MPI* much; it actually increases *MPI* for some larger caches. Smith and Goodman show that associativity can be detrimental to cache performance with looping references [SMIG85], as when arrays are being traversed. These results corroborate this observation. Chapter 5 further shows the abnormal cache effects of Sor.

The 16-megabyte caches give extremely low miss frequencies, often well below one miss per thousand instructions. This is not surprising since the 16-megabyte cache is extremely large, larger than many current main memories. Indeed, the 16-megabyte cache is large enough to hold from 15% to 50% of the 40 to 96 megabyte workloads. Without the focus of this study toward larger workloads, these caches would likely not have been exercised as well. Though the cache is large, the same trends that hold for the smaller caches also holds for the larger cache. For example, the results in Table 3.8 show that *MPI* is approximately halved each time the cache size is quadrupled. (Equivalently, the miss ratio decreases by 30% each time the cache size doubles [STON90].) This is as valid for the 16-megabyte cache as for the other caches. Thus, the 16-megabyte cache is not so large that misses never occur, it is just large enough so that the *MPI* is low. Chapter 5 shows that low miss frequencies may be necessary for good performance when the cache miss penalty is large. Of course, if the miss penalty is smaller, or if smaller workloads are used, small caches may do sufficiently well and 16-megabyte caches may not be needed.

For the secondary caches, similar to the case with the primary caches, the Mult2 traces show a consistent reduction in *MPI* with increasing process switch interval. For the 256-kilobyte cache, and for the direct-mapped caches, the reduction with increasing interval also holds for the Mult1 traces. The Mult1 traces do not follow this pattern for the larger, more associative caches, however. Again, this anomaly is likely because of the differences in the phases of execution captured in Mult1 and Mult1.2.

Table 3.8 also shows the fraction of secondary cache misses that cause the write-back of a dirty cache block. Similar to the data cache results, the fraction varies widely from trace to trace. For example, it ranges from 1% to 73% for a 1-megabyte 4-way set-associative cache. Usually, it is near 30% to

MPI×1000 **For Secondary Caches (Direct-Mapped)**				
Trace	Secondary Cache Size			
	256K	1M	4M	16M
Mult1	3.49(25%)	1.55(27%)	0.70(38%)	0.33(47%)
Mult1.2	3.14(26%)	1.45(29%)	0.69(39%)	0.32(50%)
Mult2	3.16(31%)	1.24(33%)	0.61(40%)	0.26(48%)
Mult2.2	2.82(33%)	1.18(35%)	0.59(41%)	0.27(48%)
Tv	6.13(10%)	2.63(14%)	1.88(16%)	1.03(21%)
Sor	19.44(61%)	14.77(72%)	7.54(73%)	1.97(44%)
Tree	5.13(11%)	2.16(17%)	0.59(45%)	0.30(69%)
Lin	1.39(2%)	1.16(1%)	0.09(7%)	0.02(0%)

MPI×1000 **For Secondary Caches (2-Way)**				
Trace	Secondary Cache Size			
	256K	1M	4M	16M
Mult1	2.95(23%)	1.19(31%)	0.55(43%)	0.26(52%)
Mult1.2	2.62(25%)	1.18(31%)	0.56(43%)	0.28(55%)
Mult2	2.38(29%)	1.01(35%)	0.52(42%)	0.24(49%)
Mult2.2	2.15(31%)	0.98(36%)	0.51(43%)	0.22(51%)
Tv	4.46(12%)	2.31(15%)	1.76(17%)	0.98(21%)
Sor	18.84(63%)	14.66(73%)	7.76(75%)	1.92(44%)
Tree	4.59(10%)	1.81(17%)	0.49(52%)	0.26(80%)
Lin	1.33(1%)	1.10(1%)	0.06(13%)	0.02(0%)

MPI×1000 **For Secondary Caches (4-Way)**				
Trace	Secondary Cache Size			
	256K	1M	4M	16M
Mult1	2.79(23%)	1.07(33%)	0.52(44%)	0.26(52%)
Mult1.2	2.50(25%)	1.10(32%)	0.53(45%)	0.27(56%)
Mult2	2.17(30%)	0.95(35%)	0.50(43%)	0.23(51%)
Mult2.2	1.99(31%)	0.93(37%)	0.49(44%)	0.22(51%)
Tv	3.69(13%)	2.28(15%)	1.76(17%)	0.96(22%)
Sor	18.68(63%)	14.55(73%)	8.00(76%)	2.03(45%)
Tree	4.35(10%)	1.75(17%)	0.47(54%)	0.25(81%)
Lin	1.31(1%)	1.08(1%)	0.04(23%)	0.02(0%)

Table 3.8. Secondary Cache MPI.

This table shows the cache miss frequency (expressed as misses per thousand instructions) for direct-mapped (top), 2-way set-associative (middle), and 4-way set-associative (bottom) secondary caches. In parentheses, the fraction of cache misses that cause the write-back of a dirty block is also shown.

50%. It often increases with cache size and associativity. This is precisely when the cache *MPI* decreases and blocks are retained for longer periods of time in the cache. The increasing fraction of write-backs likely occurs because longer block residence times increase the probability that a trace writes a block while it is resident. This is not always the case, however. The fraction decreases for the Sor trace as the cache size increases from 4-megabytes to 16-megabytes. Ultimately, for all workloads, the fraction of misses that cause write-backs has no clear trends for different associativities and cache sizes, though it tends to increase with lower miss frequencies.

In addition to the absolute miss frequencies as given in Table 3.8, relative performance differences also expose the cache performance effects and differences of the traces. This analysis answers

questions such as: What is the reduction in *MPI* produced by increasing associativity from direct-mapped to 2-way with the different workloads? What is the reduction in *MPI* produced by doubling the cache size? The answers to these questions help to decide, for each trace, whether a larger cache with a slower access time will be better than a smaller cache with a higher *MPI*.

Figure 3.8 shows the *miss reduction* that occurs when doubling the associativity. This is the fraction of the misses in the cache of lower associativity that were eliminated in the cache of higher associativity. The different boxes show the miss reduction occurring when the associativity is doubled from direct-mapped and 2-way, respectively. The increase from direct-mapped to 2-way eliminates a substantially larger portion than 2-way to 4-way does. This is consistent with the observations of others, including Hill and Smith [HILS89], who measure ''miss ratio spread''[9]. The advantages of doubling the associativity decreases with increasing associativity.

Again, the Sor trace stands out in the data in Figure 3.8, showing negative reductions because of its looping behavior. In general, the uniprogrammed traces seem to gain less from increasing associativity than do the multiprogrammed traces. This can be attributed to the higher locality of reference of the multiprogrammed traces. The uniprogrammed traces also have a considerably wider range of behaviors: the miss reduction in going from direct-mapped to 2-way ranges from 8% to 25% with the multiprogrammed traces while the same range is -3% to 34% for the uniprogrammed traces. This is due to the similarity of the multiprogrammed traces, and the averaging effect of including many different programs in them.

Figure 3.9 shows the miss reductions when doubling the size of direct-mapped caches. Over the range of cache sizes, the multiprogrammed traces show a reduction of 30% or more. The uniprogrammed traces again show a higher range of behavior than the multiprogrammed traces, but, the mean reduction is similar.

The comparison of Figure 3.8 with Figure 3.9 shows that doubling the cache size eliminates more misses in a direct-mapped cache than doubling the associativity. This does not follow the 2:1 ''rule of thumb'' that the doubling of either the size or associativity of a direct-mapped cache produces about an equivalent miss reduction [HENP90]. Instead, these virtual-indexed results show cache size increases to be more important in reducing the *MPI* than associativity increases. Section 5.6 of Chapter 5 examines why the 2:1 rule may not hold for multi-megabyte caches.

3.4.4. Secondary Cache Inter-Miss Distributions

Beyond the *MPI*, the inter-miss distribution is important. This distribution gives the likelihood that the number of instructions executed between consecutive misses is a given value. It characterizes the timing of the secondary cache misses (i.e. main memory accesses). Figures 3.10a, 3.10b, and 3.10c, display the probability density curves of the inter-miss times for direct-mapped secondary caches[10].

9. The miss ratio spread measures the amount that the higher-associativity *MPI* increases when the associativity is halved. If x is the miss reduction for doubling associativity, the miss ratio spread (for halving) is x/(1 - x).

10. Figures 3.10a, 3.10b, and 3.10c, only show the inter-miss intervals that are less than 500 instructions. For the 1-megabyte 4-megabyte, and 16-megabyte caches (respectively), this captures 79-83%, 72-73%, and 57-61% of all Mult inter-miss times, 91%, 89%, and 84% of the Tv inter-miss times, 93%, 87%, and 52% of the Sor inter-miss times, 81%, 56%, and 23% of the Tree inter-miss times, and 76%, 54%, and 13% of the Lin inter-miss times.

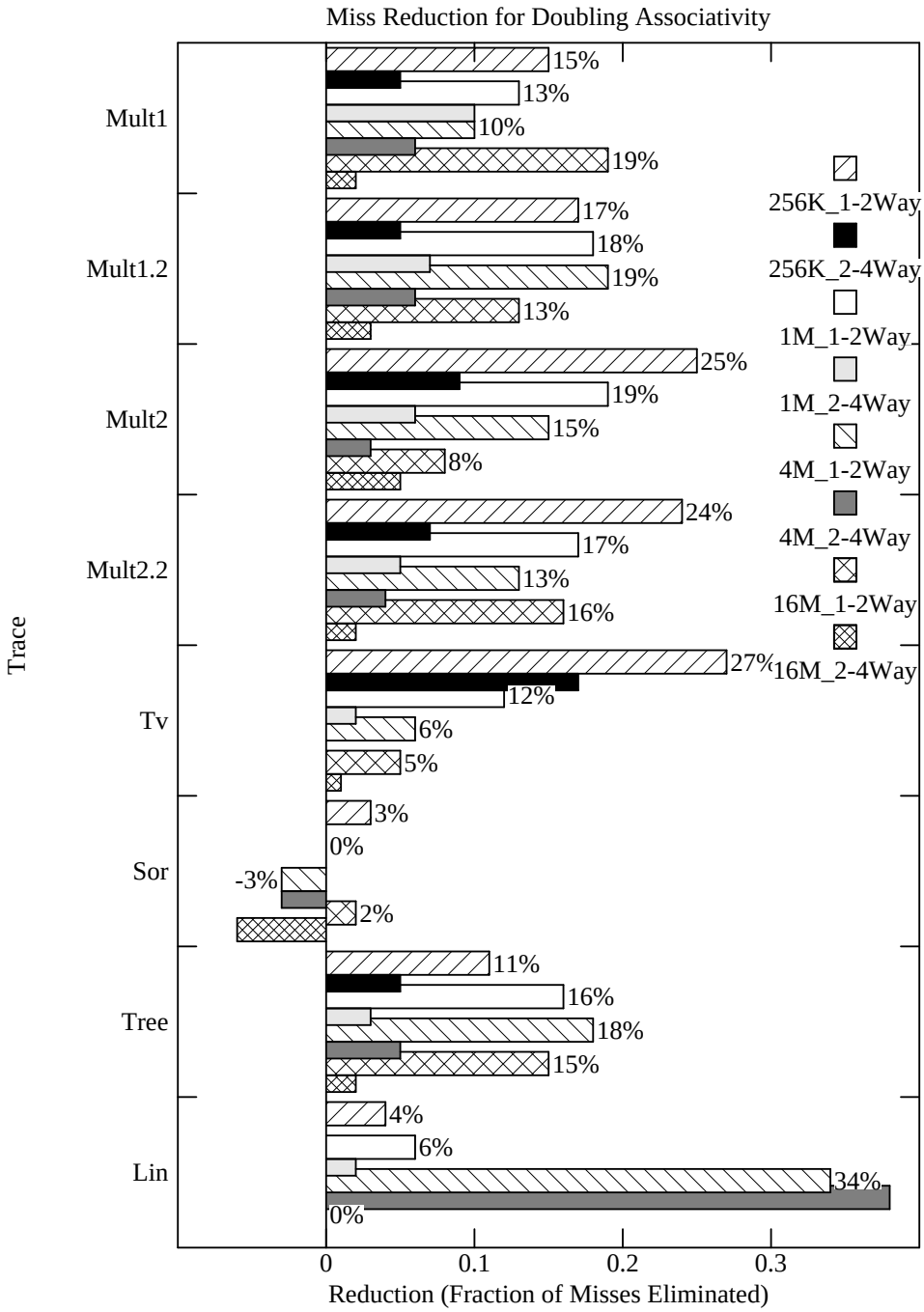


Figure 3.8. Miss Reduction for Doubling Associativity.

This figure shows the fraction of misses eliminated by doubling the associativity from direct-mapped to 2-way (1-2) and 2-way to 4-way (2-4) for each trace and several different secondary cache sizes.

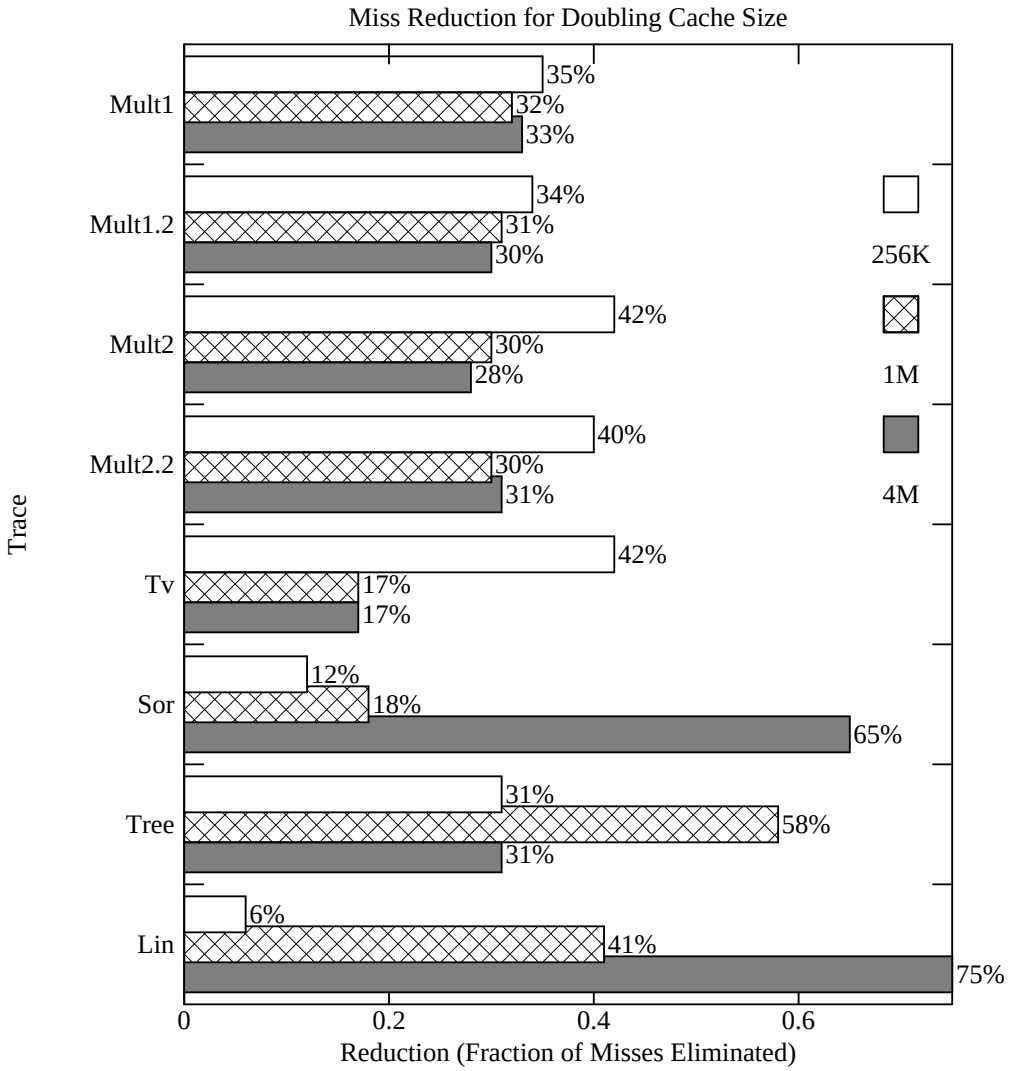


Figure 3.9. Miss Reduction for Doubling Size.

This figure shows the fraction of misses that could be eliminated by doubling the cache size for the various traces. For example, the entry for a 1-megabyte cache denotes the fraction of misses that would be eliminated by instead having a 2-megabyte cache for that trace. These results are shown for secondary cache sizes ranging from 256-kilobytes to 4-megabytes.

Figures 3.10a, 3.10b, and 3.10c show that the short inter-miss intervals occur frequently, particularly with the multiprogrammed workloads. For example, 1450 instructions of the Mult1.2 workload execute on average between each 4-megabyte miss, yet 50% of the inter-miss intervals are less than 100 instructions. This shows that the misses tend to be clustered in time. The density curves are shown out to an interval of 500 instructions, which captures most of the intervals. While there are enough long intervals to balance out short ones, the short intervals are the largest portion of the misses.

The multiprogrammed traces have inter-miss distributions that are more skewed toward the shorter intervals than the uniprogrammed traces. Process switching may lead to short inter-miss intervals when process state is reloaded at each switch. The multiprogrammed traces also may have many

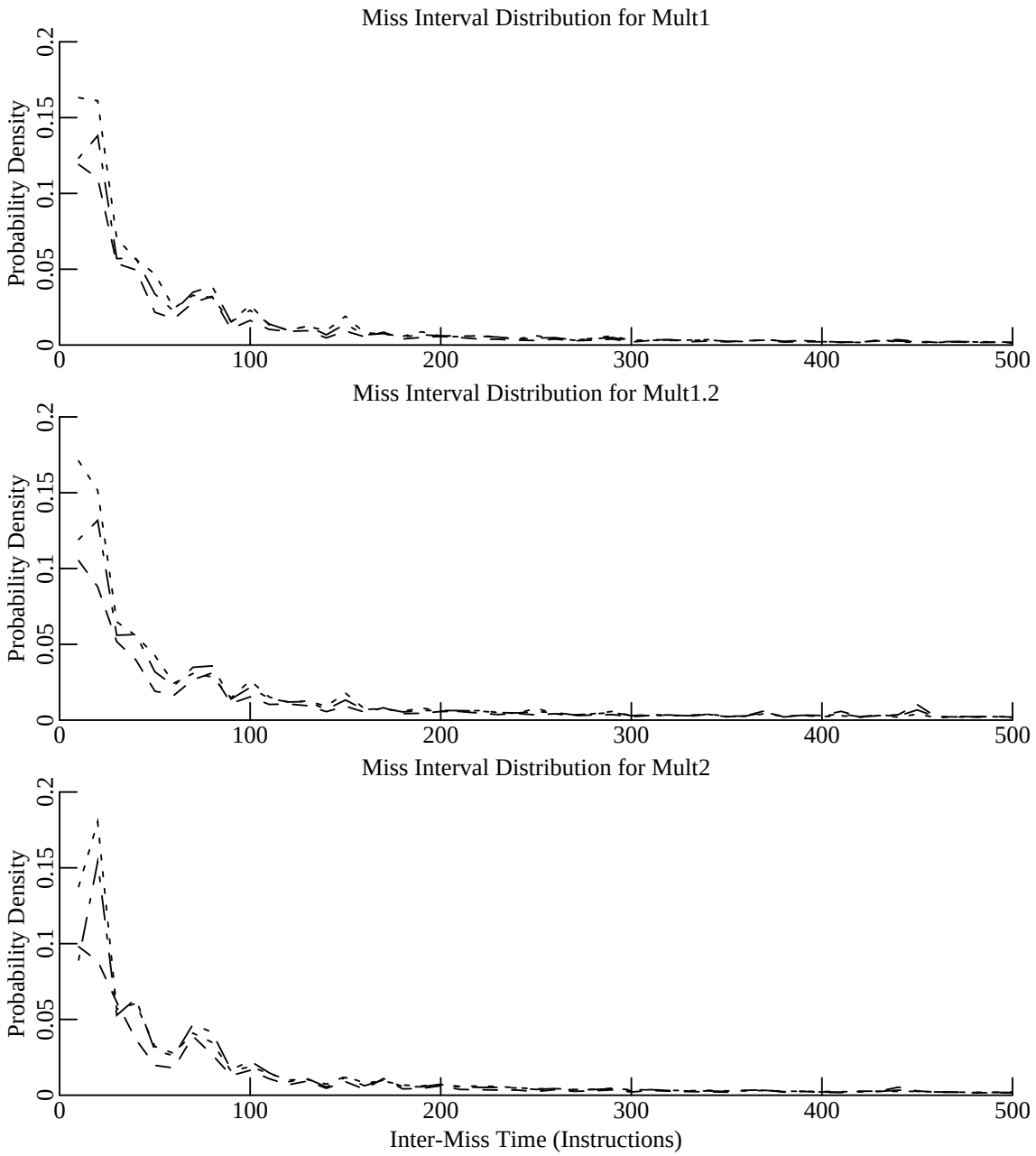


Figure 3.10a. Inter-Miss Density of Mult1, Mult1.2, and Mult2.

This figure shows the inter-miss probability density of the Mult1 (top), Mult1.2 (middle), and Mult2 (bottom) traces for 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes. Each line denotes the probability that the inter-miss interval (measured in instructions executed) occurs. The probability is aggregated over intervals of 10, thus there is a point every 10 instructions on the x-axis. Note that the scales change.

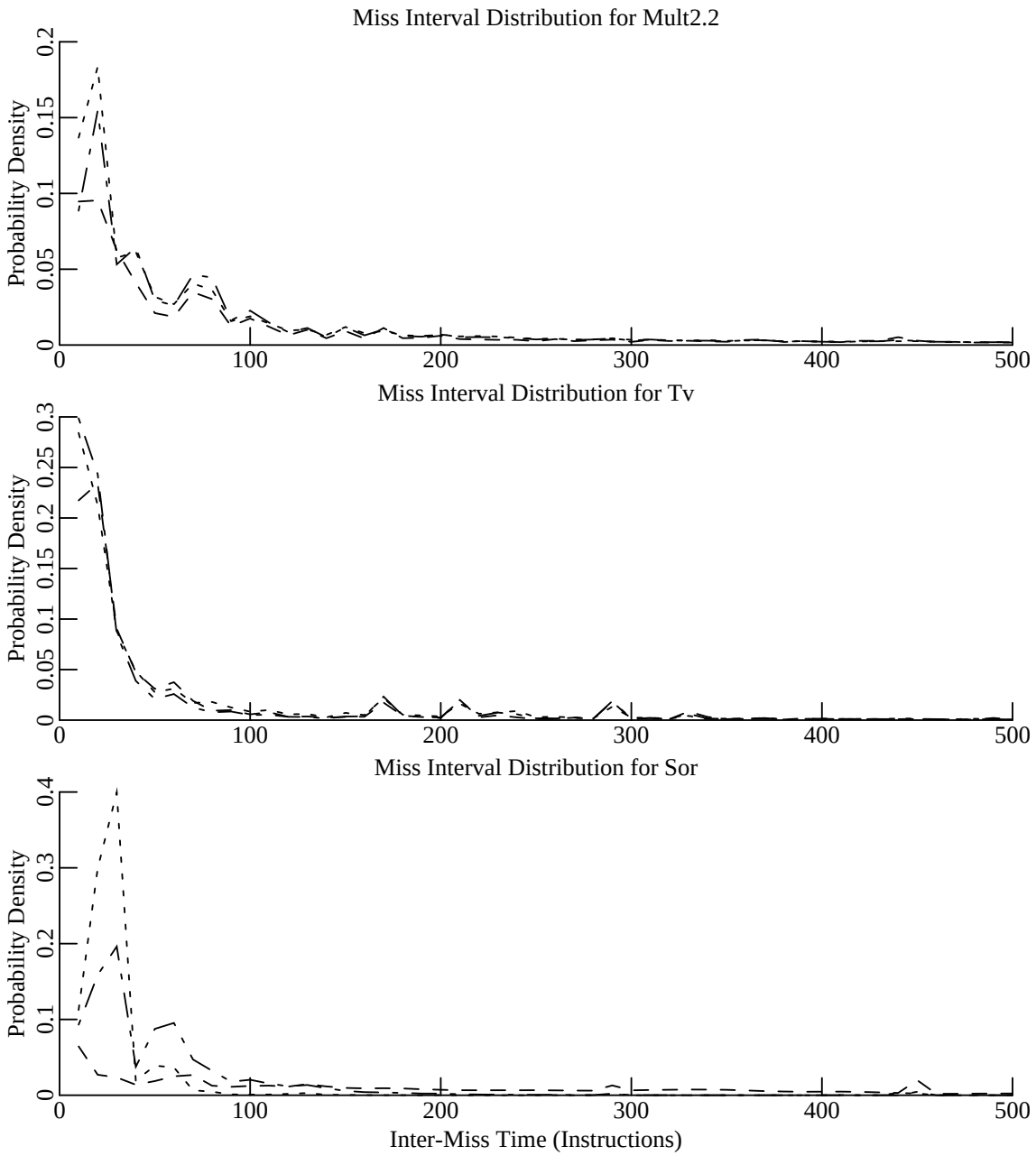


Figure 3.10b. Inter-Miss Density of Mult2.2, Tv, and Sor.

This figure shows the inter-miss probability density of the Mult2.2 (top), Tv (middle), and Sor (bottom) traces for 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes, like Figure 3.10a. Note that the scales change.

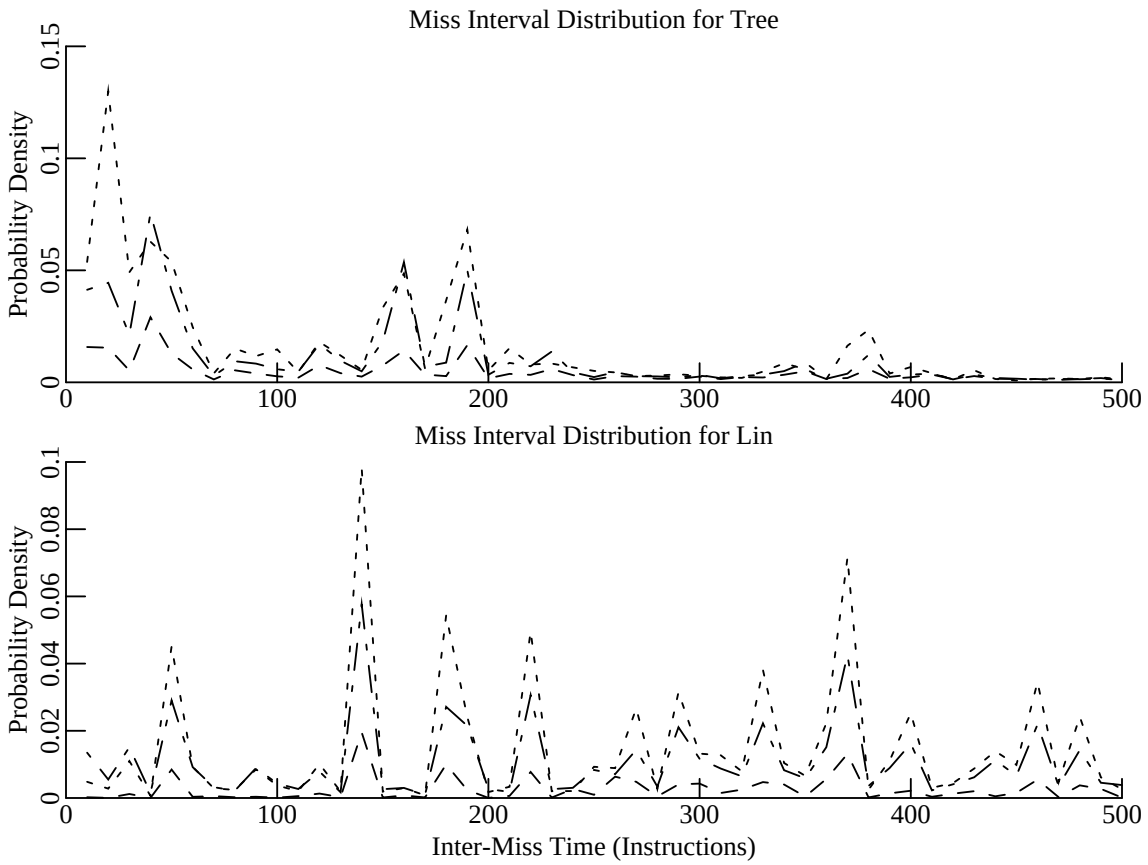


Figure 3.10c. Inter-Miss Density of Tree and Lin.

This figure shows the inter-miss probability density of the Tree (top) and Lin (bottom) traces for 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes, like Figure 3.10a. Note that the scales change.

short inter-miss intervals at the beginning of process execution. The uniprogrammed traces, in particular Tree and Lin, do not show as much of a bias toward short inter-miss intervals. The misses of Lin are much more evenly distributed among the different intervals. The most likely reason is the low cache contention for Lin; the application behavior dictates the inter-miss interval, not cache contention.

The consequence of short inter-miss intervals is that the main memory access frequency will be bursty. This may worsen queueing delays for the main memory, and reduce the effectiveness of write-buffers.

3.5. The Advantages of Long Traces

This section examines two advantages of multi-billion-instruction long traces. The first advantage is that they can more fully capture the cache behavior of a traced workload and relate cache behavior to algorithmic phases of the workloads. More trace data allows far more phases of execution to be accounted for. The second advantage is that long traces can mitigate the cold-start problem. With many misses over long periods of time, cache initialization is only a small factor in the *MPI* estimate.

3.5.1. Capturing More Workload Behavior

If a long trace can capture more of the execution of a workload, cache behavior for the workload can be more fully characterized. The result of the phase behavior of programs is that *MPI* over time may best be described as non-stationary, even when the behavior is aggregated over execution intervals of many millions of instructions. It is difficult to predict cache behavior over the entire execution of a non-stationary workload without characterization of all (or most) of the program phases.

Figures 3.11a, 3.11b, and 3.11c show the miss behavior of all the traces for secondary cache sizes of 1-megabyte, 4-megabytes, and 16-megabytes. They also show the compulsory *MPI*. Compulsory misses are caused by the first reference in a trace to a unique memory location (128-byte block). The compulsory *MPI* is a measure of the minimum cache *MPI* when cache blocks are demand-loaded. It is exactly the infinite cache miss ratio.

The behavior of the uniprogrammed traces can be related to the algorithms of the traced program [BOKW90]. The scientific Sor trace may best exemplify this. Its array traversals cause periodic jumps in the *MPI*. The Tv program spends a long time building a large intermeshed data structure that it then traverses near the end of its execution. The flat beginning of the Tv *MPI* curve shows the building of the data structure, while spikes near the end show its traversal. The behavior of the multiprogrammed traces is much more difficult to relate to the underlying algorithms in the workload since all the different processes concurrently contribute to the *MPI*.

Cache performance can change dramatically over the length of the traces, although each point given in the graph is the aggregate of 100 million instructions. A difference of a factor of ten in the cache *MPI* may easily occur over the length of the traces. The actual range varies widely from trace to trace. For example, the maximum *MPI* is 4.5 times the minimum *MPI* for the Mult1.2 4-megabyte cache, while the maximum is 154 times the minimum for the same Tv cache. Any trace sample of 100 million instructions or less could estimate the *MPI* anywhere between the the maximum and minimum, so it is important to have trace data that includes many hundreds of millions, or billions, of instructions.

Beyond the cache *MPI*, the compulsory (or infinite cache) *MPI* provides much information on the traced workloads for multi-megabyte caches. This information would be difficult to obtain without long traces. A large portion of the misses caused by the multiprogrammed traces are compulsory, particularly with the 16-megabyte caches, even after the simulation of billions of instructions. The frequent birth and death of new processes in the multiprogrammed workload causes this; both the series of compiles and the simple UNIX commands contribute to the compulsory misses throughout the lifetime of the trace. The Tree and Sor workloads eliminate compulsory misses by the end of the trace because their memory usage stabilizes. Tv allocates new memory throughout its lifetime, even as execution completes.

Figure 3.12 shows the fraction of all misses that were compulsory over the length of the trace. 50% of the misses in the 16-megabyte direct-mapped cache were compulsory with the multiprogrammed traces, a large portion. This shows that with multi-megabyte caches, many misses may be compulsory since the caches have such low *MPI*'s. Short traces would likely estimate the compulsory *MPI* even higher than these values because much trace context is needed to determine which references are compulsory; long traces are extremely useful for this kind of analysis.

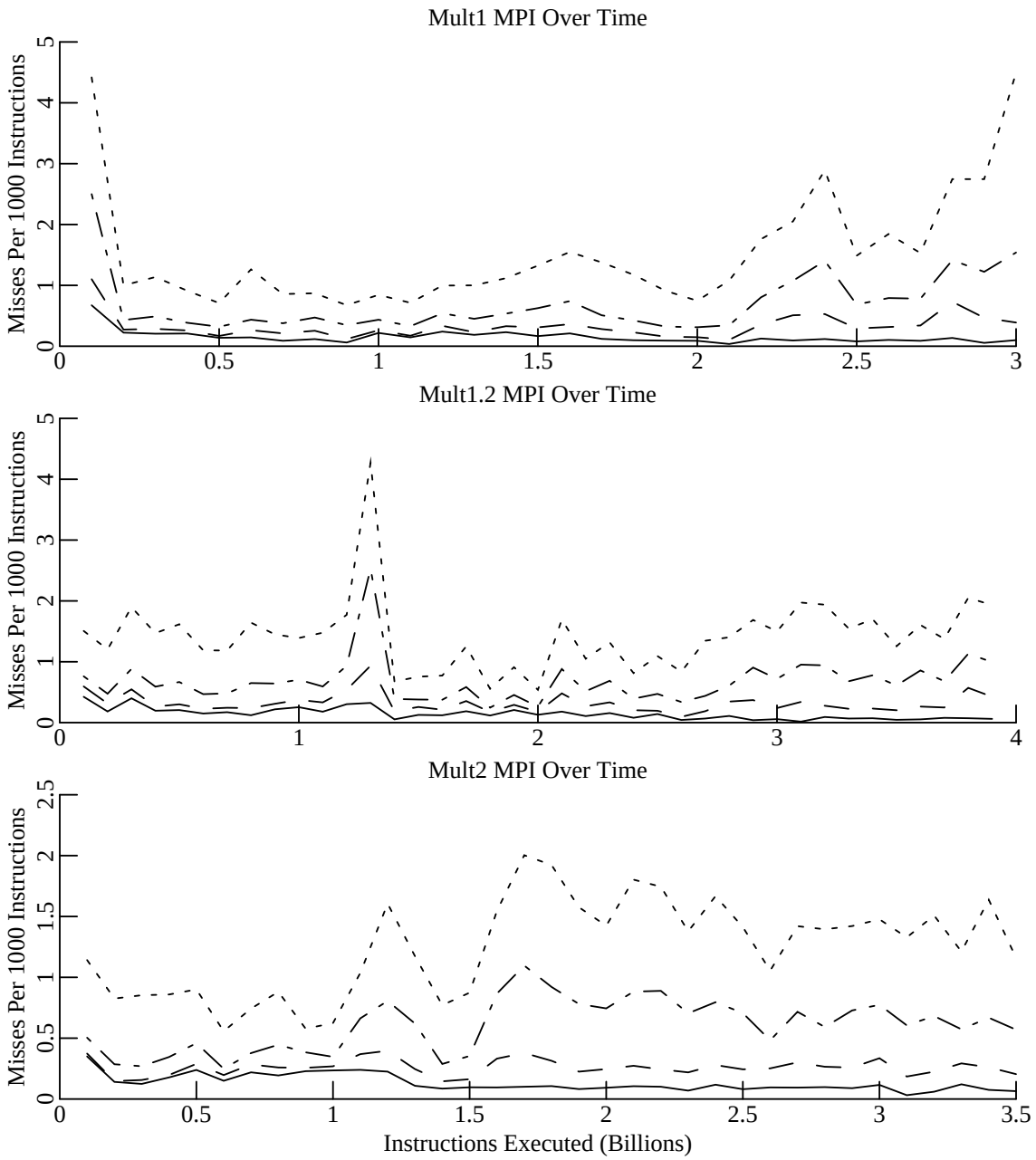


Figure 3.11a. Variability of Mult1, Mult1.2, and Mult2.

This figure shows the *MPI* behavior of the Mult1 (top), Mult1.2 (middle), and Mult2 (bottom) traces for an infinite (solid line) cache, 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes. Each point is the average value for the previous 100 million instructions. Note that the scales change.

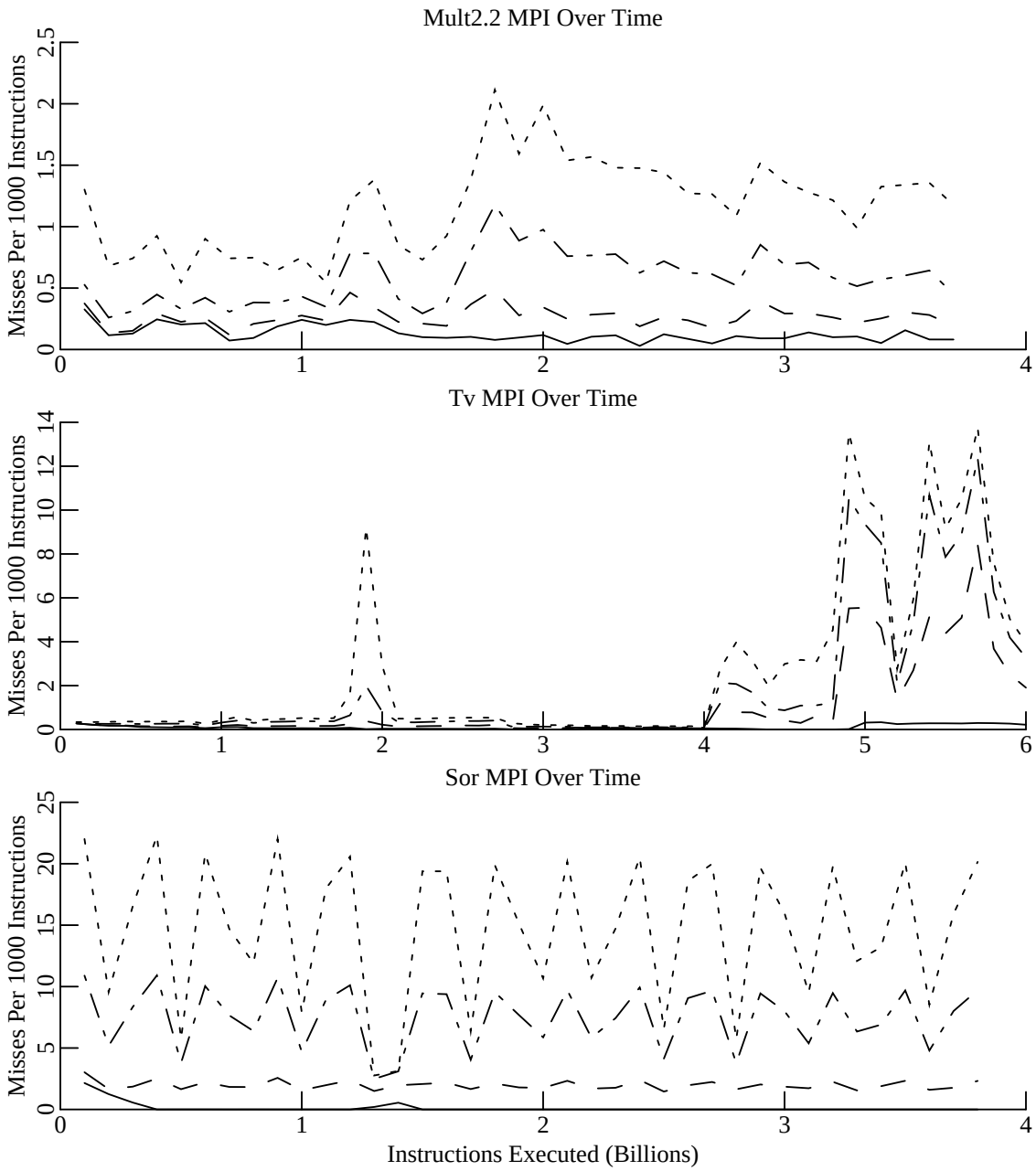


Figure 3.11b. Variability of Mult2.2, Tv, and Sor.

This figure shows the *MPI* behavior of the Mult2.2 (top), Tv (middle), and Sor (bottom) traces for an infinite (solid line) cache, 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes. Each point is the average for the previous 100 million instructions. Note that the scales change.

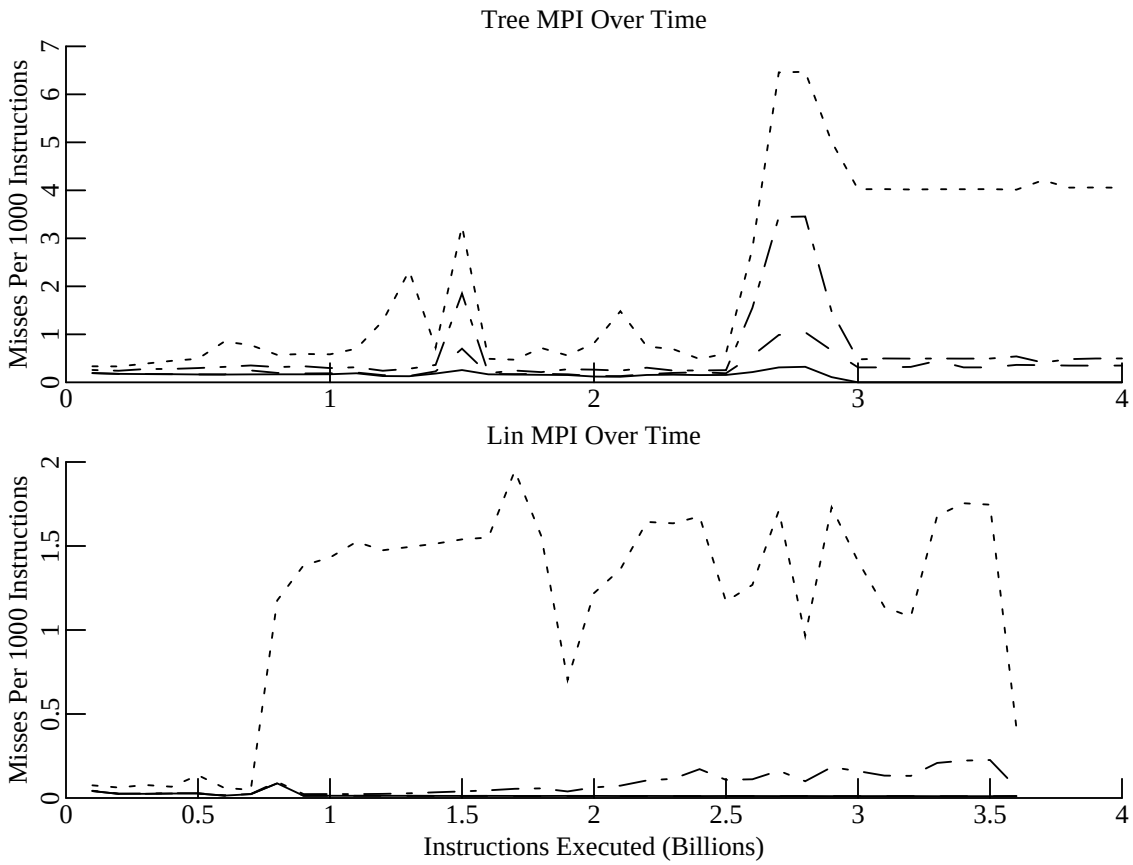


Figure 3.11c. Variability of Tree and Lin.

This figure shows the *MPI* behavior of the Tree (top) and Lin (bottom) traces for an infinite (solid line) cache, 16-megabyte (dashed), 4-megabyte (dot-dashed), and 1-megabyte (dotted line) direct-mapped secondary caches with block sizes of 128-bytes. Each plotted point is the average value for the previous 100 million instructions. Note that the scales change.

3.5.2. Overcoming Cache Initialization

Cold start is a big problem with large caches. This section shows that very long traces are needed to overcome multi-megabyte cold-start by showing that many instructions are required to meet several previous definitions of when caches are warm.

Cache initialization is the process of filling an empty cache, turning it from cold to warm. Easton and Fagin [EASF78] define the warm start miss ratio to be the miss ratio with a full cache. The cache must be completely filled before the cache becomes ‘‘warm’’; every cache block frame must be referenced at least once. Table 3.9 shows the number of instructions required to fill the primary caches completely from empty.

Table 3.9 shows that a trace of millions of instructions may be required to initialize the data caches *fully*, and that the instruction cache is often not even initialized after billions of instructions from the uniprogrammed traces. The multiprogrammed workloads require ten million instructions or more to initialize the instruction cache entirely. Millions of instructions may be required to initialize the data cache. The restrictive direct-mapped cache placement causes these large initialization lengths because it takes a considerable amount of time to reference every cache block frame. Fully associative caches,

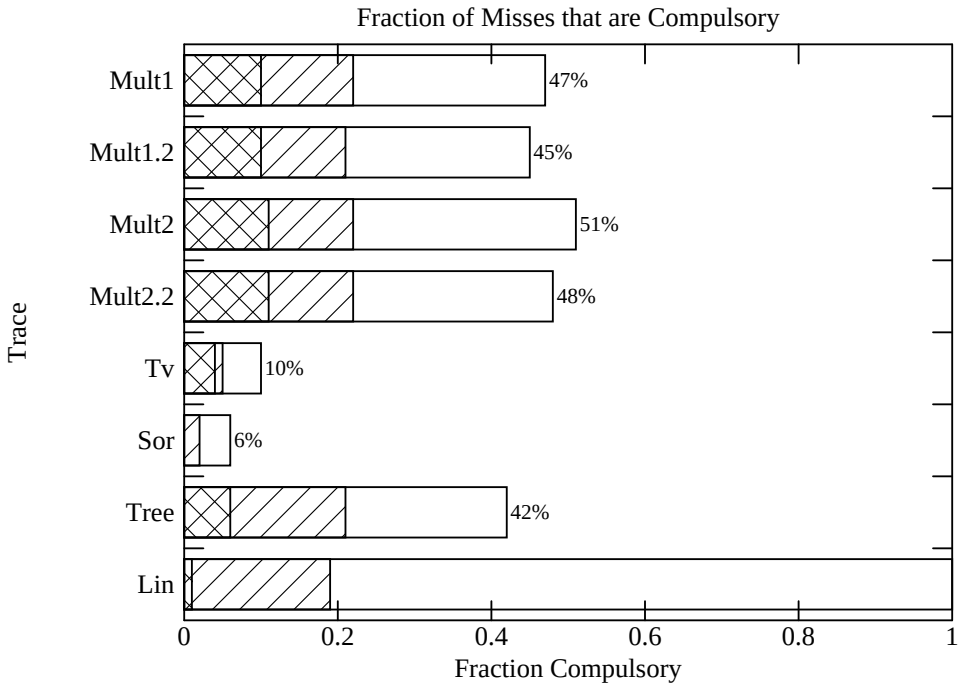


Figure 3.12. Fraction of Misses that are Compulsory.

This table shows the fraction of misses that were compulsory in 1-megabyte, 4-megabyte, and 16-megabyte direct-mapped caches. For each trace, the fraction of the misses of a 16-megabyte cache is the largest fraction (un-hatched box), with the smaller 4-megabyte cache fraction (half-hatched) and the smallest 1-megabyte cache fraction (cross-hatched).

Instructions to Warm Cache (Millions)		
Trace	Primary Caches	
	Instruction	Data
Mult1	85.8	3.5
Mult1.2	9.7	1.8
Mult2	36.4	0.5
Mult2.2	48.8	3.6
Tv	5581992.7	12.3
Sor	∞	0.2
Tree	∞	1.5
Lin	∞	13.5

Table 3.9. Instructions to Fill Primary Caches.

This table shows, for each trace, the number of instructions required to reference every cache block frame in the split primary caches. A value of infinity denotes that the cache was not filled over the entire length of the trace.

the type studied by Easton and Fagin, do not require as many instructions to initialize the cache fully since there is much more (re)placement flexibility. The results for the primary caches are not encouraging since the major focus of this study is on multi-megabyte secondary caches. These considerably larger caches can require even longer initialization times since they have many more cache blocks.

Table 3.10 shows the instructions needed to fill different direct-mapped secondary caches entirely. These results show that billions of instructions may be required to fill these multi-megabyte caches fully, and some traces could not properly initialize the largest caches, even with several billions of instructions. Sor required considerably less time to fill the cache because its frequent traversals of large amounts of memory touched the cache block frames more rapidly. The multiprogrammed traces could not reference every cache block frame in the 16-megabyte cache, although much more than 16-megabytes of memory was referenced over the billions of instructions in the traces. The instructions required to initialize the 4-megabyte cache fully ranges from 21 million instructions for the Sor trace to 2.55 billion for Lin, and the multiprogrammed traces required nearly a billion instructions. Again, the direct-mapping was a factor in the long initialization times of these large caches.

Instructions to Warm (Millions)			
Trace	Secondary Cache Size (Megabytes)		
	1	4	16
Mult1	45	964	∞
Mult1.2	93	1001	∞
Mult2	127	670	∞
Mult2.2	118	581	∞
Tv	213	1028	4956
Sor	4	21	185
Tree	48	196	798
Lin	552	2548	∞

Table 3.10. Instructions to Fill Direct-Mapped Secondary Caches.

This table shows the number of instructions required to reference every cache block frame in direct-mapped secondary caches.

Table 3.11 shows the initialization times required for 4-way set-associative secondary caches of the same size. The comparison with the direct-mapped caches shows that the flexibility of higher associativity significantly reduced the cache initialization time. The multiprogrammed traces were able to initialize the 16-megabyte cache fully here, but it still took billions of instructions. The required trace length for the multiprogrammed traces was approximately halved by changing the caches to 4-way set-associative from direct-mapped. Ten millions of instructions were required to initialize the 1-megabyte cache fully, and hundred millions of instructions were needed for the 4-megabyte cache. The Lin trace did not initialize the 16-megabyte cache because it did not reference 16-megabytes over the length of the trace.

Though the Easton and Fagin definition of a warm-start miss ratio is a safe one, the large amount of trace data that is required to initialize a multi-megabyte cache fully, especially with limited associativity, can make the definition too costly. The use of this definition to analyze the multi-megabyte caches considered in this work would lead to discarding much of the information contained in the traces. Even the information contained in reference strings that never entirely fill the cache is useful and should preferably not be wasted only on cache initialization.

These motivations prompted a new definition of a warm cache by Agarwal, et al., [AGHH88]. They define a cache to be warm either when the cache saturates as it becomes full, or when the trace saturates. A trace saturates when the original working set is loaded into the cache. They suggest that a uniform way to decide when a simulated cache meets this definition is to detect a knee in the graph of

Instructions to Warm (Millions)			
Trace	Secondary Cache Size (Megabytes)		
	1	4	16
Mult1	24	267	1820
Mult1.2	50	242	1604
Mult2	43	407	2826
Mult2.2	47	397	1860
Tv	115	580	3420
Sor	1	8	93
Tree	48	196	798
Lin	507	2534	∞

Table 3.11. Instructions to Fill 4-way Secondary Caches.

Shown are the traced instructions required to fill 4-way set-associative secondary caches.

the cumulative initialization references over the length of the trace, where an initialization reference is the first reference to a cache block frame during a cold-start simulation. The knee indicates a slowdown in the rate that cache block frames are being filled. Both trace saturation and cache saturation can cause the knee: cache saturation because the cache is already full and trace saturation since few new blocks are being loaded.

Figure 3.13 plots the cumulative initialization references for 1-megabyte, 4-megabyte, and 16-megabyte caches over the first 300 million instructions of the Mult1.2 trace. There are no obvious signs of trace saturation since the curves increase smoothly. Cache saturation seems the dominant factor in deciding the threshold, but one might get an altered perspective by changing the scaling of the axis. As an arbitrary choice, the cache was considered warm about when 80% of the cache blocks were filled. From the data in Figure 3.13 (and other data), 20 million instructions warmed the 1-megabyte cache, while the 4-megabyte and 16-megabyte caches required 120 and 700 million instructions.

For all the traces, Table 3.12 shows the trace lengths required to warm a cache using the warm definition of Agarwal, et al., (and the 80% rule). All the traces were able to meet the new definition for all cache sizes. In particular, Lin and the multiprogrammed traces were able to warm the 16-megabyte cache since the cache saturated. Comparison with the results in Table 3.10 shows that considerably shorter traces meet the conditions of the new definition. For instance, it reduces the trace length required by Tv by more than a factor of five for all cache sizes.

Though the definition of Agarwal, et al. decreases the loss of information required to warm a cache, a billion instructions may still be needed just to warm up the larger caches used in this study. The warm time is so large is because it takes many instructions to reference large amounts of memory. Locality of reference implies that small areas of memory will be referenced over short periods of time, while only over long periods will large amounts of memory be referenced. Figure 3.14 shows the memory referenced by each trace over time. The workloads show varying degrees of locality. Sor references most of the memory it uses quickly. This shows its poor locality. The memory referenced by the multiprogrammed traces increases almost linearly over the entire range of instructions, perhaps because a substantial portion of misses is a result of the continuous startup and completion of smaller processes. Billions of instructions from these memory-intensive traces may be required to reference 16-megabytes of memory. Clearly, billions of instructions will be needed to warm a 16-megabyte cache.

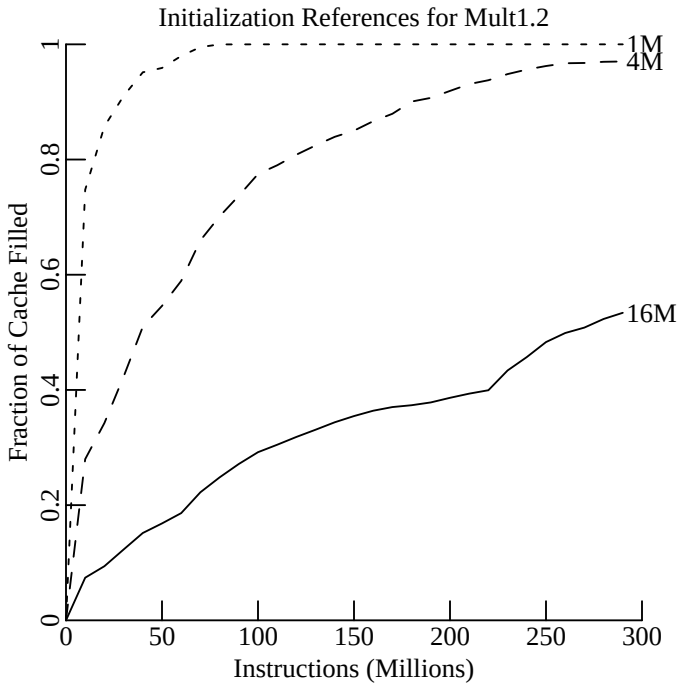


Figure 3.13. Cumulative Initialization References for Beginning of Mult1.2.

This figure plots the cold misses for the beginning of the Mult1.2 trace, expressed as a fraction of the total cache size, for 1-megabyte, 4-megabyte, and 16-megabyte direct-mapped secondary caches with block sizes of 128-bytes.

Instructions to Warm (Millions)			
Trace	Secondary Cache Size (Megabytes)		
	1	4	16
Mult1	20	40	500
Mult1.2	20	120	700
Mult2	20	80	700
Mult2.2	40	130	1000
Tv	50	190	1000
Sor	10	10	90
Tree	40	160	600
Lin	230	800	1000

Table 3.12. Instructions to Warm Cache to Knee.

This table shows the number of instructions required to warm 1, 4, and 16-megabyte direct-mapped caches with 128-byte blocks using the definition of Agarwal, et al. [AGHH88] from each trace.

Cache warming is not required to remove the cold-start bias. Alternatively, bounds on cache performance can be determined, and the cache can be *considered* warm when the bounds are sufficiently small that cold start is not a problem. Stone advocates having enough misses per simulated cache block frame (MPB) so that the cache *MPI* estimate will be accurate [STON90]. The cache need not be full for a high MPB, yet it can still be considered warm. An accurate *MPI* estimate can be obtained if the MPB is sufficiently high because the cold-start (or initialization) references are a small factor in cache *MPI*

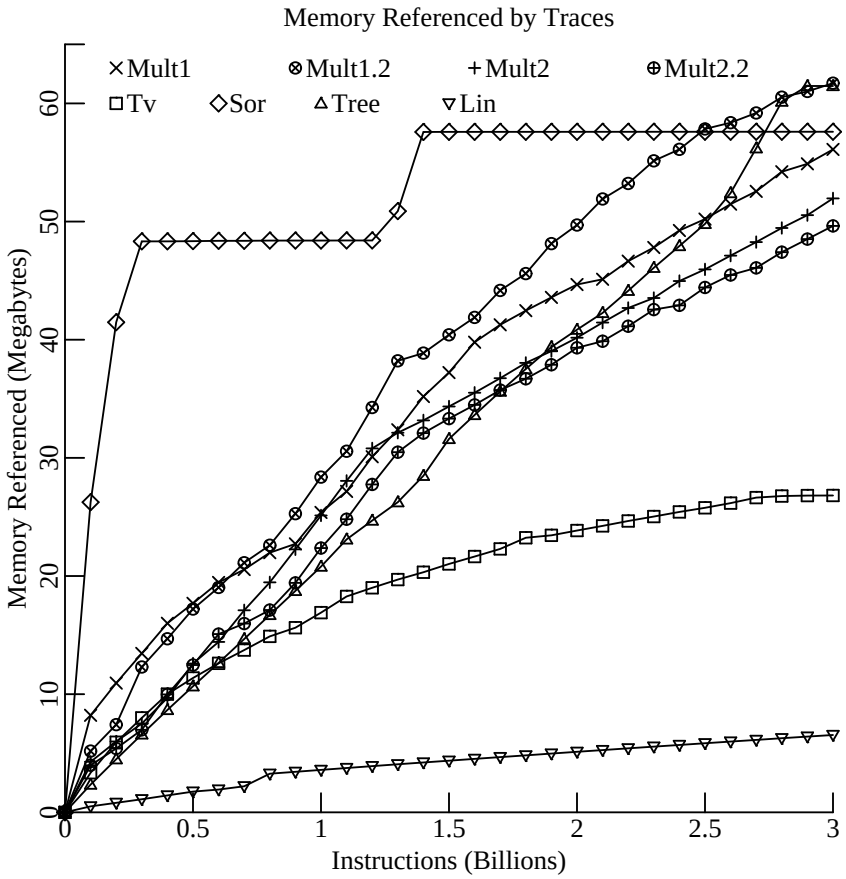


Figure 3.14. Memory Referenced Over Length of Trace.

This figure shows the memory referenced from the beginning of each trace, calculated by the number of unique 128-byte blocks referenced, over the first three billion instructions of each trace.

estimates.

An MPB of five is considered sufficient to warm the cache because the cold-start error will then be at most 12.5% of the unbiased *MPI* if half the cold-start references are guessed to be misses, and the other half are counted as hits. Table 3.13 shows the number of instructions required from each trace to produce an MPB of five (the MPB counts cold-start references). The results show that several billion instructions are needed from most traces to get an MPB of five with a 16-megabyte cache, 1 billion instructions will usually give an MPB of five for 4-megabyte caches, and 100 million instructions will usually give an MPB of five for 1-megabyte caches. The Lin trace was unable to obtain an MPB of five for the larger caches since it referenced smaller amounts of memory over the length of the trace.

Note that the trace length required for an MPB of five increases faster than the inverse of the *MPI* increases with cache size. This is because there are more block frames as the cache size increases. If the *MPI* decreases by 30% with each cache size doubling, the trace length must increase by a factor of eight when the cache size quadruples to obtain the same MPB value. (Equivalently, the trace length should increase in proportion to the cache size raised to the power 1.5 [STON90]. The next chapter also verifies this required trace length increase.) This leads to long required trace lengths, even traces containing of billions of instructions, particularly for cache sizes in the multi-megabyte range.

Instructions (Millions) for an MPB of 5			
Trace	Direct-Mapped Cache		
	1M	4M	16M
Mult1	20	50	2280
Mult1.2	30	250	1860
Mult2	30	460	2520
Mult2.2	50	440	2430
Tv	130	630	4090
Sor	10	20	310
Tree	130	590	2710
Lin	500	2800	∞

Instructions (Millions) for an MPB of 5			
Trace	2-way Cache Size		
	1M	4M	16M
Mult1	30	70	2690
Mult1.2	40	290	2260
Mult2	50	590	2640
Mult2.2	50	570	2870
Tv	130	650	4180
Sor	10	20	320
Tree	190	790	2910
Lin	800	3300	∞

Instructions (Millions) for an MPB of 5			
Trace	4-way Cache Size		
	1M	4M	16M
Mult1	30	70	2720
Mult1.2	40	330	2310
Mult2	50	630	2780
Mult2.2	50	590	2910
Tv	130	660	4190
Sor	10	20	300
Tree	200	790	2980
Lin	800	∞	∞

Table 3.13. Trace Length for Five Misses Per Block Frame.

This figure shows the number of instructions required (in Millions) to produce five misses per simulated cache block frame for direct-mapped (top), 2-way set-associative (middle), and 4-way set-associative (bottom) secondary caches with block sizes of 128-bytes. Results are shown for 1-megabyte, 4-megabyte, and 16-megabyte caches.

3.6. Conclusions

The first portion of this chapter describes the mechanism (implemented at DEC WRL [BOKL89, BOKW90]) used to gather the long traces used in this dissertation. Code modification of programs at compile time allows the efficient tracing of the memory reference behavior of programs while they are executing, slowing execution time by only a factor of ten while increasing the code size by only a factor of 2.1. The code of traced programs writes entries in a trace buffer so that a cache simulator can reconstruct all the memory references by analyzing the trace data. This trace gathering mechanism collects proper multiprocess memory reference interleavings since all processes write into a common trace buffer. It also overcomes trace length limitations using operating system modifications that

allowed many smaller traces to be concatenated into a single long trace with little distortion in the fluidity of the trace; the trace data appeared as a single long trace.

This chapter also describes the techniques used to compress and store the trace data. Techniques similar to those introduced by Samples [SAMP89] helped store the traces in a small amount of space. Each memory reference in the traces required only a few bits of storage.

A suite of workloads that are large consumers of memory (by the standards of 1990) were traced since future workloads will likely be larger, and, since large workloads can exercise multi-megabyte caches. The traced programs were combined into both uniprogrammed and multiprogrammed workloads that use memories up to 100-megabytes. The workloads are a prediction of the workloads of future engineering workstations. They included CAD (C), scientific (Fortran), Scheme (LISP dialect), and Unix utility programs. A large portion of the references within the multiprogrammed traces come from large processes that run for hundreds of millions of instructions. This chapter analyzes the traces in detail. It gives miss frequencies for a variety of large secondary caches and the primary caches. Doubling the cache size decreases the *MPI* by a larger amount than doubling the associativity of the cache.

The traced workloads have widely varying memory access characteristics, even across 100 million instructions or more. Long traces capture the variability in the various phases of application execution to more properly characterize their memory reference behavior. Long traces can be used to understand program behavior by relating cache performance to changing execution phases. Compulsory misses are a significant factor in the performance of the large caches in this study. Long traces capture more workload behavior and allow cache performance to be more adequately characterized.

Long traces overcome the large cold-start effect that occurs when simulating large caches with short traces. Even with several different definitions of a warm cache, billions of instructions may be needed to initialize the larger caches considered in this study. The next chapter examines techniques to minimize the cold-start bias in short traces (time-samples). It shows that traces of billions of instructions are not required for accurate mean cache *MPI* prediction. Still, long traces are extremely useful so that multi-megabyte cache initialization is only a small portion of a cache simulation. Long traces are most flexible since they allow many different cache performance metrics to be gathered without sophisticated cold-start reduction techniques.

Chapter 4

Trace-Sampling Techniques

4.1. Introduction

The previous chapter showed that long traces (including the memory references of many instructions) are desirable for multi-megabyte cache performance analysis. Long traces capture many phases of a workload's execution, each of which may have widely varying cache performance. Cache performance is more accurately characterized with memory references from the execution of more algorithmic phases over a longer time. Long traces also mitigate the effects of cache initialization, or cold start [EASF78], even in multi-megabyte caches. Long traces would always be used if it weren't for the difficulty in obtaining them, their large storage requirements, and the long simulation times required to use them.

This chapter is the first comparison of different trace-sampling techniques. Trace-sampling techniques can accurately estimate mean cache performance using fewer resources than with long (full) traces. Trace sampling greatly reduces simulation (and storage) requirements because it uses only a small fraction of the full trace references. With a fixed tracing budget, trace samples can give a *better* cache performance estimate than long traces because the samples can capture cache performance over a longer time frame. This chapter focuses on the accuracy and resource savings of trace sampling by comparing the simulation results from full-trace samples to the full-trace simulation results. Figure 4.1 illustrates a full trace in a time-space diagram, where every memory access is explicitly shown to reference a cache set. This chapter compares the two different trace-sampling techniques that sample the full trace by slicing this diagram. Set samples are horizontal slices in the time-space diagram, while time samples are vertical slices. Each sample includes only a portion of the full trace references.

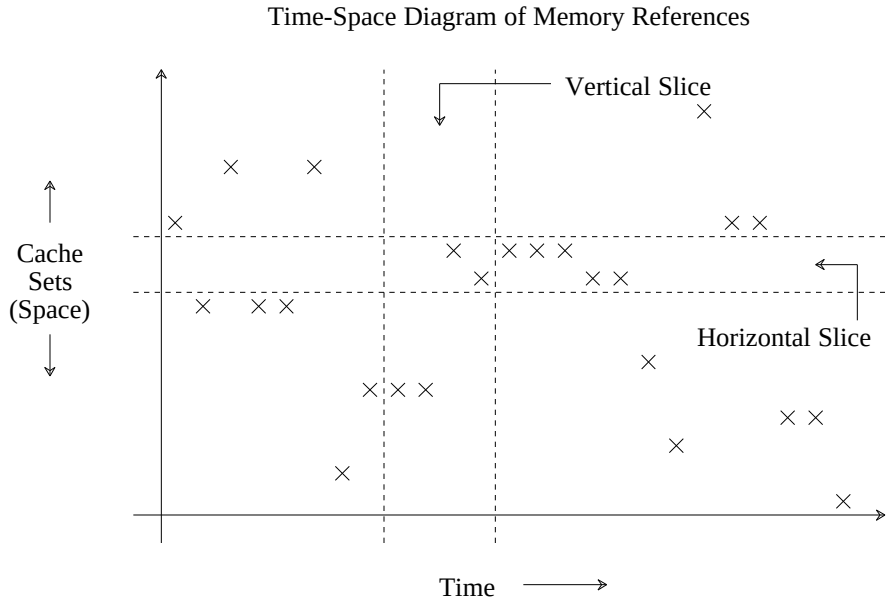


Figure 4.1. Sampling as Vertical and Horizontal Time-Space Slices.

This figure shows a time-space diagram pictorially representing the memory references within a trace during a simulation. Each memory access occurs at a certain time and references a particular cache set. The vertical and horizontal slices represent time samples and set samples of the full trace, respectively. Real traces are much longer than the trace depicted.

Puzak introduced *set sampling* (congruence class sampling) in his thesis [PUZA85]. Set sampling estimates cache performance by simulating only a portion of the cache sets over the entire trace. A set sample is a probe into the cache that captures the behavior of some cache sets. If all sets are statistically identical, then a sample of only a few sets for a long enough period will give accurate cache performance estimates [HEIS90].

Laha, et al., advocated *time sampling* to reduce trace data requirements [LAPI88, LAHA88]. Time sampling selects time-contiguous references from the full trace. The idea of time sampling is to simulate short intervals of the full trace. A time sample is a probe into a trace at a random point that captures cache behavior over a short period. Since traces exhibit widely varying memory reference characteristics over long execution periods, different time samples may give entirely different cache behavior, as evident from the results presented in Chapter 3. Laha, et al., have previously shown that 35 time-samples can adequately characterize cache behavior.

This chapter compares both the usage flexibility and accuracy of set samples and time samples. The usage flexibility of a sample is the range of cache configurations where it can be used. Usage flexibility is essential because it may be difficult or expensive (perhaps even impossible) to resample, and it may determine when samples can obviate full traces. Ideally, any performance metric of any cache could be measured with a single fixed set of samples. In practice, both set samples and time samples are only useful for a limited range of cache configurations. This chapter shows how to create usage-flexible set-samples, and how to detect when time samples should or should not be used.

This chapter compares the mean multi-megabyte cache misses per instruction (*MPI*) predictions of set sampling and time sampling using the traces described in Chapter 3. Two conditions are useful for sampling accuracy: (1) individual samples should give unbiased estimates, and (2) the sample size

should be large enough to minimize sampling error. For (2), rather than counting sets or individual time-samples, the fraction of trace data required for an acceptable sampling error is measured. The resource savings of set sampling and time sampling is then directly compared.

For unbiased set sampling, sets must be properly probed: all the full trace references to each sampled set, and no others, should be contained in the sample. This chapter introduces usage-flexible constant-bits samples that are (unbiased) set samples for multi-level caches whose set-indexing bits contain the constant bits.

The cold-start problem can bias short time sample results because the cache state is unknown at the start of a time sample [EASF78]. This chapter compares several techniques to reduce the cold-start bias: Laha, et al., lessen cold start by initializing individual cache sets rather than initializing the entire cache, and Stone does the same for direct-mapped caches [STON90]; Agarwal, et al., stitch together a longer trace out of shorter time samples [AGHH88]; and Wood, et al., predict the initialization (unknown) reference miss ratio using a renewal-theoretic model [WOHK91]. For thirty time-samples from the traces used in this dissertation, this chapter shows that the Wood, et al., technique minimizes the cold-start bias. Though the bias is greatly reduced, time samples of ten million instructions or more may still be required to overcome it.

The 10% sampling goal of this chapter is: errors of less than 10% (with 90% confidence) using less than 10% of the full trace data. Set sampling meets the 10% goal, but time sampling may not because hundreds of extremely large time-samples may be required for the desired 10% accuracy. Time sampling may only be more useful when the timing of references and the interaction among sets is important. Given that sample usage restrictions can be tolerated, set sampling gives more accurate multi-megabyte cache *MPI* estimates using less of the full traces.

Section 4.2 introduces constant-bits set-samples, shows the set-sampling errors for different fractions of the full trace data, and develops techniques to establish the level of confidence in set-sampling results. Section 4.3 compares the cold-start reduction techniques, states sufficient conditions to remove the time-sample cold-start bias, shows the time-sampling accuracy for different fractions of the full trace data, and develops techniques to establish confidence in time-sampling results. Finally, Section 4.4 summarizes the results of this chapter.

4.2. Set Sampling

Set sampling is the first trace-sampling technique examined in this chapter. This section discusses key considerations in producing and using set samples: (1) how to construct usage-flexible set-samples, and how to get unbiased and accurate *MPI* estimates from them; (2) what fraction of the trace data (equivalently, what fraction of the sets) is needed for a desired accuracy, and how can the sampling accuracy be estimated using only the sampled data?

4.2.1. Obtaining Unbiased Estimators from Set Samples

4.2.1.1. Constructing the Set Sample

A key to obtaining accurate estimates from a set sample is properly constructing the sample, or choosing the references that should be included in or excluded from the sample. When sampled, a simulated set should see (or receive) precisely the same references as in the full trace simulation. This can be guaranteed if the sample contains either all or none of the full trace references to each sampled

set[11].

Definition 4.1. Set Sample.

A *set sample* contains exactly the subset of accesses from a full trace that reference a portion of the sets, the sampled sets. The accesses also must occur in the same order in the sample as in the full trace.

A sample of the full trace references may be a set sample for one cache, while it is not a set sample for another cache. The sample is *not* a set sample if it includes at least one, but not all, of the references to a cache set. For example, a cache with a larger block size will have fewer sets; references to a single set may only be a portion of the references to a set when the block size is larger.

In effect, there are restrictions on the use of an individual set sample. To permit analysis of many caches with different block sizes, associativities, and number of sets, a sample must meet a wide range of restrictions. Puzak randomly selected sets [PUZA85], but this method is inflexible and inadequate for multi-level cache simulations since it assumes fixed set-indexing. Primary cache set-indexing is different from the secondary cache set-indexing, so a random selection of secondary cache sets is probably not a random selection of primary cache sets; for example, a sample of a single secondary cache set may not be a set sample for the primary cache because it may include only some (not all) of the references to a single primary cache set. The constant-bits sampling technique introduced in this chapter produces a single sample that includes references to many sets, and can be used with a wide range of caches, including multi-level configurations with varying primary and secondary cache set-indexing.

The simple constant-bits sample construction technique selects the portion of memory accesses that have given values for some address bits.

Definition 4.2. Constant-Bits Sample.

A *constant-bits sample* of a full trace, for bits $i \in CONSTANT$ (the constant bits), retains only those references whose addresses have given constant values in bit positions i of their address, and preserves their ordering.

For example, a constant-bits sample including about 1/4 of the memory accesses included in the full trace selects only those addresses that have zeroes in two address bits, such as bits five and six (assume bit zero is the low order bit and byte addresses). Bits five and six are then the constant bits of this sample. Theorem 4.1 proves that a constant-bits sample is a set sample exactly when the set-indexing bits of the cache contain the constant bits.

Theorem 4.1.

Consider a set-associative cache that uses simple address bit-selection[12] to choose the

11. This condition may not be sufficient if the cache organization includes prefetching or timing-dependent behavior.

12. This means the set-indexing bits come directly from the address of the memory access [SMIT82]. With other than simple bit-selection cache indexing, the scenario is more complicated. In particular, since PID-hashing is used in this study, care is taken to ensure that the hashed index bits did not overlap with the

set that a reference will access. A constant-bits sample (Definition 4.2) from an arbitrary full trace is a set sample (Definition 4.1) if and only if the set-indexing bits, *INDEX*, of the cache contain the constant bits *CONSTANT* ($CONSTANT \neq \emptyset$) (i.e. $CONSTANT \subseteq INDEX$).

Proof - The correct access orderings are clearly upheld with a constant-bits sample, so this proof ignores ordering to concentrate on whether the sample includes either all or no references to each set.

Necessary Condition -

A contradiction (contra-positive) will be shown. Suppose the indexing bits do not contain the constant bits, that is, there exists a bit i such that $i \in CONSTANT$ and $\{i\} \cap INDEX = \emptyset$. An address that differs only in bit i could be constructed from any address contained within the sample. This address would index to the same set in the cache as the address it is constructed from, yet it would not be contained in the sample. This implies that only some references to the given set from an arbitrary full trace may be contained within the sample, which further implies the sample is not a set sample.

Sufficient Condition -

A contradiction is again shown. Consider two references in the full trace that index to the same set, one that is from the sample and another that isn't. Two such references must exist when a sample is not a set sample. The addresses of these references cannot be the same, for if they were, they would either both be included or excluded from the sample. Let *DIFF* be the bit positions that the two addresses differ, note that $DIFF \cap INDEX = \emptyset$ since both addresses index to the same set. It also must be true that $DIFF \cap CONSTANT \neq \emptyset$ since an address can be included in the trace while the other is excluded only if they differ in the bits in *CONSTANT*. This means there must exist an i such that $i \in CONSTANT$ and $\{i\} \cap INDEX = \emptyset$, which implies that the indexing bits contain the constant bits.

Thus, the usage restrictions of a constant-bits sample can be precisely stated. For example, the sample with constant bits five and six would be a set sample for bit-select caches whose block sizes are less than or equal to 32 bytes and whose cache size divided by associativity is greater than or equal to 128 bytes.

Cache hierarchies impose the strictest restrictions on the use of samples, but constant-bits samples can tolerate them. When the indexing bits of all caches in a multi-level hierarchy contain the constant bits, a sample is a set sample for the hierarchy as a whole because the sampled sets of all caches in the hierarchy see (receive) the same references as they would with the full trace. The constant bits in this study were chosen so that the constant-bits samples are set samples for all the two-level cache configurations.

---

constant bits, so the constant-bits samples are also set-samples. Note that though this study uses virtual-indexing, the constant-bits technique is equally applicable to real-indexed caches, and it even works with hierarchical configurations including both real and virtual indexed caches if the constant bits are below the page boundary.

4.2.1.2. Estimating *MPI* With a Set Sample

Given a set sample *i*, the obvious way to estimate *MPI* is by $MPI_i = \frac{M_i}{I_i}$, where MPI_i is the misses per instruction estimate for set sample *i*, I_i is the number of instruction references included in *i*, and M_i is the number of misses in the simulation of *i*. While this calculation of the *MPI* estimate may make intuitive sense, it produces poor results. Consider several iterations of a tight instruction loop that limits the instruction accesses to a single set. The different iterations of the loop could cause misses to occur in all sets, though all instruction references are to only a single set. The *MPI* estimate from the rest of the sets would be infinite. Alternative *MPI* calculations must be considered to solve this problem.

This chapter estimates the *MPI* of a set sample *i* by $MPI_i = \frac{M_i}{f_i \times I}$, where *I* is the number of instructions in the full trace, and f_i is the sampling correction for *I* with set sample *i*. If f_i is the fraction of instruction references included in set sample *i*, $f_i(instr)$, then this equation is precisely $\frac{M_i}{I_i}$ since $f_i \times I = I_i$. Other straightforward f_i's are: the fraction of memory references (instruction and data) included in the set sample, $f_i(refs)$, the fraction of data (non-instruction) references, $f_i(data)$, or the fraction of sets included in the set sample, $f_i(sets)$. $f_i(sets)$ is the simplest of these options. No information from the trace data itself is needed to calculate $f_i(sets)$; it associates an equivalent portion of each instruction with each cache set, which is intuitive because any instruction (load or store) can cause a cache miss in any set.

For the traces considered in this dissertation, Table 4.1 shows the estimation errors relative to the full trace *MPI*, given by the coefficient of variation[13] of the MPI_i estimates using the different f_i's. $f_i(sets)$ gives MPI_i estimates with coefficients of variation errors that are an order of magnitude smaller than for the other f_i's. This simple $f_i(sets)$ is superior because the different samples have nearly the same number of cache misses, even though the number of references (or instructions) is not the same. That is, M_i is nearly the same as M_j for any given *i* and *j*. Using any other f_i based on the references included in a sample gives inaccurate results since it adds an unneeded random factor to MPI_i.

The Sor and Lin traces in Table 4.1 illustrate the inaccuracies of $f_i(instr)$. These are both scientific codes; the large errors are a result of the isolation of instruction references to a few sets, as expected from the previous example. Note that for these two traces, $f_i(data)$ is more accurate than $f_i(refs)$ because the data references are more evenly distributed across the sets than are instructions.

For all the set-sampling results given in this chapter, $f_i(sets)$ is used. Besides the references contained in the set sample, this requires knowing $f_i(sets) \times I$ for the sample, but this is only a tiny amount of added information.

---

13. This deviates slightly from the standard terminology. Normally, the coefficient of variation is the standard deviation divided by the (simple arithmetic) mean. Strictly speaking, this only holds for $f_i(sets)$. For the other f_i's, Table 4.1 gives the root-mean-squared error (relative to the full trace *MPI*) of the MPI_i's, divided by the full trace *MPI*.

Coefficient of Variation of MPI Calculations (percent)					
Trace	Full Trace $MPI \times 1000$	f_i Technique $f_i(sets)$	$f_i(instr)$	$f_i(refs)$	$f_i(data)$
Mult1	0.70	2.3%	35.2%	34.5%	87.7%
Mult1.2	0.69	1.9%	28.9%	31.7%	75.8%
Mult2	0.61	1.9%	24.2%	27.7%	64.3%
Mult2.2	0.59	1.3%	24.3%	25.6%	60.2%
Tv	1.88	0.6%	139.0%	85.7%	182.7%
Sor	7.54	0.3%	∞	339.0%	86.2%
Tree	0.59	6.8%	191.9%	148.2%	249.7%
Lin	0.09	7.6%	∞	886.3%	265.2%

Table 4.1. Errors of Different Set-Sample MPI Estimates.

This table shows the coefficient of variation (relative to the full trace *MPI*) of the MPI_i results for different f_i's. These results are for a 4-megabyte direct-mapped secondary cache and set samples of 1/16 the full trace.

The coefficient of variation is calculated as in Equation 2.3 in Section 2.5. MPI_{true} is the full trace *MPI* in that equation, and n is the number of samples that constitute the full trace (n = 16 for different set samples of 1/16 the sets).

4.2.2. What Fraction of the Full Trace is Needed?

This section examines the accuracy of the mean performance estimates obtained when set sampling with different numbers of sets, or equivalently with different fractions of the full trace. This study considers sample usage flexibility important, so only single constant-bits samples (each containing the references to many sets) are used. Since the constant-bits technique does not select randomly from the sets, but instead selects sets based on address bits, set samples of full traces that unevenly reference their address space could give inaccurate performance estimates. Any individual set sample could be a biased estimator of *MPI* because some sets are underutilized or overutilized compared to the others. This section examines the variations in performance estimates across set samples to find any effect of the non-random set selection in constant-bits set-samples.

Figure 4.2 compares the estimates obtained from 16 set samples of the Mult1.2 trace (each about 1/16 of the references of the full trace) with the actual cache performance over time. The figure shows the *MPI* of the full trace and the *MPI* estimates of the set samples for a 4-megabyte secondary cache. Note that the four constant bits are at the same address position for each sample. The 16 set samples, and their corresponding *MPI* estimates, come from the 16 different values of these four constant bits. The lines are difficult to distinguish on the graph. This shows that there is little deviation in the set-sample *MPI* estimates for different values of the constant bits. The *MPI* variations among the samples are modest when compared to the full trace *MPI* over the intervals of 100 million instructions. Similar behavior for the other traces examined in this study make set sampling promising. No correlation between the values of the constant bits and the accuracy of the set-sampling performance estimator was found. Non-random set-selection did not bias any individual set-sampling performance estimates for the Mult1.2 trace. Constant-bits set-samples gave accurate performance estimates with all the traces used in this study. Constant-bits samples were found to be equally or more accurate than random samples.

Using time-intervals of 100 million instructions, Figure 4.2 shows that set samples correctly estimate cache performance across different execution phases, as shown by The peaks and valleys in the

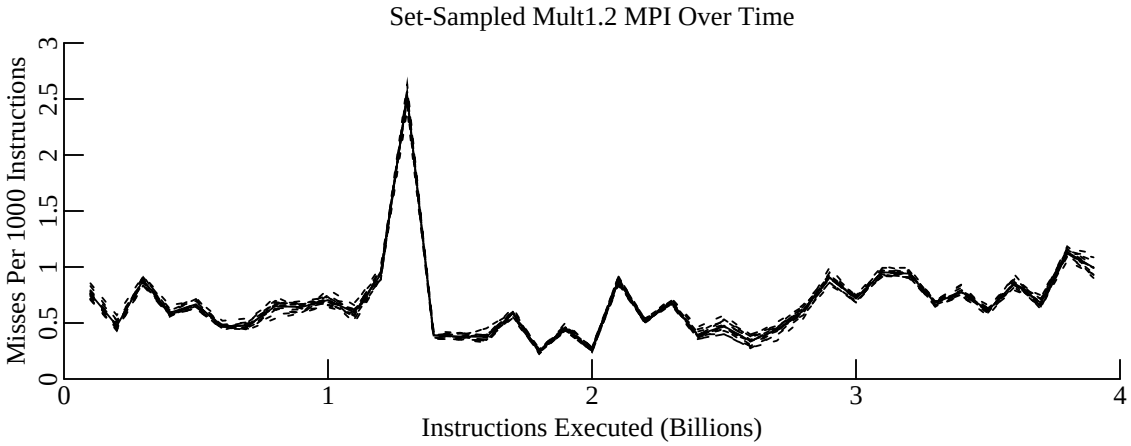


Figure 4.2. Set Sampling on the Mult1.2 Trace.

This figure shows the actual *MPI*'s (solid line) with the predicted *MPI*'s from each of 16 different set samples (dotted lines) for the Mult1.2 trace. Each set sample contains 1/16 of the trace data of the full trace. 4-megabyte direct-mapped secondary cache results are shown. Each point is the average over the previous 100 million instructions of the full trace. Bits 8-11 (bit zero is lowest-order bit) of the byte addresses are the constant bits.

MPI over the trace. The differences in *MPI* are large for different phases, but each set sample follows each peak and valley extremely closely. The inclusion of references from many phases is a major advantage of set sampling over time sampling. In contrast, a time sample contains references from only a few phases.

Figure 4.3 plots the distribution of the MPI_i estimates for the Mult1.2 trace. The figure shows the distributions for samples of 1/4, 1/16, and 1/64 of the trace data. Note that there are 4, 16, and 64 samples that produce each distribution, respectively. The performance estimates from the 1/4 samples are close to the true value of 0.69 misses per thousand instructions for the full trace, confined to only two bins about this mean in Figure 4.3. The distribution of the set samples becomes more spread out as each sample includes a smaller portion of the full trace. This is not surprising since the sample includes references to fewer sets; larger random fluctuations will inevitably occur. The 1/64 samples have a considerably more sparse distribution, only confined over a range of 15 bins, but still mostly within 10% of the mean.

Notice that the distributions shown in Figure 4.3 look normal. Since the statistics represented in this distribution are the mean for many individual sets, the central limit theorem suggests that the distributions should be normal (if the sets are independent) [MILF77]. The Kolmogorov-Smirnov one-sample test was used to compare the 1/64 sample distributions (normalized) to the standard normal distribution [DEGR75]. The quantitative results are not shown, but qualitatively the results were mixed for the different traces. Some multiprogrammed traces fit the normal curves closely, while some extreme samples resulted in a poor fit with Tree and Lin.

Table 4.2 shows coefficients of variation of the set-sample *MPI*'s for direct-mapped secondary caches. It shows that larger samples are more accurate than smaller samples; this is expected because the larger samples have more sets to even out the variations across sets. The coefficient of variation of the mean of n independent random variables (with the same mean and variance) is $\frac{1}{\sqrt{n}}$ times the

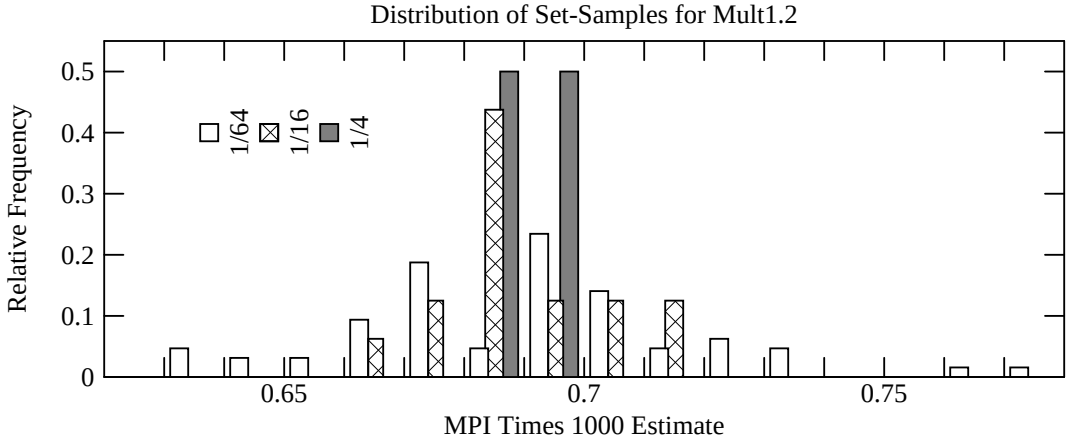


Figure 4.3. Distribution of Set Samples from the Mult1.2 Trace.

This figure shows the distribution of MPI_i performance estimates from set samples. The x-axis partitions the set samples into different bins depending on their MPI_i (at intervals of 0.01 misses per thousand instructions). The figure shows distributions for samples containing 1/4, 1/16, and 1/64 of the Mult1.2 trace. 4-megabyte direct-mapped cache results are shown. The constant bits are at bit positions 8-9, 8-11, and 7-12 for 1/4, 1/16, and 1/64 samples, respectively.

coefficient of variation of the individual random variables. Thus, if the sets are independent, the coefficient of variation should decrease as the square root of the number of sets in the sample [STON90], or it should be halved each time the sample size is quadrupled. Indeed, the results show this trend. For example with the 4-megabyte cache for Mult1.2, the two sample size quadruplings from 1/64 to 1/16 to 1/4 reduces the coefficient by 54% and 53%.

The coefficients of variation in Table 4.2 with 1/4 of the sets included in the sample are small, usually less than 2% of the actual mean; Tree has the largest one at 6% for the 4-megabyte cache. For 1/64 samples, the variations are also acceptable, often less than 5% and always less than 15% of the true mean; Tree and Lin traces have the largest ones at 14% and 15%, respectively. Generally, the errors from the set-sampling *MPI* estimates are reasonable considering the trace data reductions. If the set-sample *MPI*'s are normally-distributed[14], the the relative error of over 90% of the *MPI* samples will be less than ±1.65 times the coefficient of variation. For most of the 1/16 samples in Table 4.2, 1.65 times the coefficient of variation is well under 10%, so a set-sampling trace reduction factor of 16 gives errors less than 10% with at least 90% confidence. This suggests that set sampling meets the 10% sampling goal.

Table 4.2 shows a slight tendency for variations to decrease with larger caches. There are several reasons why this might occur. The larger caches have more sets and thus there is less competition for individual sets than with the smaller caches, so the variances between the performance of individual sets could be smaller. Furthermore, even though the samples are a fixed portion of each trace, they contain the references to more sets in the larger caches, and more averaging always reduces statistical deviations. More data needs to be gathered to find if there is a relationship between error and cache size.

14. Figure 4.3 suggests the distributions are normal. Additionally, the samples are the sum of hundreds or thousands of individual sets, so the central limit theorem [MILF77] suggests that the distributions of the samples should be close to normal if they are independent.

Set-Sampling Coefficients of Variation (percent)					
Trace	Size	Full Trace *MPI*×1000	Fraction of Sets in Sample 1/4	1/16	1/64
Mult1	1M	1.55	1.7%	4.3%	N/A
	4M	0.70	1.4%	2.3%	4.8%
	16M	0.33	1.0%	1.6%	2.7%
Mult1.2	1M	1.45	0.8%	2.9%	N/A
	4M	0.69	0.9%	1.9%	4.1%
	16M	0.32	0.4%	1.5%	3.2%
Mult2	1M	1.24	0.8%	3.4%	N/A
	4M	0.61	1.0%	1.9%	2.9%
	16M	0.26	1.1%	2.3%	3.3%
Mult2.2	1M	1.18	0.4%	2.7%	N/A
	4M	0.59	0.6%	1.3%	2.5%
	16M	0.27	0.7%	1.8%	3.4%
Tv	1M	2.63	0.7%	1.9%	N/A
	4M	1.88	0.2%	0.6%	2.1%
	16M	1.03	0.5%	0.6%	2.0%
Sor	1M	14.77	0.1%	0.4%	N/A
	4M	7.54	0.1%	0.3%	0.7%
	16M	1.97	0.0%	0.0%	0.1%
Tree	1M	2.16	4.1%	5.6%	N/A
	4M	0.59	5.3%	6.8%	13.6%
	16M	0.30	1.8%	4.1%	6.5%
Lin	1M	1.16	0.5%	3.3%	N/A
	4M	0.09	2.0%	7.6%	15.0%
	16M	0.02	0.0%	0.3%	0.5%

Table 4.2. Set Sampling Coefficients of Variation for Direct Mapped.

This table shows the actual *MPI* of the full trace for direct-mapped caches, and the coefficient of variation of the set-sampling *MPI* estimates, calculated as in Table 4.1. The samples contain 1/4, 1/16, and 1/64 of the trace data in the full trace. Some entries are marked N/A because the PID hashing overlapped with the constant bits so the samples were not set samples.

Such an error reduction could make set sampling even more useful as cache sizes continually increase.

Table 4.3 shows 2-way set-associative results like the direct-mapped results given in Table 4.2. Often, an associativity increase from direct mapped to 2-way reduces the set-sampling coefficient of variation by more than 50%. Set-sampling accuracy improves with higher associativity because set-associativity eliminates cache conflict misses [HILS89]. In direct-mapped caches, conflicts cause a substantial portion of the misses. A particularly bad conflict can cause large direct-mapped set-sampling errors because the conflicting set misrepresents the behavior of the other sets.

A large range in the distribution of the set samples is undesirable since it can lead to large variations in the cache performance estimated from an individual sample. Figure 4.4 shows the range of the set samples for 4-megabyte secondary caches to more fully understand their accuracy limitations. There is a substantially different range in the distributions for the different traces. The multiprogrammed traces exhibit similar behaviors; the range of the sample values is modest, even for 1/64 of the full trace. The Tv and Sor traces also have small ranges. However, the Tree and Lin traces have considerably larger ranges. This is closely related to the larger errors shown by Tree and Lin in Table 4.2.

Set-Sampling Coefficients of Variation (percent)					
Trace	Size	Full Trace *MPI*×1000	Fraction of Sets in Sample 1/4	1/16	1/64
Mult1	1M	1.19	1.2%	2.2%	N/A
	4M	0.55	1.0%	1.7%	3.0%
	16M	0.26	0.8%	1.6%	2.3%
Mult1.2	1M	1.18	0.7%	1.6%	N/A
	4M	0.56	0.5%	1.2%	2.2%
	16M	0.28	0.5%	1.3%	2.1%
Mult2	1M	1.01	0.3%	1.9%	N/A
	4M	0.52	0.6%	1.2%	2.0%
	16M	0.24	0.9%	1.9%	3.3%
Mult2.2	1M	0.98	0.3%	1.8%	N/A
	4M	0.51	0.5%	1.5%	1.9%
	16M	0.22	0.9%	2.1%	3.5%
Tv	1M	2.31	0.2%	0.6%	N/A
	4M	1.76	0.3%	0.3%	1.6%
	16M	0.98	0.3%	0.7%	1.9%
Sor	1M	14.66	0.0%	0.3%	N/A
	4M	7.76	0.0%	0.2%	0.5%
	16M	1.92	0.0%	0.0%	0.1%
Tree	1M	1.81	2.3%	3.7%	N/A
	4M	0.49	0.8%	1.5%	3.8%
	16M	0.26	0.3%	0.4%	1.1%
Lin	1M	1.10	0.3%	2.6%	N/A
	4M	0.06	1.2%	6.0%	9.8%
	16M	0.02	0.0%	0.3%	0.5%

Table 4.3. Set Sampling Coefficients of Variation for 2-Way.

This table shows the *MPI* of the full trace for 2-way set-associative caches, and the coefficient of variation of the *MPI* estimates, similar to Table 4.2.

The 1/64 samples from Tree and Lin were examined in more detail to find the cause of the wider performance range across the set samples in the direct-mapped case. For both the Tree and Lin traces, two of the 1/64 set-samples produced markedly higher direct-mapped *MPI*'s than the rest of the samples. These set samples are the cause of both the high maximum and the low minimum in Figure 4.4; the two samples with higher *MPI*'s biased the full trace *MPI* upward, causing the low minimum. The same two samples did not produce the highest *MPI* estimates for the 2-way set-associative caches. This shows that the cache conflicts that were eliminated by higher associativity caused the large direct-mapped Tree and Lin sample estimates, as previously suggested. The performance estimate obtained from any set sample could be biased by a heavily-conflicting set. Notice that the range of the sample estimates is much smaller for 2-way set-associativity because conflicts are eliminated.

Given an individual set-sample, it would be useful to estimate the error of its *MPI* estimate, using only the information contained within the sample. The coefficients of variation shown in Tables 4.2 and 4.3 were calculated using the full trace information. Instead, coefficients of variation can be estimated from the sample itself using the techniques from Section 2.5. Since MPI_i is the mean *MPI* from many sampled sets within sample *i*, its coefficient of variation can be estimated by measuring the variations among the sets within sample *i*. Table 4.4 compares the average estimated coefficient of variation of the MPI_i's (the ''individual'' column) with the actual coefficient of variation of the MPI_i's (the actual values come from Table 4.2) for a 4-megabyte direct-mapped cache. It also compares the average

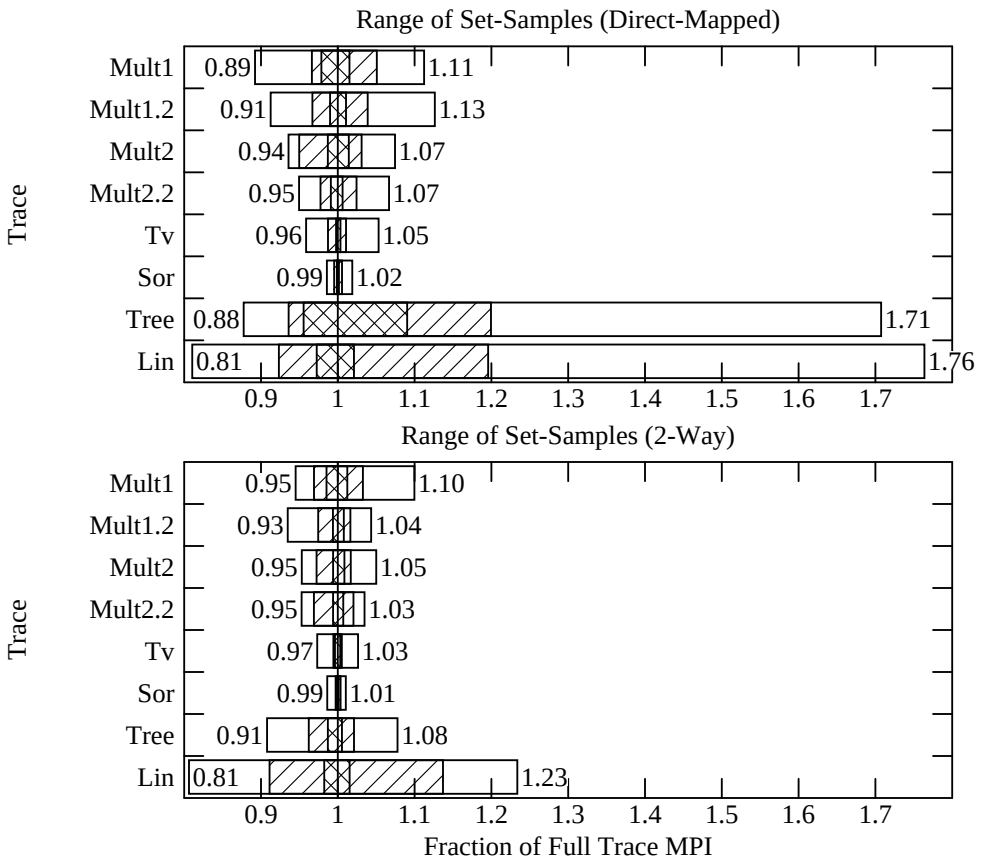


Figure 4.4. Range of Performance With Set-Samples.

This figure shows the range of the set-sample *MPI* estimates for 4-megabyte direct-mapped (top) and 2-way set-associative secondary caches with block sizes of 128 bytes. The ranges are given relative to the *MPI* of the full trace for 1/4 (smallest cross-hatched box), 1/16 (hatched), and 1/64 (largest unfilled box) sample sizes.

estimated coefficient of variation obtained with (constant-bits) batches of the sampled sets, rather than individual sets. Batching partitions (or batches) the sets within a sample, averages the sets within a batch together, and treats the batch means just as if they were individual sets[15].

The coefficient of variation results in Table 4.4 show that the individual estimate tends to be larger than both the batched estimate and the actual value. This is especially true for the Sor results. The difference in the estimates occurs because the sets within the sample (and within the batches) are not a random selection. The overlap of the Sor arrays in the virtual-indexed cache causes large contiguous chunks of the cache sets to thrash more than other chunks. Since the *MPI* differences are large across chunks, the *MPI* variance of individual sets is large. However, both constant-bits set-samples and constant-bits batches get equal pieces of each chunk, so their *MPI* estimates are a much more accurate estimate of the true *MPI* across all sets.

15. There are fewer batches than sets, but the batch means have a lower variance and a more normal distribution.

Trace	Sample Size	Coefficient of Variation			Batched 90% Confidence Interval Successes (%)
		Actual	Estimates		
			Individual	Batched	
Mult1	1/4	1.4%	1.4%	1.0%	75.0%
	1/16	2.3%	3.0%	2.9%	100.0%
	1/64	4.8%	6.0%	5.7%	93.8%
Mult1.2	1/4	0.9%	1.0%	0.8%	100.0%
	1/16	1.9%	2.2%	2.0%	100.0%
	1/64	4.1%	4.4%	4.0%	89.1%
Mult2	1/4	1.0%	0.9%	0.7%	100.0%
	1/16	1.9%	1.9%	1.9%	87.5%
	1/64	2.9%	3.8%	3.3%	93.8%
Mult2.2	1/4	0.6%	0.8%	0.5%	100.0%
	1/16	1.3%	1.7%	1.9%	100.0%
	1/64	2.5%	3.5%	3.2%	96.9%
Tv	1/4	0.2%	0.4%	0.6%	100.0%
	1/16	0.6%	0.9%	0.8%	100.0%
	1/64	2.1%	1.8%	1.4%	79.7%
Sor	1/4	0.1%	0.6%	0.1%	100.0%
	1/16	0.3%	1.4%	0.3%	100.0%
	1/64	0.7%	2.9%	0.8%	95.3%
Tree	1/4	5.3%	2.2%	2.0%	50.0%
	1/16	6.8%	4.3%	3.9%	68.8%
	1/64	13.6%	6.5%	6.2%	78.1%
Lin	1/4	2.0%	3.6%	3.0%	100.0%
	1/16	7.6%	7.2%	5.1%	87.5%
	1/64	15.0%	12.9%	12.5%	96.9%
All	1/4				90.6%
	1/16				93.0%
	1/64				90.8%

Table 4.4. Set-Sampling Error Prediction.

This table shows results for a 4-megabyte direct-mapped secondary cache. The actual coefficient of variation (taken from Table 4.2) and the average estimate for individual and batched sets are shown. Also, the table shows the probability that the batched 90% confidence interval contained the full trace *MPI*.

The mean coefficient of variation estimate from all set-samples is shown, calculated as shown in Equation 2.4 in Section 2.5. MPI_{mean} from Equation 2.4 refers to the *MPI* estimate from a particular set-sample (it is called MPI_i in this section). MPI_j from Equation 2.4 is the *MPI* estimate from a particular set (or batch) j within that set-sample. N is the number of sets (batches) in the entire cache, and n is the number of sets (batches) in that set-sample.

Using the batched coefficient of variation estimate, confidence intervals can be calculated as shown in Section 2.5. The 90% confidence interval for set sample i contains all values in the range $MPI_i(1 \pm 1.895 \times CV_i)$, where CV_i is the coefficient of variation estimate (for MPI_i), and MPI_i is the *MPI* estimate for i. Table 4.4 shows the fraction of times that this 90% confidence interval contained the full trace *MPI*. The results show that, in general, this fraction is 90% or more. The success rate is lower for Tree because conflicts distorted the set-sample *MPI* estimates. Though the intervals were inaccurate for Tree, these simple batched confidence intervals are generally excellent for establishing the level of confidence in results. Since the average of $1.895 \times CV_i$ is always less than 10% for 1/16, these results also confirm that set sampling meets the 10% sampling goal.

4.2.3. Advantages and Disadvantages of Set Sampling

The most important advantage of set sampling is that it meets the 10% sampling goal for our simulations: less than 10% errors with less than 10% of the full traces. The accuracy of set sampling may improve as the cache size increases. This makes it appropriate for multi-megabyte caches. The constant-bits technique provides a simple way to build flexible set-samples. References are either saved or discarded depending on the values of certain bits in the address of the reference. A set sample automatically includes references from many execution phases, so an individual sample can accurately characterize the *MPI* of a full trace, including its *MPI* phase-distribution. The reduced trace data requirements of set sampling allows for simulation of longer traces, and therefore more algorithmic phases, in a smaller amount of time. Besides the data reduction, set sampling also reduces the memory required to simulate a given cache configuration. A set sample containing 1/16 of the full trace needs to simulate only 1/16 of the sets. For example, the performance of a 4-megabyte cache can be estimated with a 1/16 set-sample using the same amount of memory needed for a full trace simulation of a 256 kilobyte cache.

Set sampling does have its limitations. A major disadvantage is that a set sample for one cache may not be a set sample for another cache. Thus, once a study chooses the constant-bits, a sample is only useful for a portion of the overall cache design space. There are even multi-level cache configurations such that no set samples exist, as when the block size of the secondary cache is larger than a primary cache. To avoid resampling, take care to ensure that a sample of a full trace is a set sample for the range of cache configurations being studied. This can be a difficult task, particularly when the range is large.

Another problem is that set sampling limits the performance metrics that can be gathered. Although the attention in this chapter focuses on estimating mean *MPI*, there are other important cache performance metrics that are difficult to gather when set sampling. The timing of interactions between accesses contained in different set samples is lost. For example, this might be important when estimating the performance of a write-buffer shared by all sets. Since the write-buffer handles write-back (or write-through) requests from all sets, it will be difficult to estimate its performance accurately knowing only the references to a portion of the sets. Thus, set sampling may not be appropriate for this type of analysis. Nevertheless, for those cases where set sampling is sufficiently flexible, it can be extremely useful.

4.3. Time Sampling

The alternative to set sampling is to characterize a trace with time samples. This section discusses key questions in producing and using time samples: (1) how to get unbiased *MPI* estimates from a single time-sample; and (2) what fraction of the trace data is (or equivalently, how many time samples are) needed for a desired accuracy, and how can the sampling accuracy be estimated?

4.3.1. Obtaining Unbiased Estimators from Time Samples

Obtaining accurate, unbiased estimators of mean *MPI* from individual time samples is difficult. The problem lies in eliminating the cold-start, or initialization, bias. By default, caches are often assumed to begin a time-sample simulation in the empty state. This implies that the first references of a time sample initialize the cache by loading new data in the cache. These *initialization (or unknown) references* are often assumed to be misses, since it is unknown whether they would have missed if the

cache were fully initialized. This approximation gives pessimistic (biased) *MPI* estimates because some references that initialize cache would have hit in a fully initialized cache. While it is a reasonable approximation for a long trace with a small cache, it is inaccurate for all but the biggest time-samples with the multi-megabyte caches considered in this dissertation. To find the exact number of misses in a time sample, it is necessary to know the state of the cache at the beginning of the sample. Unfortunately, this state is typically not known[16]. Furthermore, Chapter 3 shows that a time sample of many millions of instructions may be required to initialize a multi-megabyte set-associative cache fully, so it is prohibitively expensive to generate the cache state. Therefore, an effective time-sampling technique must not depend on knowledge of the initial cache state.

Several cold-start reduction techniques have been proposed; this chapter compares five different ones. The first, called COLD, is the default case that simply ignores the cold-start problem and assumes that all initialization (or cold start) references are misses. The second, called HALF, attempts to warm the cache to remove the cold-start bias. The first half of a time sample partially initializes the cache, and the rest estimates the *MPI*.

These two simple techniques are included with three more sophisticated ones. PRIME warms individual cache sets rather than the entire cache. It counts secondary cache references only after the corresponding set has been *primed*, that is, after initialization of all block frames in the set. For direct-mapped caches, the first reference to each set (the one that primes the set) is used only for priming, and the cache simulator counts the rest of the references to estimate the steady state miss ratio [STON90]. For higher associativities, the simulator counts references only when it touches a non-most-recently-used block frame after set priming [LAPI88]. PRIME directly estimates the secondary cache local miss ratio, not *MPI*, so all PRIME accuracy measurements compare the miss ratio[17]. These compares are nearly equivalent to *MPI* compares because *MPI* equals the local miss ratio times the secondary cache references per instruction. They are not exactly equivalent because primary cache initialization alters the secondary cache references per instruction.

STITCH approximates the state of the cache at the beginning of a sample by the state of the cache after the simulation of the previous sample. That is, the cache simulator *stitches* the samples together and simulates them as a single trace [AGHH88]. The simulator uses the first 1/4 of the instructions from the stitched trace to warm the cache.

The final cold-start reduction technique is INITMR. INITMR estimates μ, the fraction of cache initialization references that would have missed had the cache been warm. The actual misses, *A*, are those that would occur when simulating a given time-sample starting with a warm cache. $A = M + \mu U$, where *M* is the known misses and *U* is the initialization (unknown) references during the cold-start simulation of this time-sample (note that the simulation does not assume that initialization references are misses). INITMR estimates μ by $\hat{\mu}_{split}$ for each time-sample [WOHK91]. $\hat{\mu}_{split}$ depends on (1) the

16. Przybylski estimated the cache state by prefixing previously-referenced blocks onto the beginning of time samples [PRZY88]. With many trace-gathering techniques, it is impossible (or extremely difficult) to determine this prefix. Furthermore, with multi-megabyte caches the storage space required for the prefix can be as large as the time-sample itself.

17. This secondary cache local miss ratio is the secondary cache misses per secondary cache reference (aggregated over all sets) [PRHH89]. The simulator uses Primary cache misses and write-backs (from both primed and unprimed primary cache sets) to prime secondary cache sets. PRIME does not directly estimate the secondary cache *MPI* because primary cache misses, *not instructions*, prime the secondary cache sets.

fraction of time that cache block frames hold blocks that will not be referenced before being replaced, and (2) the fraction of block frames that the simulator initializes during the cold-start simulation of a sample. When the samples could not estimate (1), it was assumed to be 0.7.

To give a statistically significant comparison of the alternative cold-start elimination techniques, it is necessary to extract many samples from each trace and average the results. This focuses the attention on reducing the cold-start bias of a set of time-samples from a given workload, rather than an individual sample. The rest of Section 4.3 compares averages, rather than comparing each individual sample value. Thirty samples of length 100 thousand, 1 million, 10 million, and 100 million instructions were taken at equal intervals from each full trace used in this dissertation. The availability of the full traces allows the state of the cache at the beginning of each time-sample to be determined, and thus the *unbiased* (or true) misses of the thirty samples could be precisely calculated. Unbiased means there is no cold-start problem because the caches are properly initialized at the start of the sample. This section evaluates the cold-start elimination techniques by comparing their estimates to the average of the unbiased samples. In other words, this section finds how well the techniques eliminate the cold-start bias. Note that the unbiased average of the thirty samples is *not* the true value for the full trace. Section 4.3.2 varies the number of samples and compares *MPI* estimates to the full trace *MPI* to learn the sampling error. This section concentrates only on the bias, not the sampling error.

The comparison between the techniques is not simple. Simulations were run for the thirty samples of each trace. The simulated secondary caches are 1-megabyte, 4-megabyte, and 16-megabyte direct-mapped and 4-way set-associative with block sizes of 128 bytes. For different sample lengths, Table 4.5 shows the cold-start bias of the different cold-start reduction techniques using the thirty Mult1.2 samples. The data in Table 4.5 is for direct-mapped caches. The most striking characteristic is the large bias with the shortest samples and the biggest caches. The initialization references dominate the COLD estimates, for example. None of the techniques can eliminate the cold-start bias when the sample length is too short. Table 4.5 shows that the COLD estimates are always larger than the unbiased *MPI*, as expected. HALF and STITCH typically overestimate the unbiased *MPI*, while PRIME underestimates. Over the direct-mapped caches and sample lengths, INITMR produces the most accurate estimates for the thirty time samples from Mult1.2.

The results for the Mult1.2 trace in Table 4.5 are typical, though they change for different workloads. Table 4.6 shows errors for all traces used in this dissertation. The data in the table is for direct-mapped caches and samples of 10 million instructions. It shows that the accuracy trends are the same across the different traces. The Lin trace is an exceptional case where considerably lower unbiased *MPI*'s caused much larger errors. Similarly to Table 4.5, the results in Table 4.6 show that PRIME underestimates the unbiased *MPI* of direct-mapped caches. This is consistent with the observations of Laha, et al., [LAPI88]. STITCH produces substantial overestimates for all the traces except Sor. The overestimation of STITCH agrees with the results in Wood's appendix on stitching [WOOD90]. With the huge matrix traversals of the Sor workload, however, stitching gave an optimistic cache state estimate at the beginning of each sample. Normally, previous accesses have cleared out the cache while the data is being accessed. With stitching, there is a chance that useful data may have been available at the end of the previous sample of Sor. HALF tends to overestimate, particularly with the larger caches, but it is more accurate with longer samples. For the 16-megabyte cache with Mult1.2, HALF has a 100% bias with the 10 million instruction samples.

Cache Size	Sample Length	True Sample *MPI*×1000	COLD	HALF	PRIME	STITCH	INITMR
1M	0.1	1.73	+254%	+150%	-73%	+198%	+103%
	1	1.45	+88%	+74%	-50%	+57%	+21%
	10	1.57	+16%	+2%	-18%	+2%	+2%
	100	1.48	+2%	-3%	-3%	+0%	+0%
4M	0.1	0.87	+594%	+387%	-80%	+439%	+123%
	1	0.70	+262%	+234%	-73%	+164%	+63%
	10	0.77	+66%	+25%	-51%	+27%	-5%
	100	0.72	+12%	-3%	-23%	+6%	-2%
16M	0.1	0.35	+1629%	+1093%	-97%	+1201%	+400%
	1	0.31	+699%	+625%	-91%	+448%	+100%
	10	0.37	+200%	+103%	-80%	+90%	-3%
	100	0.33	+67%	+32%	-61%	+5%	-17%

Table 4.5. Accuracy of Cold-Start Techniques for Mult1.2.

This table shows the accuracy of the cold-start techniques for several sample lengths and direct-mapped caches with the Mult1.2 trace. For each cold-start bias reduction technique, the table shows the relative error of the estimated average *MPI* of the thirty time samples from the unbiased average *MPI* of the samples. (The secondary cache local miss ratio relative error, not the relative *MPI* error, is shown for PRIME.) An error of 100% means the estimate is double the unbiased value while an error of -50% means the estimate is half the unbiased value.

To find the effect of associativity on the alternative cold-start elimination techniques, Table 4.7 shows the same results as Table 4.6 for 4-way set-associative caches, rather than direct-mapped. The change in the errors of PRIME may be the largest between the tables. With higher associativity, the heuristic compensation of Laha, et al., reduces the PRIME underestimation bias. The simulator counts references to a primed 4-way associative set only when it touches a non-most-recently-used block, rather than counting immediately after set-priming as in the direct-mapped case. This heuristic removed the bias of PRIME with some success, but often its *MPI* estimate is still substantially in error.

Table 4.8 scores the different cold-start techniques based on the accuracy of their estimates and the comparison with the other estimation techniques. The table shows the times the estimate is within 10% of the unbiased value (10%) and the times the estimate is the closest (in percent) to the unbiased value (Win). Over all the cache configurations, traces, and sample lengths, INITMR produces the best estimates. INITMR had the most estimates within 10% of the unbiased value: 69 of 192. This is over twice as many as COLD, PRIME, or STITCH, and is 15% more than HALF. INITMR also produces the most accurate result about three times more often than any of the other techniques: 63% of the time INITMR was as good as or better than the others. When the time-sample lengths are short, or barely long enough, INITMR is superior to any of the other techniques examined in this study. As the sample lengths increase to where cold start is a smaller factor, HALF is also good. INITMR is generally the best cold-start reduction technique of those examined in this study.

There are several important problems with PRIME that make it less accurate than INITMR. The first is that PRIME is unsuccessful at removing all biases. Since the simulator counts the hits immediately following the priming of the set before any later misses, counting references immediately after set priming underestimates. This is particularly troublesome for direct-mapped caches. The heuristic compensation of Laha, et al., improves its estimates for caches of higher associativity [LAPI88], but it is still often inaccurate. The second important problem is that PRIME wastes the references used to prime the

Trace	Cache Size	True Sample *MPI*×1000	COLD	HALF	PRIME	STITCH	INITMR
Mult1	1M	1.45	+18%	+5%	-18%	+23%	+0%
	4M	0.62	+77%	+27%	-50%	+52%	-11%
	16M	0.28	+233%	+114%	-80%	+131%	-12%
Mult1.2	1M	1.57	+16%	+2%	-18%	+2%	+2%
	4M	0.77	+66%	+25%	-51%	+27%	-5%
	16M	0.37	+200%	+103%	-80%	+90%	-3%
Mult2	1M	1.21	+18%	+2%	-26%	+23%	-3%
	4M	0.60	+70%	+31%	-62%	+53%	-24%
	16M	0.25	+264%	+168%	-85%	+147%	-9%
Mult2.2	1M	1.18	+19%	+15%	-27%	+29%	-1%
	4M	0.62	+71%	+50%	-61%	+56%	-13%
	16M	0.29	+233%	+180%	-84%	+141%	-3%
Tv	1M	2.55	+4%	-0%	-33%	+32%	-2%
	4M	1.76	+15%	+9%	-56%	+37%	-4%
	16M	0.95	+79%	+61%	-76%	+71%	+37%
Sor	1M	15.68	+0%	-0%	-5%	-11%	-0%
	4M	8.08	+18%	+2%	-18%	-8%	+6%
	16M	2.00	+190%	+60%	-76%	-8%	+114%
Tree	1M	2.00	+13%	-0%	-10%	+29%	-1%
	4M	0.51	+107%	+8%	-50%	+43%	+24%
	16M	0.30	+217%	+35%	-77%	+69%	+18%
Lin	1M	0.75	+20%	+7%	-29%	-0%	+16%
	4M	0.06	+1113%	+535%	-62%	+217%	+903%
	16M	0.01	+4648%	+2248%	---%	+873%	+1037%

Table 4.6. Accuracy of Cold-Start Techniques With Direct-Mapped Caches.

For samples of 10 million instructions, this table shows the accuracy of the cold-start techniques for several direct-mapped caches with all the traces. The errors are calculated as in Table 4.5.

cache sets. Many references can be consumed initializing a set, particularly with larger set-associativities. For several sample lengths and secondary caches, PRIME lost so much information that it could not produce a statistically significant estimate.

Trace stitching also produces *MPI* estimates that are less accurate than INITMR. Usually, STITCH overestimated the unbiased *MPI*. STITCH did better with the longer samples because the time between the samples is shorter. With samples that are closely spaced in time, the cache state at the end of a previous sample may closely predict the state at the beginning of the next sample, but it doesn't in general. INITMR is superior in this study.

The simple HALF scheme produces better estimates than either PRIME or STITCH. For the longest samples, as many estimates are within 10% of the unbiased mean with HALF as with INITMR. This suggests that HALF is a useful technique for long samples. Over the widest range of sample lengths, however, INITMR is superior to HALF.

An accurate estimate of the initialization reference miss ratio, μ, is an important contribution of INITMR. Table 4.9 shows μ values and the accuracy of the $\hat{\mu}_{split}$ estimates of INITMR. Previous researchers postulated that μ values should be in a range similar to the miss ratio of the cache [STON90]. Similar to Wood, et. al [WOHK91], these results show μ values much larger than that, as high as 92%. μ depends on the fraction of time that a cache block frame is uselessly retained in the cache; it has little relationship to the overall cache miss ratio.

Trace	Cache Size	True Sample *MPI*×1000	COLD	HALF	PRIME	STITCH	INITMR
Mult1	1M	0.94	+21%	-5%	-6%	+36%	-11%
	4M	0.44	+106%	+29%	-51%	+80%	-4%
	16M	0.22	+313%	+157%	-99%	+167%	-8%
Mult1.2	1M	1.20	+15%	-5%	-9%	+6%	-7%
	4M	0.60	+81%	+21%	-40%	+43%	+1%
	16M	0.32	+232%	+118%	-57%	+104%	-3%
Mult2	1M	0.92	+14%	-5%	-18%	+33%	-16%
	4M	0.49	+84%	+34%	-64%	+68%	+2%
	16M	0.22	+316%	+202%	-78%	+170%	-9%
Mult2.2	1M	0.96	+16%	+10%	-14%	+38%	-10%
	4M	0.52	+84%	+54%	-52%	+73%	-1%
	16M	0.25	+285%	+221%	+15%	-161%	-14%
Tv	1M	2.14	+4%	-2%	-22%	+32%	-2%
	4M	1.53	+14%	+6%	+12%	+39%	-8%
	16M	0.82	+99%	+75%	+195%	+87%	+32%
Sor	1M	15.46	+0%	-0%	+0%	-11%	-0%
	4M	8.57	+9%	-1%	-12%	-8%	-2%
	16M	2.17	+158%	+34%	-81%	-4%	+60%
Tree	1M	1.60	+11%	-3%	-9%	+35%	-6%
	4M	0.41	+124%	-5%	-32%	+70%	+18%
	16M	0.25	+263%	+38%	+83%	+77%	-17%
Lin	1M	0.69	+26%	+6%	+9%	+6%	+21%
	4M	0.02	+2763%	+1322%	+81%	+778%	+1797%
	16M	0.01	+4648%	+2248%	---%	+873%	+1037%

Table 4.7. Accuracy of Cold-Start Techniques With 4-Way Set-Associativity.

For samples of 10 million instructions, this table shows the accuracy of the cold-start techniques for several 4-way set-associative caches with all the traces. The errors are calculated as in Table 4.5.

A major contribution of Wood, et. al [WOHK91], was a renewal-theoretic model of block residences in the cache as generations. One generation of a cache block frame is the time that it holds a given block. A cache block frame is *dead* when the next reference to it causes a miss that loads a new block (and starts a new generation). When the assumptions of the renewal-theoretic model of Wood, et. al, hold, $\mu = \frac{D}{G}$, where *D* is the average *dead* time, and *G* is the average generation time. In practice, *D/G* is a good estimate of the fully initialized μ although the assumptions of the model are violated. Furthermore, the behavior of μ can be estimated by $\hat{\mu}_{split}$ when the sample does not entirely initialize the cache. $\hat{\mu}_{split}$ is a function of both *D/G* and the fraction of the cache that the simulator initializes by the end of the sample. When the sample entirely initializes the cache, $\hat{\mu}_{split}$ is the estimated value for *D/G*. When the sample initializes only a small portion of the cache, $\hat{\mu}_{split}$ decreases because μ is smaller; the first initialization references are less likely to be misses than the later ones.

Table 4.9 shows decreasing values for μ as the cache size increases and the sample length decreases. This is precisely when the portion of the cache that the sample initializes decreases, which confirms that the first initialization references are less likely to be misses and that μ is smaller when the sample initializes less. Examination of μ values for set-associative caches, beyond the direct-mapped cache results shown in Table 4.9, shows similar μ behaviors, though the absolute values of μ tended to increase slightly with associativity.

Cache Size	Sample Length (Mill)	COLD		HALF		PRIME		STITCH		INITMR	
		10%	Win	10%	Win	10%	Win	10%	Win	10%	Win
1M	0.1	2	0	2	5	0	2	0	0	0	9
	1	2	1	4	4	3	5	1	2	3	5
	10	4	2	15	13	7	1	4	3	12	8
	100	16	5	16	7	16	6	6	2	16	12
4M	0.1	0	0	0	0	0	1	1	2	1	13
	1	0	0	0	1	1	2	2	1	4	13
	10	1	0	6	6	0	1	2	1	10	8
	100	7	1	14	4	3	2	5	3	12	7
16M	0.1	0	0	0	0	0	0	0	0	0	16
	1	0	0	0	0	0	0	1	2	0	14
	10	0	0	0	0	0	1	2	4	6	11
	100	0	0	3	2	1	0	5	9	5	5
All	0.1	2	0	2	5	0	3	1	2	1	38
	1	2	1	4	5	4	7	5	5	7	32
	10	5	2	21	19	7	3	8	8	28	27
	100	23	6	33	13	20	8	16	14	33	24
All	All	32	9	60	42	31	21	29	29	69	121

Table 4.8. Scoring of Different Cold-Start Techniques.

This table scores the accuracy of the different cold-start techniques for all the different traces. It gives a ''10%'' point each time the average estimate of thirty time-samples is within 10% of their unbiased average. It gives a ''Win'' point each time the estimate is closest ($\log(\frac{estimate}{unbiased})$ closest to 0.0) of all schemes to the unbiased average of the thirty samples. Multiple points are awarded on ties, so the winners may not be constant for each row. The maximum points is 192 in each category at the bottom. The scores shown are the sum for direct-mapped and 4-way caches.

$\hat{\mu}_{split}$ is more accurate when the cache size is smaller and the sample is longer. There are two reasons for this: (1) μ can be more tightly bounded as the cache becomes more fully initialized, and (2) more misses give a more accurate estimate of *D/G*. When a sample fully initializes a cache during a cold-start simulation, the μ value for the sample tends to be close to *D/G*, making prediction simple. It is more difficult to predict μ when the sample does not fully initialize the cache, however, because μ then depends on the distributions of the generation and dead times. $\hat{\mu}_{split}$ does not use knowledge of these distributions. Instead, it is the middle value between upper and lower bounds given by Wood, et. al [WOHK91]. These bounds tighten as the fraction of the cache initialized by a sample increases. Thus, the error of $\hat{\mu}_{split}$ decreases with the initialization fraction, provided *D/G* is known. *D/G* can be estimated remarkably well with few cache block frame lifetimes, but more lifetimes give a higher accuracy. More lifetimes can be provided by a longer sample.

The combination of a more fully initialized cache and many misses leads to the most accurate estimates of μ as the sample length increases and the cache size decreases. When the samples are too short, Table 4.9 shows that $\hat{\mu}_{split}$ can poorly estimate μ. The Lin samples are a particular example of this, where samples of 100,000 instructions initialize about 0.11% of the 16-megabyte cache, an extremely small portion. This leads to a poor estimate by $\hat{\mu}_{split}$.

The ultimate usefulness of INITMR is determined not by the accuracy of the μ estimate, but by the accuracy of its *MPI* estimates. Inaccuracies in the μ estimates do not translate into *MPI*

μ Values (and Error of $\hat{\mu}_{split}$)					
Trace	Cache Size	Sample Length (Millions of Instructions) 0.1	1	10	100
Mult1	1M	0.25 (68%)	0.28 (115%)	0.49 (3%)	0.65 (-4%)
	4M	0.12 (102%)	0.15 (186%)	0.38 (-25%)	0.66 (-14%)
	16M	0.05 (258%)	0.09 (380%)	0.26 (-19%)	0.50 (-35%)
Mult1.2	1M	0.21 (111%)	0.36 (40%)	0.54 (11%)	0.63 (0%)
	4M	0.13 (96%)	0.21 (98%)	0.41 (-12%)	0.65 (-11%)
	16M	0.08 (143%)	0.12 (121%)	0.27 (-11%)	0.49 (-36%)
Mult2	1M	0.19 (67%)	0.35 (27%)	0.59 (-14%)	0.67 (-2%)
	4M	0.10 (73%)	0.24 (39%)	0.46 (-39%)	0.63 (0%)
	16M	0.05 (230%)	0.14 (146%)	0.25 (-15%)	0.42 (-18%)
Mult2.2	1M	0.18 (102%)	0.37 (33%)	0.58 (-6%)	0.68 (-3%)
	4M	0.13 (83%)	0.25 (41%)	0.44 (-27%)	0.62 (-2%)
	16M	0.09 (90%)	0.15 (81%)	0.27 (-11%)	0.42 (-1%)
Tv	1M	0.20 (70%)	0.47 (-21%)	0.74 (-21%)	0.82 (1%)
	4M	0.15 (40%)	0.39 (25%)	0.62 (-27%)	0.75 (0%)
	16M	0.08 (148%)	0.24 (50%)	0.38 (62%)	0.45 (33%)
Sor	1M	0.92 (-56%)	0.90 (-17%)	0.91 (-5%)	0.91 (-5%)
	4M	0.55 (-47%)	0.54 (15%)	0.55 (15%)	0.55 (7%)
	16M	0.17 (17%)	0.18 (200%)	0.30 (122%)	0.58 (-9%)
Tree	1M	0.14 (369%)	0.24 (71%)	0.51 (-32%)	0.65 (-12%)
	4M	0.04 (1450%)	0.13 (122%)	0.41 (-22%)	0.76 (-29%)
	16M	0.03 (729%)	0.11 (223%)	0.35 (-28%)	0.71 (-60%)
Lin	1M	0.25 (5%)	0.36 (43%)	0.40 (115%)	0.66 (32%)
	4M	0.02 (1178%)	0.05 (418%)	0.08 (758%)	0.17 (312%)
	16M	0.01 (1621%)	0.03 (463%)	0.05 (277%)	0.16 (61%)

Table 4.9. μ and the Accuracy of its Estimator ($\hat{\mu}_{split}$).

This table shows average μ values for the thirty samples and the error of the INITMR $\hat{\mu}_{split}$ average estimates (+100% means the estimate is double of the true value and -50% means the estimate is half the true value). These results are shown for direct-mapped secondary caches and several sample lengths.

inaccuracies of the same magnitude. INITMR mitigates the effect of μ errors because it includes known misses, in addition to misses from initialization references, in the *MPI* estimate. Known misses are those that are not a result of an initialization reference.

For the thirty samples of different lengths taken from each trace, Table 4.10 shows the error of the INITMR *MPI* estimates. The results clearly show that samples of 100,000 instructions are too short to overcome the cold-start bias of the large caches considered in this dissertation. (This is not too surprising since the largest caches contain over 100,000 cache block frames!) INITMR cannot accurately estimate *MPI* when the simulation spends most of the references initializing the cache. This is why an accurate estimate of the *MPI* of the larger caches is difficult with the shorter samples, and it is also why the Lin estimates are inaccurate. INITMR requires many known misses to obtain an accurate estimate of the *MPI*. This is partially because μ cannot be accurately estimated without many known misses: known misses are needed to estimate the fraction of time that block frames hold blocks that are not referenced before being replaced (D/G). Furthermore, with many known misses ($M >> U$), accurate approximation of μ is less essential because known misses are a larger portion of the total misses. In Table 4.10 the cases where initialization references fill at least half the cache and there are more known misses than initialization references are marked with a '*'. These markings occur with longer samples

and smaller cache sizes since a larger portion of all misses are known misses in these cases. Larger caches require more instructions to initialize, which implies a larger sample is needed for the same accuracy.

Trace	Cache Size	Full Trace *MPI*×1000	Sample Length (Millions of Instructions) 0.1	1	10	100
Mult1	1M	1.55	86%	47%	0%*	0%*
	4M	0.70	156%	120%	-11%	-3%*
	16M	0.33	281%	335%	-12%	-17%
Mult1.2	1M	1.45	103%	21%	2%*	0%*
	4M	0.69	123%	63%	-5%	-2%*
	16M	0.32	400%	100%	-3%	-17%
Mult2	1M	1.24	49%	20%	-3%*	0%*
	4M	0.61	48%	39%	-24%	0%*
	16M	0.26	212%	146%	-9%	-3%
Mult2.2	1M	1.18	127%	24%	-1%*	0%*
	4M	0.59	127%	60%	-13%	0%*
	16M	0.27	170%	106%	-3%	8%
Tv	1M	2.63	36%	-10%	-2%*	0%*
	4M	1.88	34%	-9%	-4%	0%*
	16M	1.03	145%	39%	37%	12%
Sor	1M	14.77	-41%	-3%*	0%*	0%*
	4M	7.54	-27%	44%	6%*	0%*
	16M	1.97	83%	386%	114%	-2%*
Tree	1M	2.16	249%	36%	-1%*	0%*
	4M	0.59	1407%	121%	24%	-7%*
	16M	0.30	796%	198%	18%	-37%
Lin	1M	1.16	-30%	-14%	16%	1%*
	4M	0.09	1437%	946%	903%	113%
	16M	0.02	2567%	1318%	1037%	176%

Table 4.10. Accuracy of INITMR Time-Sample MPI Estimates.

This table shows the relative error of the time-sample INITMR *MPI* estimates over a range of direct-mapped secondary cache sizes and sample lengths. The error is the difference between the average *MPI* estimate of the thirty samples and the unbiased average *MPI* of the thirty samples (as a percent of the unbiased average). A '*' denotes the constraints that: (1) the caches were at least half initialized by the samples (on average, $U \geq 0.5 \times CacheSize$), and (2) there were more known (non-initialization) misses than initialization references (on average, $M \geq U$).

When the ratio of known misses to unknown misses increases, the maximum error of INITMR shrinks. The assumption that none of the initialization references miss ($\hat{\mu} = 0.0$) gives a lower bound on the unbiased *MPI*. Similarly, the assumption that all initialization references hit ($\hat{\mu} = 1.0$) gives an upper bound. Since $\hat{\mu}_{split}$ gives a more accurate estimate of μ than either the lower bound or the upper bound, the error of the *MPI* estimate from INITMR is usually much smaller than these bounds, particularly with more known misses.

When there is a '*' in the entry in Table 4.10, INITMR always produces estimates within 10% of the unbiased values. This is a useful property of INITMR: when the constraints listed in Table 4.10 are satisfied, the resulting *MPI* estimates are unbiased. Unfortunately, simulations require sample lengths of millions of instructions to meet these restrictions with multi-megabyte caches. For the traces in this study, samples of 1-10 million instructions were usually enough for 1-megabyte caches, 10-100 million instructions for 4-megabyte caches, and 100 million instructions or more are needed for 16-megabyte

caches. These results agree with the rule-of-thumb that trace length should be increased by a factor of eight each time the cache size is quadrupled [STON90]. Accurate results could be obtained with shorter samples if the *MPI*'s of the traces were higher. With the larger caches, the Lin trace has *MPI*'s that are too low to get enough misses, even with samples of 100 million instructions, but the Sor trace had high enough *MPI*'s to produce good estimates with much shorter samples.

Table 4.10 shows INITMR results for direct-mapped caches, but similar results have been observed for caches of higher associativity. Since these caches have a lower *MPI* and more initialization references, simulations require longer samples to produce the same ratio of known misses to initialization references. This makes it more difficult to obtain accurate *MPI* estimates. However, with a sufficiently high ratio of known to initialization references, accuracy is assured with set-associative caches just as with direct-mapped caches.

Results from multiple samples can be unified in several ways with INITMR. An alternative to calculating an individual $\hat{\mu}_{split}$ for each sample, as done in all other results in this section, is to merge the statistics of the samples and produce a single $\hat{\mu}_{split}$ that is used for all. The potential advantage of merging is a more accurate *D/G* estimate. The disadvantage is that different phases of execution could cause large differences in the actual μ across samples, even if the samples come from the same full trace; treating them the same could lead to inaccuracies. Furthermore, some samples could inordinately bias $\hat{\mu}_{split}$ and consequently distort the *MPI* estimate.

For the thirty samples of 10 million instructions taken from each full trace, Table 4.11 shows the accuracy of individual and merged $\hat{\mu}_{split}$ calculations. For the merged $\hat{\mu}_{split}$, *D* and *G* are calculated with the statistics of the generations from all samples, disregarding the differences in the generations across samples. The fraction of the cache that is initialized is the average over all the cold-start simulations.

Table 4.11 shows that there is a slight advantage to the individual $\hat{\mu}_{split}$ calculation. This is true over all the sample lengths and associativities. It shows that merging statistics from multiple samples, even if the samples come from the same full trace, may not give accurate results. The phase changes across the different samples leads to distortions in the INITMR *MPI* estimate when the samples are merged. While merging may be a good idea for samples that are similar, these samples were sufficiently different to produce poor results. This chapter uses the individual calculation of $\hat{\mu}_{split}$.

4.3.2. What Fraction of the Full Trace is Needed?

This section finds the time-sampling error by comparing time-sample mean *MPI* estimates to full trace results. It shows the errors for varying portions of the full trace data, or equivalently when the sample size is constant, varying numbers of time-samples. The *MPI* estimate from a single time-sample has a much larger variance than the *MPI* estimate from a set-sample of the same size because each time-sample captures only one (or perhaps a few) phases of workload execution. Even over long intervals of execution, the *MPI* of different time samples fluctuates widely because cache performance changes with the different execution phases. This makes it necessary to have many time samples to predict the mean *MPI* of a full trace accurately. More samples scattered across the entire execution of an application will improve estimation accuracy, but they also may increase required storage space and simulation times; the sample size and the number of samples gives the trace data requirements for time sampling.

Trace	Size	Full Trace *MPI*×1000	$\hat{\mu}_{split}$ Calculation Individual	Merge
Mult1	1M	1.55	0%	1%
	4M	0.70	-11%	-10%
	16M	0.33	-12%	21%
Mult1.2	1M	1.45	2%	3%
	4M	0.69	-5%	-11%
	16M	0.32	-3%	17%
Mult2	1M	1.24	-3%	-3%
	4M	0.61	-24%	-20%
	16M	0.26	-9%	35%
Mult2.2	1M	1.18	-1%	0%
	4M	0.59	-13%	-16%
	16M	0.27	-3%	24%
Tv	1M	2.63	-2%	0%
	4M	1.88	-4%	2%
	16M	1.03	37%	-3%
Sor	1M	14.77	0%	0%
	4M	7.54	6%	5%
	16M	1.97	114%	190%
Tree	1M	2.16	-1%	0%
	4M	0.59	24%	34%
	16M	0.30	18%	39%
Lin	1M	1.16	16%	19%
	4M	0.09	903%	1014%
	16M	0.02	1037%	1562%

Table 4.11. Individual Versus Merged $\hat{\mu}_{split}$ Estimation.

This table shows the errors of different ways to calculate $\hat{\mu}_{split}$ (for the thirty samples). The alternatives are to calculate $\hat{\mu}_{split}$ for each individual trace or to calculate a single merged $\hat{\mu}_{split}$ for all the samples. The table shows errors of the *MPI* estimates from the unbiased *MPI* of the samples. The results are for direct-mapped caches and samples of 10 million instructions.

Figure 4.5 illustrates this sampling effect for the Mult1.2 trace; the left graph illustrates the case where there is no cold-start bias (unbiased). Each shaded area corresponds to time sampling with a different sample length. The area displays, for varying portions of the trace data, the interval where 90% of the *MPI* estimates fall (5% of the estimates are above and 5% are below). Each individual *MPI* estimate consists of samples spaced periodically across the full trace to capture pieces of as many different execution phases as possible. Since a unique portion of the trace data produces each *MPI* estimate, the number of *MPI* estimates is inversely proportional to the trace data fraction. The areas are in the shape of a cone with a certain thickness along the y-axis and length along the x-axis. With no cold-start bias, all the cones are almost centered on the ratio of 1.0, which says that the sample estimates correctly predict the full trace *MPI* (on average). The left-most edge of each cone represents a single sample; the thickness of the edge is the interval where 90% of the single-time-sample *MPI* estimates fall. Shorter sample lengths produce longer cones because a single sample is a smaller portion of the full trace data. The distinctive cone shape is produced because adding more samples (i.e. using a larger portion of the full trace data) reduces the variance of the aggregated *MPI* estimate. Note that the cone thickness may not strictly decrease with increasing fractions of the trace data because each estimate captures different execution phases for each fraction. The cone thickness decreases by approximately a $\frac{1}{\sqrt{n}}$ factor as the number of samples[19] (n) increases because the standard deviation of the sample mean decreases at

19. The number of samples at a given fraction of the trace data is the distance from the left end of the cone. The number of samples at the right end of each cone is 30 one hundred million instruction samples and

approximately that rate.

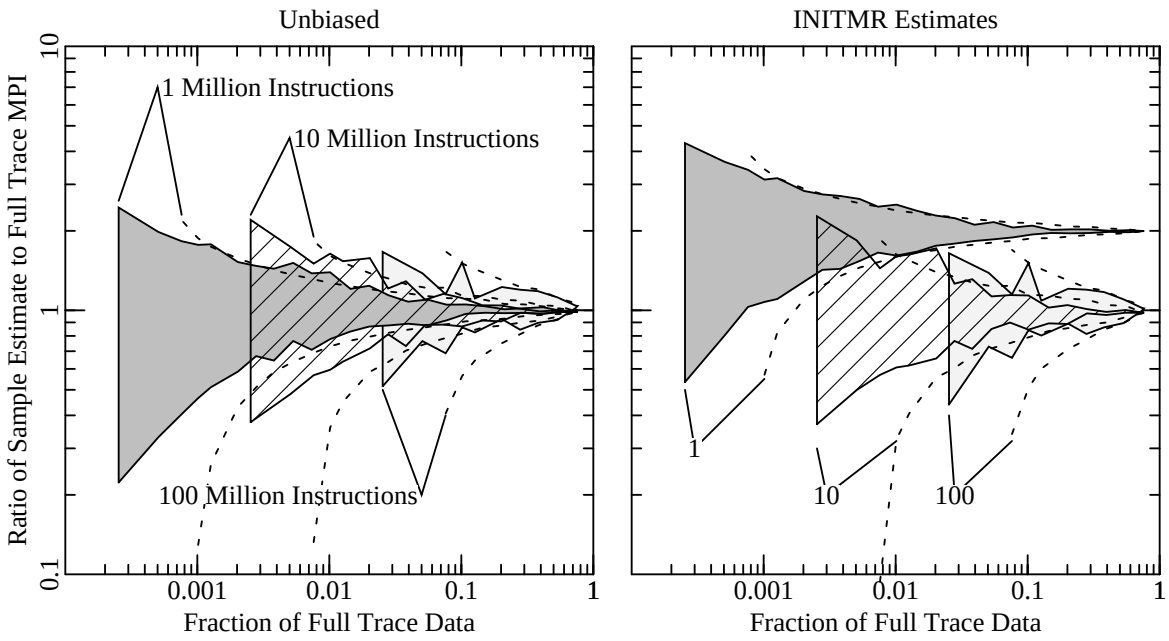


Figure 4.5. Accuracy of Time Sampling with Mult1.2.

This figure has areas (cones) for different sample lengths from the Mult1.2 trace showing the accuracy of time sampling for a 4-megabyte direct-mapped secondary cache. The left graph shows the unbiased accuracy results while the right graph is for the INITMR Estimates. For a given fraction of the full trace data, the enclosed interval shows where 90% of the *MPI* estimates fall. The left end of each area corresponds to a single sample. The dotted lines show the mean 90% confidence intervals (upper and lower bound) for each sample size.

Figure 4.5 also shows the average 90% upper and lower confidence bounds (dashed lines) for each cone. Section 2.5 shows how to calculate the confidence intervals. Note that the log scale distorts the lower bound; the average upper and lower bounds are (linearly) equi-distant from the center of their corresponding cone. With only a few samples, the 90% confidence intervals may not successfully contain the true mean 90% of the time because the *MPI* of the individual time samples is not normally distributed. With 30 or more samples, however, the 90% confidence intervals typically did contain the true mean 90% or more of the time. This shows that the confidence interval techniques from Section 2.5 are appropriate with 30 or more samples.

The shape of the cones (and their corresponding average confidence bounds) suggests the need for many samples. At a given fraction of the full trace data, the accuracy of time sampling is the thickness of the cone. The cone may be narrow enough (or equivalently, the variance of the sampling estimates may be small enough) only after hundreds of samples from the Mult1.2 trace. Laha, et al., suggest that 35 samples were enough to characterize their traces [LAPI88]. The cones in Figure 4.5 suggest that, in general, more samples may be needed, particularly for high accuracy with the shorter sample lengths. Instead of strictly using 35 samples, it is better to have more samples as the sample length decreases to keep accuracy (cone thickness) constant. It is intuitive that more samples should be used since, in the

3000 one million instruction samples.

extreme case, 35 samples of a single instruction could hardly be expected to estimate an *MPI* of about one per thousand instructions. The variance of the sample *MPI* estimates is slightly larger with smaller samples because each sample includes less trace data, so more samples are needed for a desired accuracy. For example, with no cold-start bias, to ensure with 90% probability that the *MPI* error is less than 10%, 200 samples of 1 million instructions, 65 samples of 10 million instructions, and 20 samples of 100 million instructions may be required. This is roughly a factor of three increase as the sample size decreases by a factor of ten.

Placing the cones for different sample lengths on the same graph in Figure 4.5 allows and examination of the tradeoffs among different sample lengths. For large fractions of the trace data with no cold-start bias, all sample lengths give estimates near the true mean *MPI* of the full trace. The cones for the shorter sample lengths are more narrow at a given fraction of the full trace data because there are many more samples to reduce the variance of the *MPI* estimate. This means that a smaller fraction of the full trace is needed to assure a given accuracy with shorter samples. In the absence of cold-start bias, it is better to have many small samples than a few large ones because more smaller samples capture more execution phases; samples of a single instruction would give the most accurate results at a fixed portion of the trace data.

But the cold-start bias plays an important role in time-sampling accuracy. The previous section shows that the bias is larger for shorter samples than for long samples. Although INITMR minimizes the bias, it does not eliminate it entirely. The graph on the right side of Figure 4.5 shows the time-sampling results using INITMR to compensate for cold start. This graph is similar to the one with no cold-start bias (on the left) except that the 1-million instruction cone shifts away from the value of 1.0 where it correctly predicts the full trace mean *MPI* (on average). This shift occurs because the cold-start bias could not be eliminated by INITMR. INITMR may either overestimate or underestimate μ, and consequently the cones are either shifted up or down. The overall shape of the cones remains largely unchanged, which suggests that INITMR preserves the distribution of the sample *MPI* estimates, except for the bias.

Figures 4.6a and 4.6b show the INITMR cones for the rest of the traces. In Figure 4.6a, the multiprogrammed traces have similar cones to Mult1.2. The cold-start bias is larger for Mult2 and Mult2.2, however; samples of 10 million instructions are insufficient to remove the bias in the 4-megabyte cache. Tree gives wide cones with few samples because the sample variance is high, but the cone thins out well because larger portions of the full trace data reduce the variance of the *MPI* estimate. For single samples (the left edge of the cones), Tree has wider intervals with 10 million instruction samples than with the 1 million instruction samples. While the range of the Tree 1 million instruction samples is much larger than the range of the 10 million instruction samples, 90% of the 1 million samples fall within tighter bounds because there are many more samples and more cushion for extreme sample values.

Figure 4.6b shows that the Tv *MPI* estimates also have a high variance with few samples, and the cone thickness does not decrease as quickly as the other traces with larger trace data fractions. The phase behavior of Tv causes this. For much of the trace, it is building its large data structure. When Tv traverses the structure for a short time at the end of the trace it gives much worse cache performance and there are many phase changes. It is difficult to capture the true mean performance of this final Tv stage with only a few time samples. The Sor trace cones are abnormal because they appear asymmetric about their center, with lower values captured in the cone. This occurs because the time samples of Sor

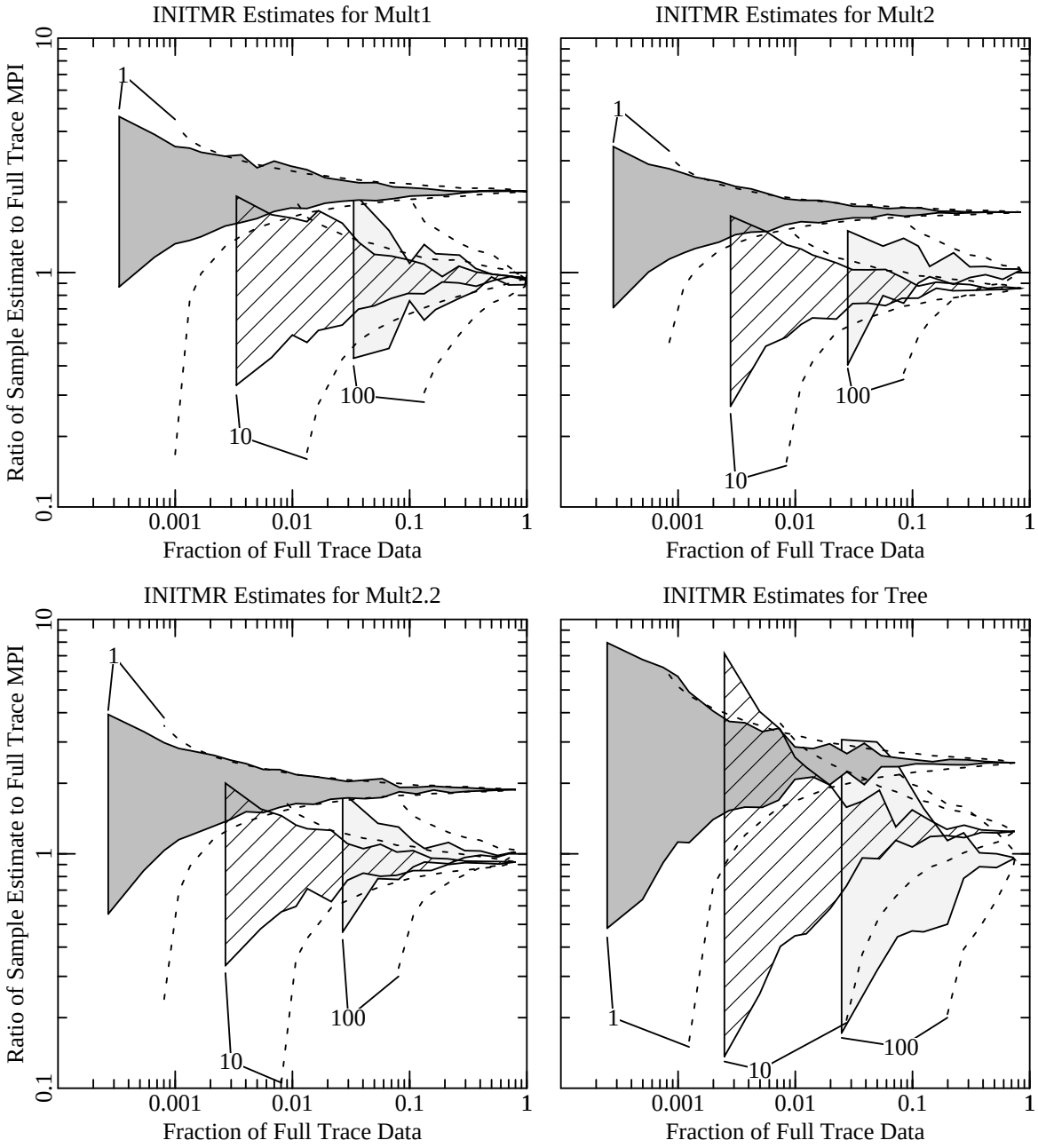


Figure 4.6a. Accuracy of Time Sampling with Mult1, Mult2, Mult2.2, and Tree.

This figure shows accuracy cones like those in Figure 4.5 for the Mult1, Mult2, Mult2.2, and Tree traces. The areas correspond to time-sample lengths of 1, 10, and 100 million instructions. The enclosed area shows where the *MPI* estimates for 4-megabyte direct-mapped secondary caches fall with 90% probability. The dashed lines show the average estimated 90% confidence interval (upper and lower bound).

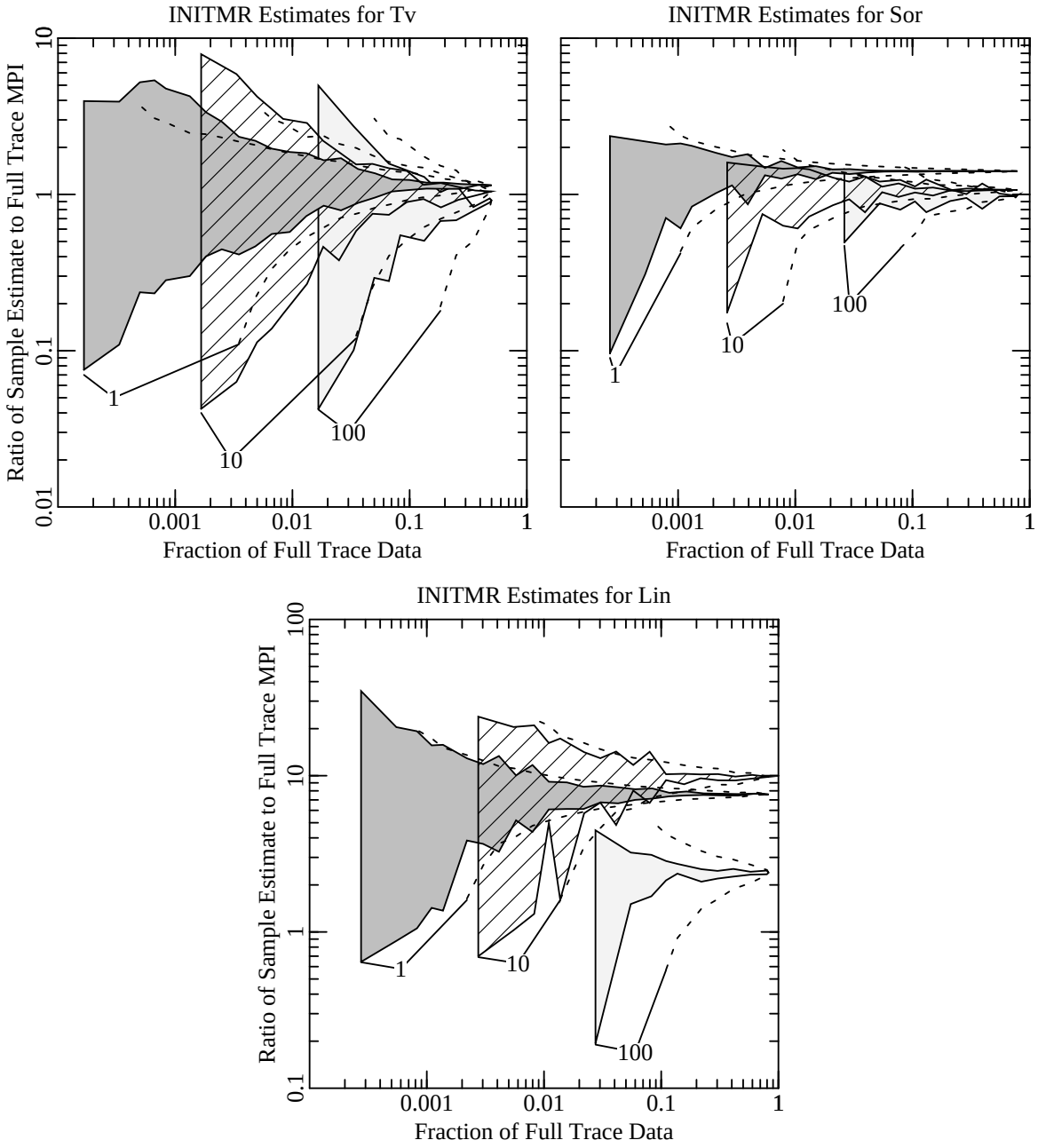


Figure 4.6b. Accuracy of Time Sampling with Tv, Sor, and Lin.

This figure shows accuracy cones like those in Figure 4.5 for the Tv, Sor, and Lin traces. The areas correspond to time-sample lengths of 1, 10, and 100 million instructions. The enclosed area shows where the *MPI* estimates for 4-megabyte direct-mapped secondary caches fall with 90% probability. The dashed lines show the average estimated 90% confidence interval (upper and lower bound). Note that these graphs are at a different scale.

tend to have a bi-modal distribution, with either high or low *MPI*'s. The low portion of the Sor curve shows that the low values occurred more than 5% of the time. The poor *MPI* estimates of INITMR for the Lin trace are evident. The cold-start bias dominates the Lin results, particularly for shorter sample lengths, since the *MPI* is so low.

Cache size has a large effect on INITMR time-sampling accuracy. Figure 4.7 shows the Mult1.2 cones for 1-megabyte and 16-megabyte caches. The cones are similar to the 4-megabyte cones in Figure 4.5, but their shape changes, showing the different distribution of *MPI* estimates for the different cache sizes. By comparing with the INITMR results in Figure 4.5, the larger cold-start effect with bigger caches is evident. While samples of 1 million instructions were clearly insufficient for the 4-megabyte cache, they have a smaller bias for the 1-megabyte cache. This implies that accuracy can be achieved with a smaller portion of the full trace data, less than that needed with the 4-megabyte caches. The cones for the 16-megabyte cache all show a cold-start bias in Figure 4.7. This illustrates the need for long samples to analyze these large caches, more than one hundred million instructions of the traces considered in this dissertation.

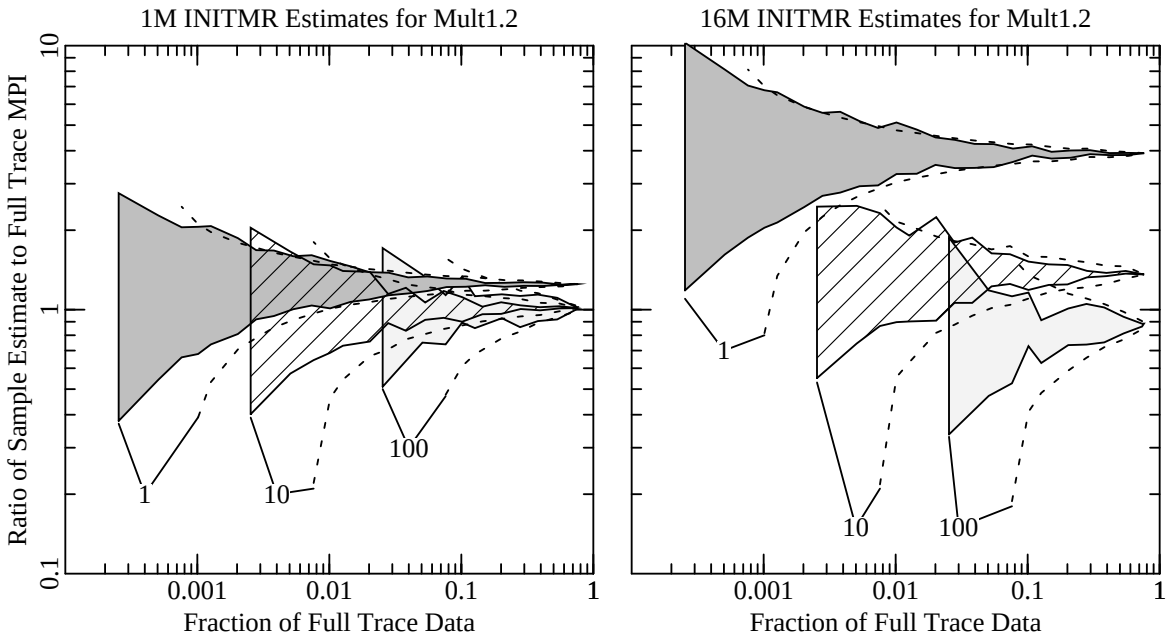


Figure 4.7. Accuracy of Time Sampling for Different Cache Sizes.

This figure shows Mult1.2 accuracy cones like those in Figure 4.5 for direct-mapped secondary caches of 1-megabyte and 16-megabytes. The areas correspond to time-sample lengths of 1, 10, and 100 million instructions. The enclosed area shows where the *MPI* estimates for the secondary caches fall with 90% probability. The dashed lines show the average estimated 90% confidence interval (upper and lower bound).

Since the range of the *MPI* estimates is as important as the interval that 90% of the estimates fall in, Figure 4.8 shows the range of the estimates for the Mult1.2 trace. It can be considerably higher for few samples, particularly for the shorter sample lengths. The range grows as the sample length decreases since with shorter samples there is greater opportunity for variance. Furthermore, there are more samples included in the statistics for the shorter samples, which increases the probability of an extreme value. The range of an individual cone decreases as the amount of trace data increases, of course, since more samples average together to moderate the values.

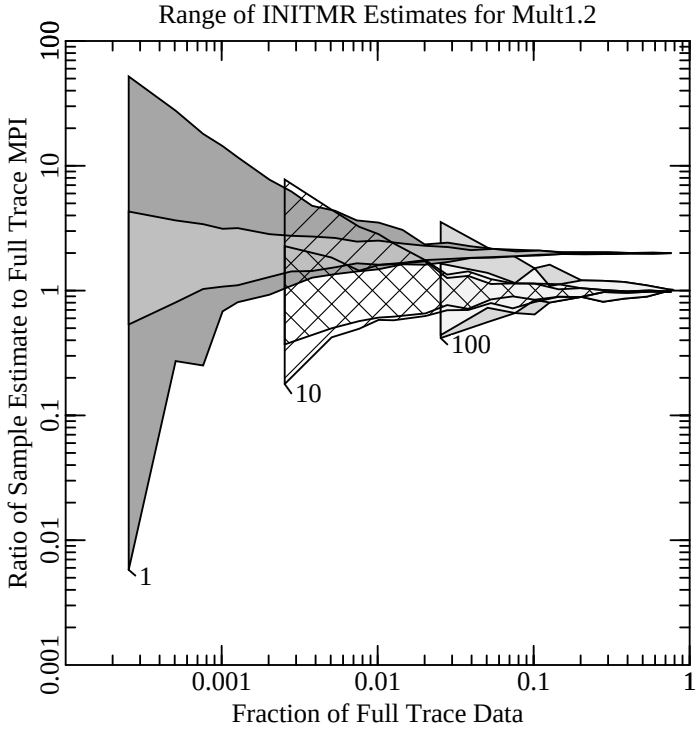


Figure 4.8. Range of Time-Sampling Estimate for Mult1.2.

This figure shows Mult1.2 accuracy cones where 90% of the estimates fall, similar to those in Figure 4.5, with the enclosing range of all the estimates. The areas correspond to the *MPI* estimates of a 4-megabyte direct-mapped cache with time-sample lengths of 1, 10, and 100 million instructions.

The cones for the different workloads illustrate that obtaining accurate time-sampling results requires solving two sub-problems: removing the cold-start bias (centering the cone about 1.0 on the graph), and removing the sampling error (using enough samples to be in the narrow portion of the cone). To the first order, these problems are independent of each other; the thickness of the cone (or the variance of the *MPI* estimates) is the same with or without the bias, and the cold-start bias is present for the shorter time-samples no matter what fraction of the trace data is used. This decoupling of the problems is useful since it allows them to be solved independently. If a cold-start simulation of a sample has more known than initialization misses and fills the cache at least half full (i.e. if it satisfies the constraints listed in Table 4.10 of Section 4.3.1), INITMR can remove the cold-start bias. Then, a large enough portion of the full trace data will ensure accurate results. The problem is that the cold-start constraints require long time-samples, and many samples are needed to reduce the sample variance and the thickness of the cone.

Time sampling becomes less effective for larger caches because the cold-start problem is worse. The problem is bad enough that time sampling may not meet the 10% sampling goal for the largest caches. Given the sample length increases needed for larger caches, and since many samples are needed, roughly 2/3 more trace data is required each time the cache size is doubled. Accurate time sampling requires much trace data with large caches. For example, with Mult1.2 (Figure 4.5), the 10 million instruction samples must be used rather than the 1 million ones because of the cold-start bias. To get the 10 million instruction cone within 10%, at least 10% of the full trace data is needed. Bigger

caches require an even larger portion of the trace data because the cold-start bias is worse. Even the 100 million instruction samples from Mult1.2 were not long enough to eliminate the bias for a 16-megabyte cache. Consequently, the sampling goal cannot be met. The need for ever-larger trace data quantities with larger caches is a distinct disadvantage of time sampling.

In considering INITMR to this point, only its accuracy in predicting the *mean MPI* has been shown. Prediction of the distribution of the individual samples is also useful to characterize the *MPI* time-behavior. For the 10-million instruction samples from Mult1.2, Figure 4.9 compares the distribution of 300 INITMR *MPI* time-samples, the same 300 unbiased *MPI* time-samples, and the unbiased *MPI* of all 390 samples across the full Mult1.2 trace. All of the distributions are similar. This shows that 300 samples can adequately characterize the full 390 samples in the trace. Furthermore, although the samples were not long enough to receive a '*' in Table 4.10, INITMR still gives a similar distribution to the unbiased samples. This shows that INITMR preserves the *MPI* distribution of time samples, even when the samples are too short.

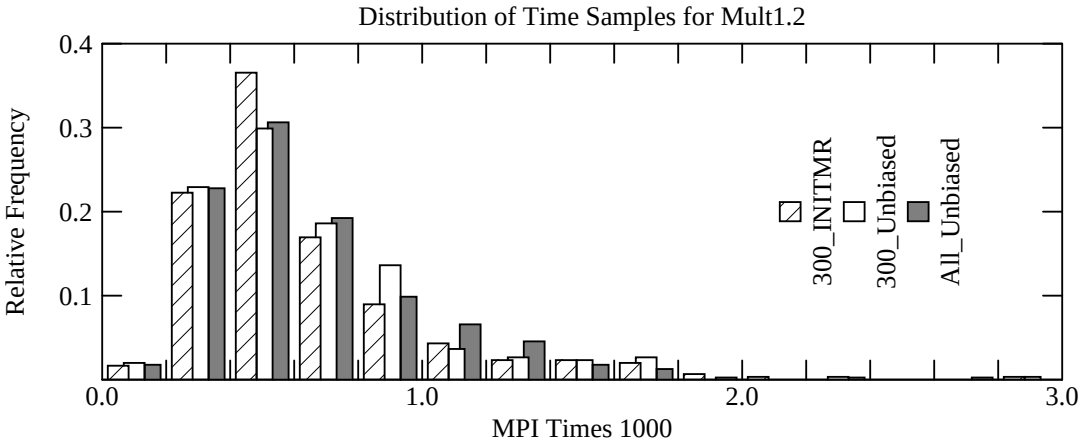


Figure 4.9. Time-Sample Distributions.

This figure shows, for samples of 10 million instructions from the Mult1.2 trace, the distribution of the *MPI* samples for a 4-megabyte direct-mapped cache. The x-axis partitions the samples into bins based on their *MPI*; a portion of all samples fall in each bin. The figure shows the distribution of all 390 unbiased samples from the full trace, 300 unbiased samples, and the same 300 INITMR samples.

4.3.3. Advantages and Disadvantages of Time Sampling

The major problem with time sampling is cold start, which is exacerbated by multi-megabyte caches. INITMR is the most accurate cold-start reduction technique considered in this chapter. It is accurate because it predicts the initialization reference miss ratio (μ). A sample should fill at least half the cache and have more known misses than initialization references for accurate INITMR estimates. Unfortunately, these are rigorous requirements for all but the largest time-samples. One implication of this is that cold start imposes stringent usage restrictions on an individual time-sample, similar to the restrictions on set-samples, since the time-sample must be long enough to mitigate the cold-start bias. For the traces used in this dissertation, a 1-megabyte cache needs samples of 1-10 million instructions. The required sample length increases by about a factor of eight for each cache size quadrupling.

When time sampling eliminates the cold-start bias, enough samples will ensure accurate results. Many samples are needed to capture adequate portions of each execution phase, even 100 or more. This is a disadvantage as compared to set sampling since single set-samples automatically capture pieces of each execution phase and their corresponding cache performance variations. For the traces used in this dissertation, the number of samples should be increased by about a factor of three as the sample size is reduced by a factor of ten. Since time sampling requires so many large samples, it is difficult to meet the 10% sampling goal. Furthermore, the required fraction of the trace data increases with cache size, which suggests time sampling will be even less desirable with larger future caches.

Any full trace is only a time-sample, so time sampling techniques are always useful. Time samples retain the timing of the memory references; time-dependent behavior and the interactions among different cache sets can be modeled. Time sampling also may conform to the limitations of existing trace gathering mechanisms. For example, the ATUM trace gathering mechanism produces short time-samples because of finite trace buffers [AGSH86].

4.4. Conclusions

The examination of multi-megabyte caches requires large amounts of trace data. Long traces overcome cold start and provide the most information about a workload's cache performance, but finite resources make them difficult to obtain and use. Trace sampling can reduce the computation and storage requirements of trace-driven simulation by an order of magnitude. It can give the most accurate estimate of true mean cache performance because more cache behavior from more workloads can be examined with a fixed tracing budget. This dissertation chapter is the first comparison of set sampling and time sampling, two previously proposed trace-sampling techniques.

An important aspect of any sample is how flexible it is, or the range of cache configurations for which it can be used. Both set samples and time samples have usage restrictions. A constant-bits method that produces set samples for the largest range of hierarchical cache configurations was introduced in this chapter; it selects references based on the values of their address bits (constant bits). With a single set-sample, the explorable cache design space is well defined: set samples are useful when the cache set-indexing bits contain the constant bits. The restrictions on time-sample usage are a little more vague: time samples are useful when the cold-start bias can be removed. Under the widest range of conditions, the initialization reference miss ratio estimation of INITMR gives superior cold-start bias reduction. INITMR gives accurate results when the cache is at least 50% initialized by the sample, and when there are at least as many known misses as initialization references. These restrictions require long time-samples. For the workloads examined in this dissertation, a 1-megabyte cache needs time samples of up to 10 million instructions, and a 16-megabyte cache needs time samples of 100 million instructions or more.

For the traces used in this dissertation, set sampling produces better mean *MPI* estimates with less trace data than time sampling; set sampling met the 10% sampling goal of less than 10% errors with less than 10% of the full trace data. Provided sample usage restrictions can be tolerated, set sampling is better than time sampling when estimating mean cache *MPI*. One reason for its success is that set samples include references from many execution phases, rather than just one or a few like a single time-sample; a single set-sample retains much of the phase-dependent cache behavior, while many time-samples are needed to capture the same effect. Another reason for the relative success of set sampling is the time-sampling cold-start problem, which dictates the minimum time-sample length, particularly

for large caches. In this study, time sampling could not meet the 10% sampling goal for the largest caches. Time samples may only be more useful when the timing of references or the interactions among sets is important.

This chapter simulates a variety of multi-megabyte secondary caches of different size and associativity for several different traces. The comparison shows that cache size and associativity have an important effect on the set-sampling vs. time-sampling comparison. Both size and associativity increases make set sampling more desirable: size increases because time sampling requires more trace data as cache sizes increase (about 2/3 more for every cache size doubling), and associativity increases because the conflict misses from individual sets are less likely to bias individual sample results. These trends make set sampling extremely useful for multi-megabyte cache analysis, and even more useful for the ever larger caches of the future.

Chapter 5

Memory System Design With Multi-Megabyte Caches

5.1. Introduction

Whereas many other cache design studies focus on smaller caches, this chapter focuses on multi-megabyte cache design. Using traces of large workloads, this chapter shows that high-performance memory systems with multi-megabyte caches can greatly improve performance beyond what is available with smaller caches. A multi-megabyte cache at the bottom of a two-level CPU cache hierarchy reduces the main memory access frequency required to support processor references, and consequently eliminates most of the performance loss due to cache misses, even when each miss is very expensive. Together with faster primary caches that service most references, multi-megabyte secondary caches can allow even the fastest processors to execute near their maximum speeds.

This chapter finds many similarities between the design of multi-megabyte caches and smaller caches, as well as many differences. For multi-megabyte caches, the results of this chapter confirm that 2-way set-associativity gives much of the performance benefits of higher associativities, which was well-known for smaller caches [HILS89]. As a striking difference, the results also show that the performance improvement from doubling associativity is smaller than the improvement from doubling the size of a direct-mapped cache; this contradicts Patterson and Hennessy's 2:1 rule of thumb for smaller caches [HENP90]. This chapter finds that set-associativity is most necessary in a secondary cache when the cache is not much larger than its primary caches. It also shows that direct-mapping performs so well with the largest caches (e.g., caches that are 32 or more times the primary cache size) that an increase in associativity beyond direct-mapped could worsen performance; this extends the work of Hill [HILL87, HILL88] and Przybylski [PRZY88, PRHH89], who showed that direct-mapping outperforms

higher associativities in large primary caches (e.g., greater than 64-kilobyte). This chapter shows that non-random replacement policies eliminate many cache misses in some workloads, but few in others, so random replacement may still be preferred because it is easy to implement. This chapter also finds that equalizing the memory access and transfer time latency components usually gives a near-optimal block size, a result that Przybylski [PRZY90] and Smith [SMIT87] showed for smaller caches. Finally, this chapter discusses the implementation of hardware multi-level inclusion in a uniprocessor system.

Trace-driven simulation results from the traces described in Chapter 3 form the basis for the analysis of this chapter. Section 5.2 examines the need for multi-megabyte caches, emphasizing the reduced memory traffic that they allow. Section 5.3 motivates the hierarchical cache configuration that is assumed throughout this dissertation, the one depicted on the far right in Figure 1.1. (Section 2.2 gives definitions and default parameters of caches.) This chapter uses the *SCPI* cache performance metric, introduced in Section 2.4, because it is important to add timing to the *MPI* estimates obtained from the previous chapters when comparing alternative cache designs. Section 5.4 discusses primary cache design considerations. Section 5.5 shows the affect that different cache sizes can have on the processor stall time, pointing out the affect of main memory speeds, workload choice, and cache speed degradation on preferred cache size. Section 5.6 considers alternative secondary cache associativities and Section 5.7 considers different cache block sizes. Section 5.8 considers the implementation of multi-level inclusion. Section 5.9 summarizes the results of this chapter.

5.2. Importance of Multi-Megabyte Caches

With faster processors and expanding main (DRAM) memory sizes, the design of a memory system to satisfy processor demands becomes a more difficult problem. Future processors will be able to execute 100 or more operations during the same time it takes to service a single main memory reference, so the slow main memory access time can dominate processing speeds unless main memory accesses are rare. Future main memories will grow to hundreds of megabytes and more, so it is inevitable that future applications will reference much more memory, especially since faster processors allow larger problems to be solved. This section shows that multi-megabyte caches require only very infrequent main memory accesses, which allows them to satisfy the challenging requirements of future computing environments. The next section shows how cache hierarchies can provide the fast access times of a smaller cache together with the low main memory access requirements of multi-megabyte caches.

For the traces described in Chapter 3, Figure 5.1 shows the ratio of the main memory traffic with a 4-megabyte cache divided by the main memory traffic without a cache inserted between the processor and the main memory. (When there is no cache, every instruction requires at least one main memory access.) The main memory traffic reduction of multi-megabyte caches has two components: access frequency reduction and bandwidth reduction. The access *frequency* is the number of times that a contiguous block of memory is either read or written (the secondary cache is write-back, so all main memory accesses either read or write large blocks). The access *bandwidth* is the number of data words that the main memory either reads or writes (excluding address and control information). Both traffic reduction components are important to the performance improvements obtained with multi-megabyte caches. Frequency reduction may be more essential than bandwidth because bulk data transfers are often considerably more efficient than smaller, more frequent, data transfers.

The results in Figure 5.1 show that the multi-megabyte cache greatly reduces access frequency for each trace; the access frequency with the multi-megabyte cache relative to the access frequency without

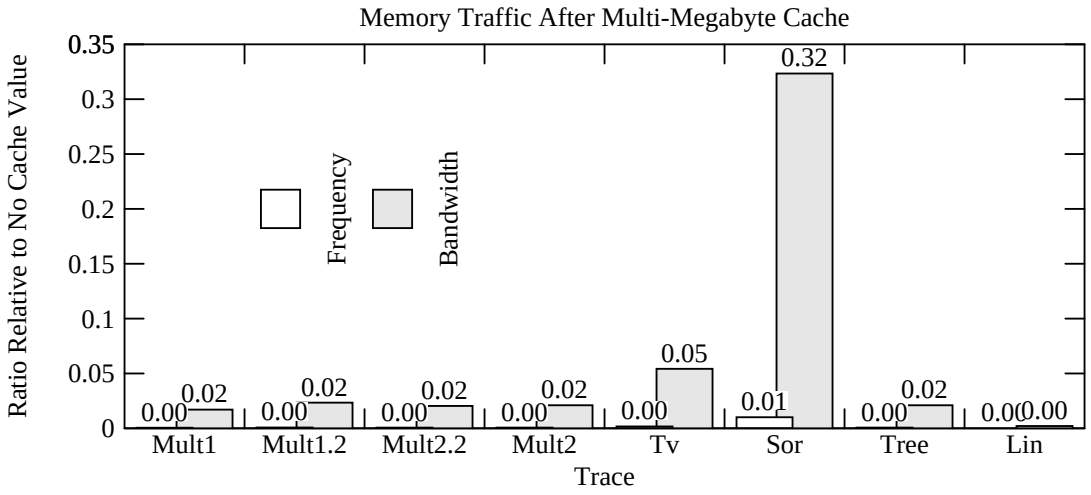


Figure 5.1. Memory Traffic Reduction With Multi-Megabyte Cache.

This figure shows the memory access frequency and bandwidth reductions with a 4-megabyte cache memory for the traces of this study. It shows the ratio of results relative to the values without a cache inserted between the processor and memory, including both instruction accesses and data accesses. The cache is direct-mapped with a block size of 128-bytes, and is write-back.

the cache is usually too small to see. The cache reduces bandwidth to a lesser extent because each main memory access occurs as the result of a cache miss, so each access involves the transfer of entire cache blocks, which are considerably larger than the single-word loads and stores required by the processor. Nevertheless, the 4-megabyte cache often gets a factor of 20-50 bandwidth reduction, and always reduces the access frequency by more than a factor of 100. The Sor trace gives traffic ratios that are considerably larger than the other traces. This shows the poor Sor cache performance, which will become more evident as this chapter progresses.

This is the first study to analyze multi-megabyte caches, perhaps because they have not become cost-effective and necessary until recently. Traffic reductions like those shown in Figure 5.1 are required when main memory access times are high enough and cannot be overlapped with other useful work. Smaller caches cannot achieve these traffic reductions with these large workloads. This chapter examines the design alternatives and requirements of these multi-megabyte caches.

5.3. Importance of the CPU Cache Hierarchy

Multi-megabyte caches reduce main memory traffic, but that alone is inadequate for a high-performance memory system. A multi-megabyte cache has a much faster access time than main memory, but it is not fast enough. Figure 5.2 shows the multi-level cache configuration introduced in Chapter 1 that solves this problem. This multi-level structuring solves the speed problems of the multi-megabyte cache because the faster primary CPU caches service most references, and the multi-megabyte secondary cache services only the misses in the primary caches. The primary instruction (ICACHE) and data (DCACHE) caches are split so that they can concurrently service both an instruction and data access, but both primary caches share the use of a unified multi-megabyte secondary cache (SCACHE). A main memory reference occurs only when there is a miss in both the primary and the secondary cache. This hierarchical structuring bypasses the speed limitations of multi-megabyte

caches, yet it retains their storage capacity (and infrequent misses). Of course, adding another level also increases the complexity of the design since the interactions between all elements of the memory system must be considered. The need for ever-faster memory accesses outweighs the disadvantages of increased complexity, however, and multiple levels of CPU caches are certain to be prevalent in many future designs.

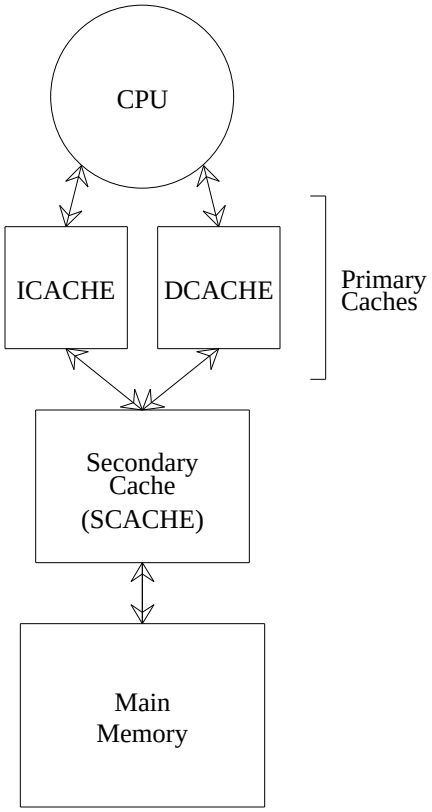


Figure 5.2. The Simulated Cache Configuration.

This figure represents the assumed cache configuration used throughout this dissertation. It consists of split primary (level 1) instruction (ICACHE) and data caches (DCACHE) backed up by a shared secondary (level 2) cache (SCACHE) and main memory. (It is the same configuration depicted on the far right in Figure 1.1).

Other recent studies focus on CPU cache hierarchies for bridging the CPU-main memory speed gap [BUKB90, PRHH89, SHOL88, WABL89]. While implementation technology demands a multi-level cache hierarchy, often with small primary caches and larger secondary caches, a cache hierarchy alone is not adequate to ensure good performance. Without a multi-megabyte secondary cache, a hierarchy will not sufficiently reduce main memory access frequency. This dissertation focuses on hierarchies including multi-megabyte secondary caches at the bottom.

A great many options become available to CPU cache designers when considering multiple-levels of caches, too many to be immediately examined. In practice, structural constraints often dictate portions of the design. For example, if the primary caches are constrained to reside on the processor chip, their size may be fixed for a given processor implementation. This study restricts the primary caches to 32-kilobytes each. This restriction is appropriate since a design using commercially-available processors, or any fixed-primary-cache processor design, would be similarly constrained. This study also only

considers direct-mapped primary caches, as advocated by Hill [HILL88] and Jouppi [JOUP89]. This dissertation further prunes the design space by considering only two-level hierarchical configurations. For the near future, two levels in the CPU cache hierarchy will be both practical and useful since they provide the advantages of a multi-level hierarchy, limit the complexity, and fit well with the physical design of computer systems.

The focus of this dissertation is on secondary multi-megabyte cache design, not primary cache design or hierarchy design. Though this dissertation considers only a limited range of the multi-level cache design space, many results also apply to many different hierarchies. Provided the secondary caches are large compared to the upstream caches, secondary cache performance is largely independent of the number of hierarchy levels or the particular primary cache configuration [PRZY88].

5.4. The Primary Caches

Table 5.1 shows local miss ratios (misses divided by references) and *SCPI* (stall cycles per instruction) components for the data and instruction caches of 32-kilobytes each with block sizes of 32-bytes. Section 2.4 of Chapter 2 discusses the advantages of the cache performance metrics *MPI* (misses per instruction) and *SCPI*. *SCPI* factors in both the cache reference frequency and timing so that the effect of cache misses is evident. The results for the different traces show that the miss ratio of the instruction cache is consistently smaller than the miss ratio of the data cache. However, since the processor uses the instruction cache more frequently than the data cache (loads and stores are only 40% of the instructions), the effect of the instruction cache misses on processor performance ($SCPI_{ICACHE}$) is considerably closer to that of the data cache ($SCPI_{DCACHE}$). This is one indication that (local) miss ratio can be a misleading indicator of the performance effects of a cache. Both *MPI* and *SCPI* correctly account for the cache access frequency and the fraction of accesses that cause misses, allowing the effect of misses on performance to be properly characterized. Miss ratio may obscure the cache access frequency and give a misleading characterization. Therefore, this dissertation uses *MPI* and *SCPI* rather than miss ratio. This chapter uses *SCPI* in many cases to compare alternative cache configurations because it includes the timing behavior of a cache, as well as the miss frequency. This is important because the timing may change with different cache configurations.

The data in Table 5.1 shows that the *SCPI* components resulting from the primary caches can reach a value of 0.2 although the cache sizes are 32-kilobytes each. While it is not the major focus of this thesis, primary cache design is important to the performance of the memory system hierarchy. For the Mult1.2 trace, Figure 5.3 emphasizes the contributions of each cache in the hierarchy by showing the fraction of *SCPI* that is a result of the misses from each component of the hierarchy. Clearly, a large portion of *SCPI* results from the primary caches. Their contribution to overall memory system performance should not be ignored.

Recognizing the importance of primary caches to memory system performance, several recent studies propose enhancements to primary cache design that allow processors to realize their high performance potential. For primary caches, particularly instruction caches, prefetching may be used to overlap the latency of secondary cache accesses with useful work. Smith [SMIT82] describes several prefetching options, some of which were examined by Hill [HILL87]. Farrens and Pleszkun [FARP89] advocate the use of queues that allow instructions to be fetched before they are used. Jouppi [JOUP90] advocates stream buffers that prefetch sequential locations after cache misses. Jouppi also introduces victim caches, small fully-associative buffers to eliminate conflict misses in direct-mapped caches, and

Local Miss Ratio of Primary Caches		
Trace	ICACHE	DCACHE
Mult1	0.0055	0.0202
Mult1.2	0.0058	0.0171
Mult2	0.0061	0.0196
Mult2.2	0.0055	0.0176
Tv	0.0026	0.0444
Sor	0.0000	0.0613
Tree	0.0049	0.0248
Lin	0.0000	0.0070

SCPI For Primary Caches		
Trace	$SCPI_{ICACHE}$	$SCPI_{DCACHE}$
Mult1	0.06	0.10
Mult1.2	0.06	0.08
Mult2	0.06	0.08
Mult2.2	0.06	0.07
Tv	0.03	0.17
Sor	0.00	0.23
Tree	0.05	0.12
Lin	0.00	0.04

Table 5.1. Primary Cache Performance.

This table shows the local miss ratios (top) and *SCPI* components (bottom) for the primary instruction and data caches ($Tmiss_{ICACHE} = Tmiss_{DCACHE} = 10$). (Section 2.4 of Chapter 2 defines *SCPI* and *Tmiss*.) The *MPI* of this configuration for these traces is *SCPI*/10 (and is also given in Figure 3.7 of Chapter 3).

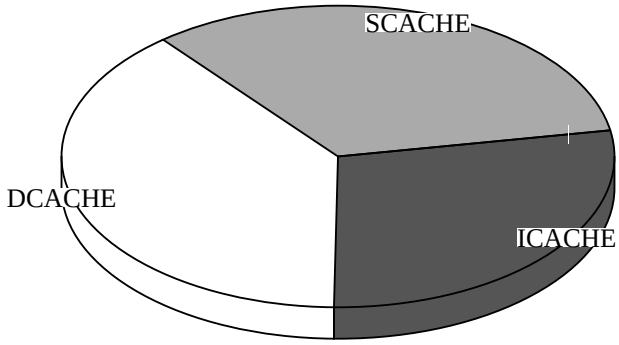


Figure 5.3. Portions of SCPI for Mult1.2.

This figure shows the division of *SCPI* of 0.21 into components due to instruction, data, and secondary cache misses for Mult1.2. The secondary cache is direct-mapped of size 4-megabytes with a block size of 128-bytes.

finds them particularly useful for data caches. Data cache design becomes more difficult and important as processors are fast enough to execute more than one instruction during the time it takes to access the cache memory. Sohi and Franklin [SOHF91] introduce a lockup-free [KROF81] primary data cache organization that meets processor data memory bandwidth requirements under such conditions. These studies show that primary cache design should not be neglected.

5.5. Secondary Cache Size Choice

Perhaps the most important secondary cache design consideration is size. The advantage of a larger cache is a lower *MPI*. Unfortunately, a larger cache also has a higher cost, requires more space, and has a slower access time than a smaller cache. This section examines these tradeoffs. Section 5.5.1 considers the effects of slower main memory access times, shows that a larger cache is desired, but does not factor in the implementation considerations of a larger cache. Section 5.5.2 shows that the desired cache size depends on the workloads executed on the machine. Finally, Section 5.5.3 factors in some implementation considerations of cache size increases, for a more balanced perspective of the performance implications of cache size increases.

5.5.1. Effect of Processor-Main Memory Speed Gap

As processor speeds increase relative to the access times of main (DRAM) memories, main memory accesses become more costly. If techniques cannot be developed to eliminate or overlap these large latencies, a high performance memory system must reduce the frequency of main memory accesses or they will dominate processor performance.

Figure 5.4 shows that the reduced *MPI* of multi-megabyte caches becomes more essential as the main memory access time increases. For the Mult1.2 trace, the figure shows the *SCPI* values for different cache sizes with *Tmiss* ranging from 30 to 200 cycles. The results make the optimistic assumption that the secondary cache access time is independent of cache size ($Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$ are constant, unlike Section 5.5.3). Note that *SCPI* does not approach zero with increasing cache size because the primary cache misses also contribute a portion of the *SCPI*. (Table 5.1 shows that $SCPI_{ICACHE}$ and $SCPI_{DCACHE}$ sum to 0.14 for Mult1.2.)

The magnitude of $Tmiss_{SCACHE}$ is crucial to the secondary cache size choice. While a system with a 128-kilobyte secondary cache might provide a sufficiently low *SCPI* with $Tmiss_{SCACHE} = 30$, cache sizes of 1-megabyte or more are required for similar performance with the higher main memory access times. For example, as the dotted lines in Figure 5.4 show, more than a 1-megabyte cache is needed for the same *SCPI* as a 128-kilobyte cache as $Tmiss_{SCACHE}$ increases from 30 to 100 cycles. This is more than a factor of eight size increase. Similarly, the cache size must be at least 4-megabytes for the same *SCPI* when $Tmiss_{SCACHE}$ increases to 200 cycles.

$Tmiss_{SCACHE}$ consists of several components. The first portion, and perhaps the largest, is the fixed latency of a memory access. This is the part that is independent of the amount of data transferred (the block size). It includes the time to send the address to the main memory, cycle the memory array, and return the first portion of the data. The second portion of $Tmiss_{SCACHE}$ is the time required to transfer the remaining portion of the cache block, which depends on the bandwidth of the cache-main memory interconnect and the block size. Queueing delay also could be a large portion of the main memory access time. The combination of these factors can lead to delays of hundreds of cycles for fast processors. $Tmiss_{SCACHE}$ could easily be 100 cycles when each cycle is five nanoseconds or less. In multiprocessors, the cache miss latency only gets worse.

The secondary cache needs to be large enough to make *SCPI* small. An *SCPI* of 0.5 will slow a processor that executes a single instruction per cycle, and it will devastate the performance of a processor that executes multiple instructions per cycle. That is, the memory system becomes the performance bottleneck as processor speeds increase, so a small *SCPI* reduction can greatly improve processor performance.

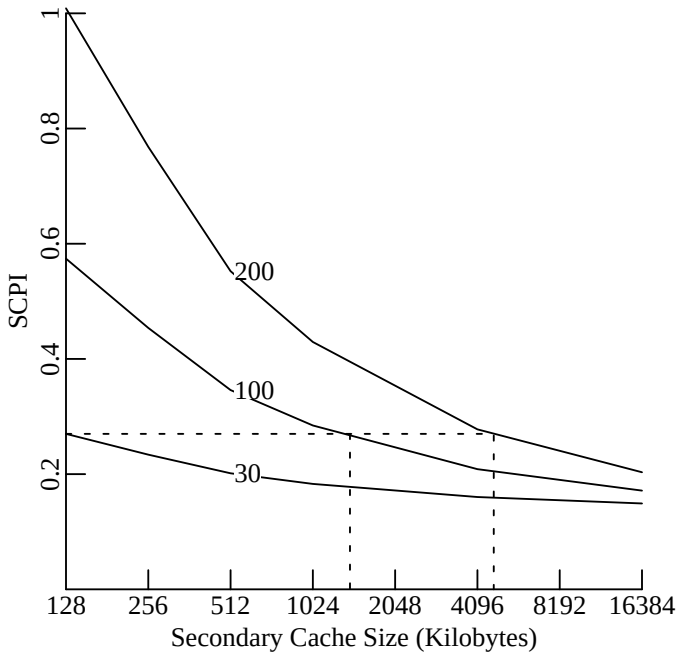


Figure 5.4. Effect of Miss Penalty on SCPI for Mult1.2.

This figure shows the Stall Cycles Per Instruction (*SCPI*) versus the secondary cache size for the Mult1.2 trace With various main memory access times ($Tmiss_{SCACHE}$ = 30, 100, and 200 Cycles). The direct-mapped secondary caches have block sizes of 128-bytes.

5.5.2. The Effect of Workload Choice

The type of programs for which a memory system is being designed have a large effect on its design. If working set sizes are no more than 512-kilobytes, a 512-kilobyte cache may be more than enough to provide good performance. On the other hand, a program that references large amounts of memory with poor locality will perform poorly with a small cache. Since the traced workloads used in this dissertation reference large amounts of memory, this study favors larger caches. This bias was intentional since the appearance of larger main memories (100-megabytes and more) is expected to bring larger workloads. As more resources become available, users naturally adjust to utilize them, both by increasing the size of previous applications and by creating new applications.

Figure 5.5 shows the difference in performance for the different workloads captured in the traces. There are large differences in *SCPI* that result from the various memory reference behaviors included in the trace data. The Sor trace has the worst performance, its *SCPI* is much higher than for the rest of the workloads. Sor operates on representations of large, sparse, matrices. This leads to large locality sets. It is widely accepted that caches perform poorly with scientific workloads. A counter-example to this generalization is Lin. Lin is another scientific program that uses sparse matrices, yet gives good cache performance because it has much higher locality. This shows that scientific workloads are not always hard on caches; *SCPI* depends on the locality of the workload.

One could argue that the Sor workload is not an average workload, and one might even argue that its *SCPI* is pathological. Even so, the Sor results point out one important fact: when choosing a secondary cache size, the applications that can efficiently execute on the system are also being chosen. The Sor workload would perform poorly on a system with only a 1-megabyte secondary cache (and the 100

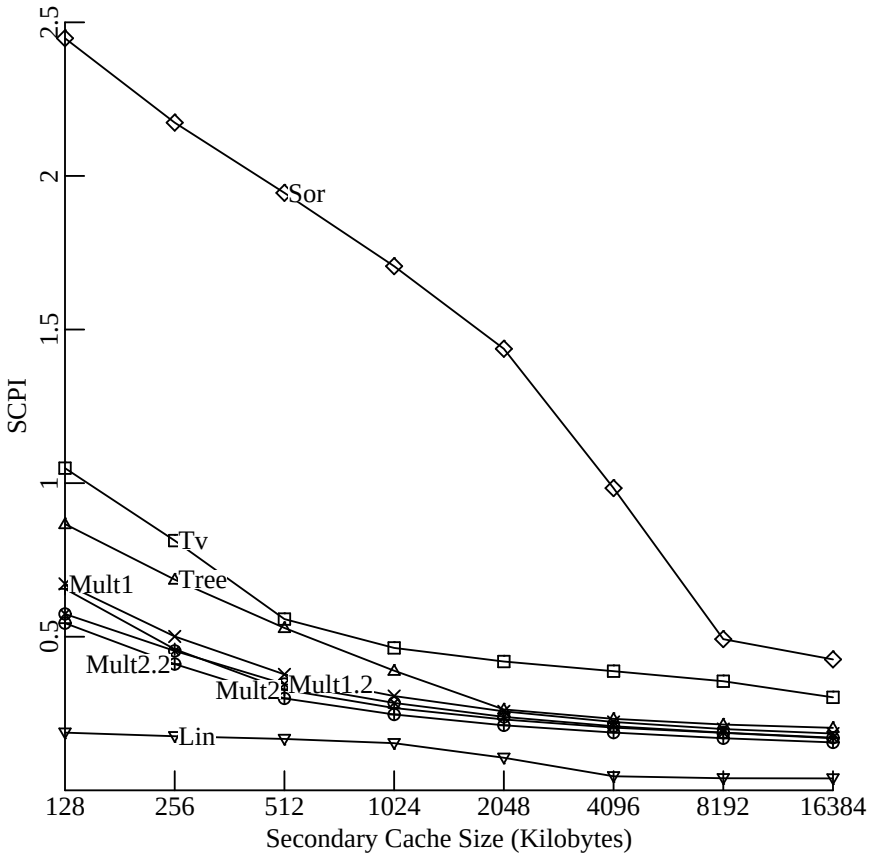


Figure 5.5. Effect of Workload on Cache Size Choice.

This figure shows the Stall Cycles Per Instruction (*SCPI*) versus the secondary cache size of the different traces considered in this study. The secondary caches are direct-mapped with block sizes of 128-bytes and $Tmiss_{SCACHE} = 100$.

cycle secondary miss penalty), while the multiprogrammed workloads perform well. If scientific applications like Sor are expected, a larger cache size might be preferred.

Figure 5.5 shows that multiprogramming does not necessarily lead to poor cache performance. The multiprogrammed traces give the second lowest *SCPI* overall, below the uniprogrammed Sor, Tv, and Tree *SCPI* results. The *SCPI* of the multiprogrammed traces are almost the same, probably because of the similarity of the traced workloads. Mult1.2 (Mult2.2) has a slightly lower *SCPI* than does Mult1 (Mult2), showing the well known result that cache efficiency improves with increasing process switch interval [SMIT82]. This is fortunate since switch intervals are likely to increase with the faster processors of the future. However, with hundreds of thousands of instructions executed between process switches, the locality of the workload is a more important determinant of cache performance than the process switch interval. Mult1 (Mult1.2) has a higher multiprogramming level and more memory active at any time than Mult2 (Mult2.2), which gives it slightly higher *SCPI* values.

5.5.3. Effect of Speed Degradation With Larger Caches

Not surprisingly, since there was no penalty to increasing the cache size, the results in Figure 5.5 shows that *SCPI* is minimized with larger caches. But larger caches tend to be slower than smaller caches because they have a wider fanout and fanin. This speed degradation of larger caches can be

factored into the *SCPI* model by increasing the access time of the larger secondary caches. This section starts by considering the *SCPI* with different caches if there is a 10% access time increase for each doubling of the cache size. Figure 5.6 shows *SCPI* versus the cache size with the 10% degradation per per secondary cache size doubling. The results are similar to Figure 5.5, except that the *SCPI* increases with larger caches.

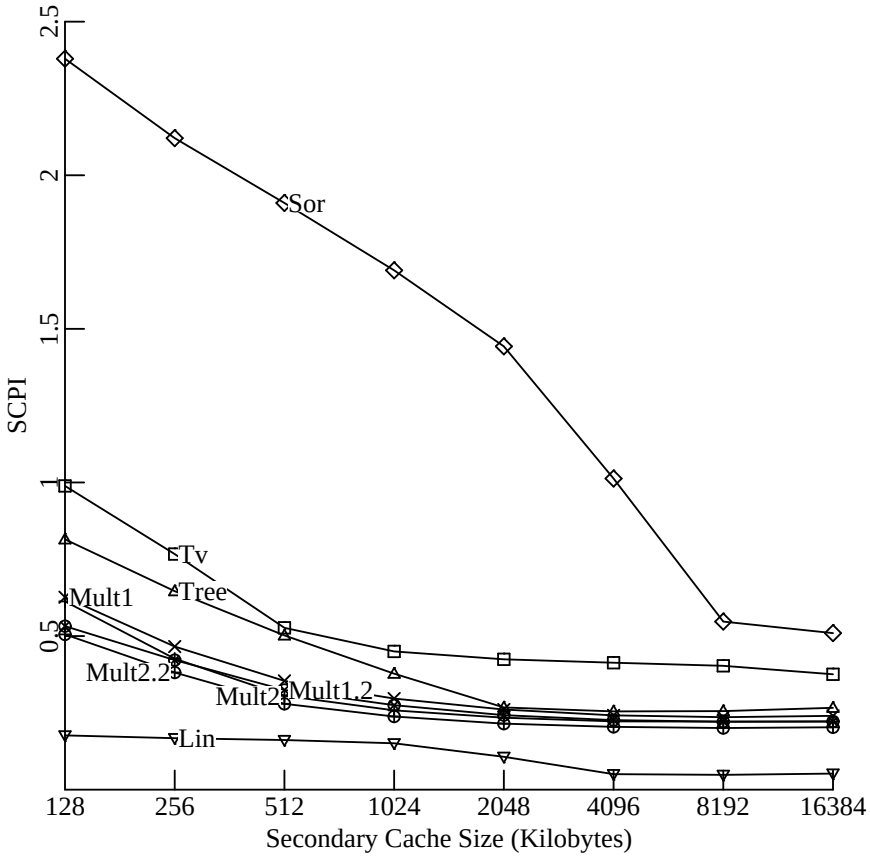


Figure 5.6. Effect of 10% Speed Degradation Per Cache Size Doubling.

This figure shows the Stall Cycles Per Instruction (*SCPI*) versus the secondary cache size of the different traces considered in this study. The secondary caches are direct-mapped with block sizes of 128-bytes and $Tmiss_{SCACHE} = 100$. The latency of a secondary cache access increases by 10% with each doubling of the secondary cache size ($Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$ are eight cycles for the 128-kilobyte cache and increase by 10% each time the cache size is doubled).

In Figure 5.6, many traces show *SCPI* reaching its minimal value at a cache size below 16-megabytes, usually 8-megabytes. With a 10% degradation in secondary cache access time per size doubling, a 16-megabyte cache would probably not be preferred over an 8-megabyte cache. The 8-megabyte cache does not minimize *SCPI* for all the traces, however. Tv and Sor have monotonically decreasing *SCPI* values with increasing cache size, even with the 10% speed degradation. For those two workloads, the advantage of the reduced *MPI* always outweighs the slower access time of larger secondary caches (up to 16-megabytes).

An alternative way to examine the access time and miss reduction tradeoffs is to find the maximum access time penalty that can be tolerated in a larger cache while keeping *SCPI* constant. Using the parameters of the *SCPI* model, the question is: by how much can $Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$

increase as the cache size is doubled (while $SCPI_{SCACHE}$ decreases) so that *SCPI* does not increase as a result? This is the break-even access time penalty, $T_{breakeven}$. (Przybylski also uses a breakeven analysis [PRZY88].) Of course, for an implementation of the larger cache size to be preferable, the actual implementation time penalty should be considerably smaller than $T_{breakeven}$ so that *SCPI* is smaller with the larger cache. $T_{breakeven}$ is a useful measure showing the access time tolerance as the cache size is doubled.

The break-even access time penalty is simply calculated using the formula for *SCPI*. The higher secondary cache access times can be equated to the fewer main memory access times as in the following equation:

$$(MPI_{ICACHE} + MPI_{DCACHE})T_{breakeven} = Tmiss_{SCACHE}\Delta MPI_{SCACHE} \quad (5.1)$$

where $T_{breakeven}$ is the break-even access time penalty and ΔMPI_{SCACHE} is the decrease in MPI_{SCACHE} with the larger cache size. This can then be solved for $T_{breakeven}$. Note that this calculation assumes that $Tmiss_{SCACHE}$ is not adversely affected by the cache speed slowdown, only $Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$ increase. That is, the latency of a secondary cache access increases, but the latency of a secondary cache miss does not with a cache size doubling.

Figure 5.7 shows $T_{breakeven}$ for different secondary cache sizes with the different traces (as a percent of the normal secondary cache access time of ten cycles). Some break-even penalties are more than 100%, indicating that the secondary cache access time could be doubled to twenty cycles when a cache size doubling and the *SCPI* would still be reduced. This suggests a performance improvement for larger cache sizes, even if the access time of the secondary cache is substantially worsened.

For most of the traces, $T_{breakeven}$ decreases with increasing cache size. This shows the decreasing need for larger caches as the cache size is increased. As $SCPI_{SCACHE}$ becomes a smaller and smaller portion of the total *SCPI*, *SCPI* shrinks by a smaller amount when doubling the cache size. Then, the smaller *MPI* reduction cannot compensate as much for a slower access time.

For the Tv, Sor, and Lin traces, $T_{breakeven}$ does not monotonically decrease with cache size. Intermediate cache sizes instead maximize $T_{breakeven}$, and $T_{breakeven}$ is lower for the smaller and larger cache sizes. This abnormal behavior is because ΔMPI_{SCACHE} is not constant. ΔMPI_{SCACHE} is maximized at the intermediate-sized caches, while the smaller and larger caches have lower ΔMPI_{SCACHE} values. At the intermediate cache sizes, MPI_{SCACHE} drops precipitously because the cache suddenly holds the working set, and so $T_{breakeven}$ is large.

Figure 5.7 shows that, particularly for the smaller cache sizes, $T_{breakeven}$ is slightly smaller for the multiprogrammed workloads with longer process switch intervals. This shows that a longer interval reduces the need for a larger secondary cache.

5.6. Secondary Cache Associativity Alternatives

Another important secondary cache design consideration is the associativity of the cache, or the number of locations where a given block can reside in the cache. This section examines the effect of different associativity implementations. The structure of this section is much like the previous one (5.5). Section 5.6.1 shows the *SCPI* improvement produced by associativity if there is no access time implementation penalty of increased associativity. Section 5.6.2 shows the *MPI* improvement of sophisticated replacement policies. Section 5.6.3 brings in the timing considerations of associativity increases. Finally, Section 5.6.4 considers inexpensive implementations of set-associativity.

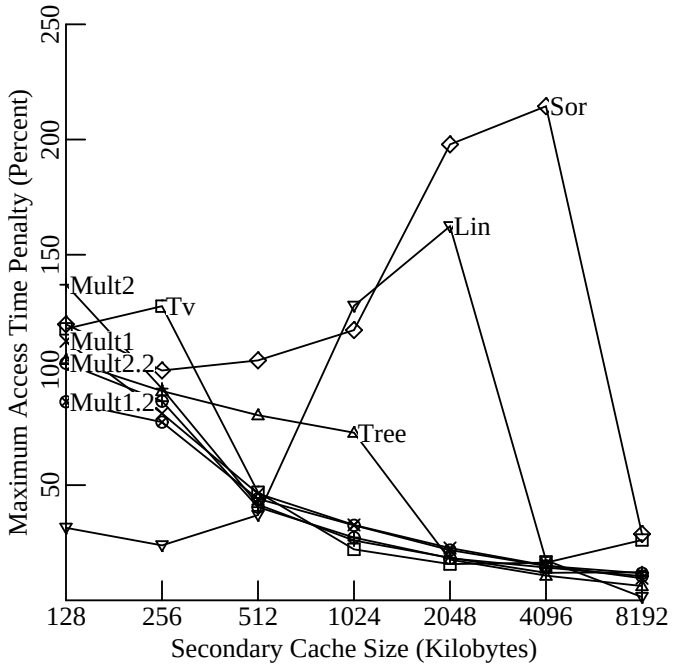


Figure 5.7. Break-Even Access Time Penalties for Cache Size Doubling.

This figure shows break-even access time penalties (as a percent of the secondary cache access time) for doubling the cache size. This is the maximum percentage that $Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$ can be increased beyond ten cycles when the cache size is doubled that does not allow *SCPI* to increase ($Tmiss_{SCACHE} = 100$). A value of 100% implies a cache of twice the size and a twenty cycle access time will give the same *SCPI*. The secondary caches are direct-mapped with block sizes of 128-bytes.

5.6.1. The Usefulness of Associativity

There is more mapping flexibility with a set-associative cache than with a direct-mapped cache. This flexibility provides the motivation for set-associativity: a lower *MPI*. The motivation is strong for secondary caches. Primary caches satisfy most of the processor references, but the rest must be serviced by the secondary cache. This makes the (local) miss ratio of secondary caches much higher than a primary cache of the same size [KEJL89, PRHH89]. Higher secondary cache associativity can satisfy a larger portion of the primary cache misses. Since the major task of a secondary cache is to reduce the *MPI*, it may seem that higher associativity is clearly advantageous. However, the potential disadvantage of increased associativity is a slower access time, and probably more hardware, since more cache block frames must be searched on a given access. A slower secondary cache access time may only be affordable because the secondary caches are accessed less frequently than the primary caches.

To consider the potential of associativity, Figure 5.8 plots SCPI versus the cache associativity for a range of secondary cache sizes with the Mult1.2 workload. These results make the optimistic assumption that associativity and cache size changes cause no increase in secondary cache access time (Section 5.6.3 factors in some implementation timing considerations); the only changing parameter in these results is the *MPI*. As expected, higher associativities reduce the *MPI* and thus reduce the *SCPI*.

The results in Figure 5.8 show that, for the Mult1.2 trace, associativity substantially reduces the *SCPI* of the smaller caches, but, is less useful for the larger caches. The higher *MPI* in the smaller caches leaves much opportunity for associativity miss elimination. With the larger caches, however, associativity is less needed since direct-mapped caches already perform well. This conclusion is

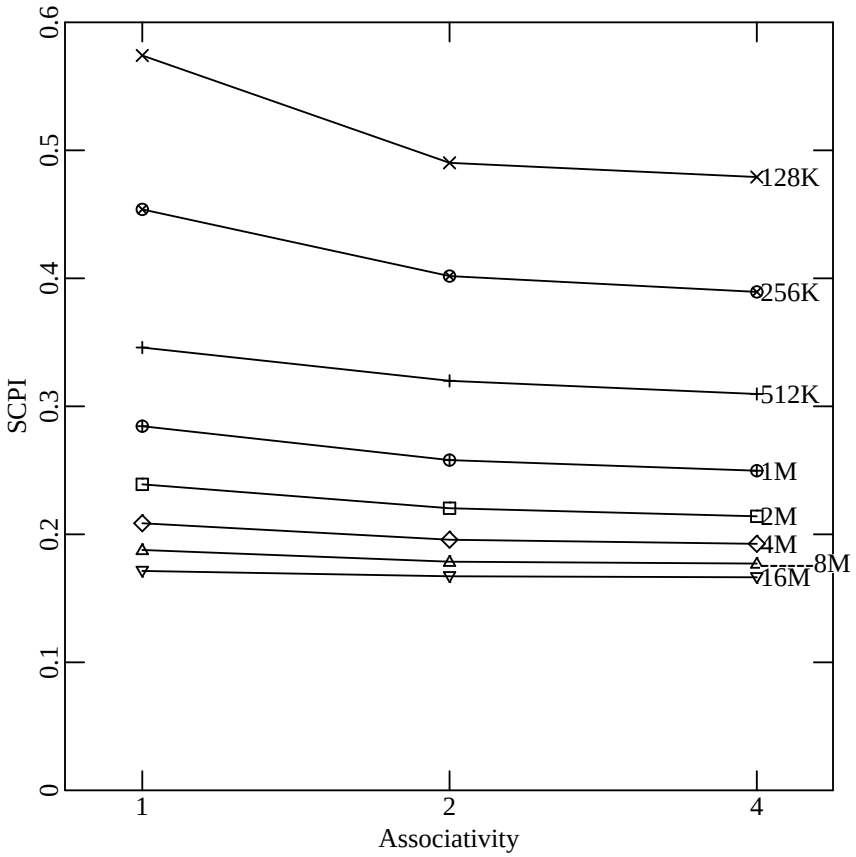


Figure 5.8. Performance Over a Range of Associativities.

This figure shows the Stall Cycles Per Instruction (*SCPI*) versus cache associativity for the Mult1.2 trace over a range of cache sizes. The block size of the secondary caches is 128-bytes. The results make the optimistic assumption that associativity and cache size changes do not increase secondary cache access time.

consistent with several studies of non-hierarchical configurations [HILL88, PRHH88]: once the (local) miss ratio becomes sufficiently low, there is not much performance gain from higher associativity. While associativity is worth considerable design effort for the smaller caches, with a larger cache the effort is less necessary (for this trace and these parameters). Were the main memory access time or the *MPI* higher, however, a different decision might be reached.

The 2:1 ‘‘rule of thumb’’ says that a cache size doubling and an associativity doubling produce about the same direct-mapped cache performance improvement [HENP90]. The Mult1.2 results in Figure 5.8 (and Chapter 3) are a counter-example to this rule. They show that a cache size doubling changes the direct-mapped *SCPI* by about 50% more than an associativity doubling does. The key reason for this contradiction is that multi-megabyte caches index to thousands of sets with little contention. Virtual-indexing spreads memory locations evenly across the cache, which makes associativity less beneficial and necessary. Chapter 6 introduces software techniques that make a real-indexed cache perform like a virtual-indexed one, in which case a real-indexed cache also violates the 2:1 rule. Multi-megabyte caches do not follow the 2:1 rule because they have little cache contention and, consequently, less need for associativity. This is different from the results of previous studies focusing on smaller designs [HENP90].

For all traces, Table 5.2 shows the *SCPI* for 1-megabyte caches of varying associativities. The other traces have similar associativity improvements to those for Mult1.2, though there are large variations for the different workloads. Again, the Sor trace stands out. It gives a considerably higher *SCPI* which does not decrease with associativity because of its looping references.

SCPI For Different Traces

Trace	Secondary Cache Associativity		
	Direct-Mapped	2-Way	4-Way
Mult1	0.31	0.27	0.26
Mult1.2	0.28	0.26	0.25
Mult2	0.27	0.24	0.24
Mult2.2	0.25	0.23	0.22
Tv	0.46	0.43	0.43
Sor	1.71	1.69	1.68
Tree	0.39	0.35	0.35

Table 5.2. Effects of Associativity on Cache Performance.

This table shows *SCPI* values for 1-megabyte secondary caches. It holds all parameters other than cache associativity constant.

The 2-way SCPI results are about 10% better than the direct-mapped *SCPI* results, given the optimistic implementation assumptions. For the 1-megabyte cache with random replacement, an increase to 4-way set-associativity does not substantially reduce *SCPI*. This illustrates an important point that has also been observed for smaller caches [HILS89]: 2-way set-associativity eliminates many direct-mapped cache conflicts. An associativity increase beyond 2-way provides only modest reductions in the cache *MPI*. The next section examines whether a more sophisticated replacement policy can further improve the performance of set-associative caches.

5.6.2. Replacement Policy Effects

Associativity allows some freedom when choosing where a block will reside in the cache. With a direct-mapped cache, each block can reside in exactly one frame. With higher associativities, any of the block frames in the set can hold the block. The added flexibility of associativity thrusts more responsibility on the cache decision making. On a miss, the cache must choose a frame to hold the block. Normally, the cache must replace another block from the set since all frames are used. The cache replacement policy chooses the block from the set that will be replaced.

Two common replacement policies are random and least-recently-used (LRU) [HENP90]. This dissertation uses random replacement because it is simple and requires no state information. Random chooses a random block in the set for replacement. LRU, on the other hand, chooses the block whose last reference was furthest in the past. LRU requires information about the past references to the set to make its replacement decision. Since past reference patterns are good predictors of the future, LRU can significantly reduce the *MPI*, particularly with workloads that have good locality of reference. However, the need for storing and updating the LRU information is a disadvantage.

With hierarchical cache configurations there are more replacement policy options than with a single level of caches. For example, the secondary cache replacement decisions can be based locally on the references to the secondary cache; alternatively, the primary cache references may also be a factor. The options are: should the LRUth block be the last block referenced (locally) in the secondary cache or

the last block referenced (globally) by the processor? Note that the two options are different because the primary caches filter out processor references before they reach the secondary cache. The first option (local) is easier to implement since only secondary cache references change the set LRU state. However, the second option (global) may lead to better replacement decisions.

This section compares four replacement policies that were chosen for their intuitive appeal and implementation simplicity. The first is the simple random policy. The second is a simple LRU policy based on the local secondary cache references. The third is the same LRU policy, except that write-backs (from the data cache to the secondary cache) do not affect the LRU ordering of a set. Intuitively, write-backs should not update the LRU information because they are only a result of primary cache replacement and are not directly a part of the processor reference stream. The final replacement policy considers information about primary cache contents. Inclusion-random replaces a random secondary cache block that is not held in the primary caches[20].

Table 5.3 shows the fraction of the random replacement misses that are eliminated by the final three replacement policies. The results show that the performance of LRU is largely independent of whether write-backs update the LRU information. While the *MPI*'s were improved slightly when write-backs did not update LRU information, the difference was not significant enough to appear in the results in Table 5.3.

LRU replacement gives a substantial *MPI* reduction for the multiprogrammed traces in Table 5.3. This is probably because the smaller multiprogrammed processes reference only a small portion of the cache, so LRU could hold a number of them in the cache at the same time. But the *MPI* improvement is modest for the uniprogrammed traces. In fact, LRU increases the Lin *MPI* compared to random. For the other (non-Lin) uniprogrammed traces, LRU has only a small advantage over random replacement. Overall, it is not clear whether it is worth the extra effort to implement LRU replacement in a 4-megabyte secondary cache. The choice depends on whether behavior like the multiprogrammed or uniprogrammed traces will predominate.

Table 5.3 shows that inclusion-random improves *MPI* slightly relative to random replacement. Though this shows that it is better not to replace secondary cache blocks that are (at least partially) held in the primary caches, inclusion-random requires considerable design complexity beyond simple random replacement, so it is probably not worthwhile for its *MPI* reduction alone. However, much of this complexity is already required with inclusion. The motivation for inclusion is more than simply *MPI* reduction. Section 5.8 discusses inclusion.

5.6.3. Effect of Speed Degradation With Associativity

The Figure 5.8 results did not have any increased access time effects of associativity. Increasing associativity may increase the cache access time [HILL88]. A speed degradation with higher associativity can be factored into the *SCPI* equations by increasing the secondary cache access time with increased associativity, just like Section 5.5.3 included the latency increase of a cache size doubling. Figure 5.9 shows the *SCPI* for different associativities with a 10% speed degradation for each associativity doubling. (Hill discusses some associativity implementations that increase the direct-mapped

20. If the primary cache holds all blocks in the set, the simulator chooses a random block for replacement. Note that since the primary cache block size is smaller than the secondary cache block size, the primary cache holds only a portion of blocks at any time.

Miss Reduction of Replacement Policies				
Trace	Assoc	Policy		
		LRU	LRUNO	INCRAND
Mult1	2	10%	10%	3%
	4	13%	13%	3%
Mult1.2	2	11%	11%	3%
	4	15%	15%	3%
Mult2	2	11%	11%	3%
	4	13%	13%	3%
Mult2.2	2	10%	10%	3%
	4	13%	13%	2%
Tv	2	2%	2%	0%
	4	2%	2%	0%
Sor	2	3%	3%	1%
	4	5%	5%	0%
Tree	2	5%	5%	2%
	4	6%	6%	1%
Lin	2	-21%	-21%	0%
	4	-10%	-10%	1%

Table 5.3. Improvement of Replacement Policies.

This table shows the miss reduction of several replacement policies for 2-way and 4-way 4-megabyte set-associative secondary caches. The miss reduction is the random replacement misses eliminated by the policy. LRU is a secondary-cache-local least-recently-used replacement policy. Some global LRU sample results were well under 1% different from the local results here. LRUNO is the same as LRU except write-backs do not adjust the LRU lists. INCRAND is random replacement except blocks with copies in the primary cache are not replaced.

access time by about 10% [HILL87].) The results are for 4-megabyte secondary caches.

Figure 5.9 shows that direct-mapped secondary caches gave the lowest *SCPI* for all traces except Mult1. The *MPI* reductions of associativity were not substantial enough to overcome the increased latency of the 10% degradation.

If a 10% degradation is too large, what degradation can be afforded? Similar to the break-even analysis for cache size doubling, another way to look at the usefulness of associativity is to calculate the access time implementation penalty that holds *SCPI* constant with increasing associativity. This is the break-even access time penalty, $T_{breakeven}$, which can be calculated using Equation 5.1. The question to be answered for a 2-way associative cache is: by how much can $Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$ be increased when the associativity is increased from direct-mapped to 2-way so that *SCPI* is not increased as a result? This gives the implementation leeway available to designers of a cache of higher associativity. An access time penalty lower than this value will give a lower higher-associativity *SCPI*.

Figure 5.10a shows the direct-mapped to 2-way $T_{breakeven}$ (in percent of the normal secondary cache access time of ten cycles) for different cache sizes. Figure 5.10b shows a blowup of the same results for cache sizes of 1-megabyte or more. Figure 5.10a shows the high break-even times that provide large window of opportunity for associativity in the smaller secondary caches. Since the 128-kilobyte cache is not a lot larger than the instruction and data caches of 32-kilobytes each, a higher associativity is preferable with the 128-kilobyte cache.

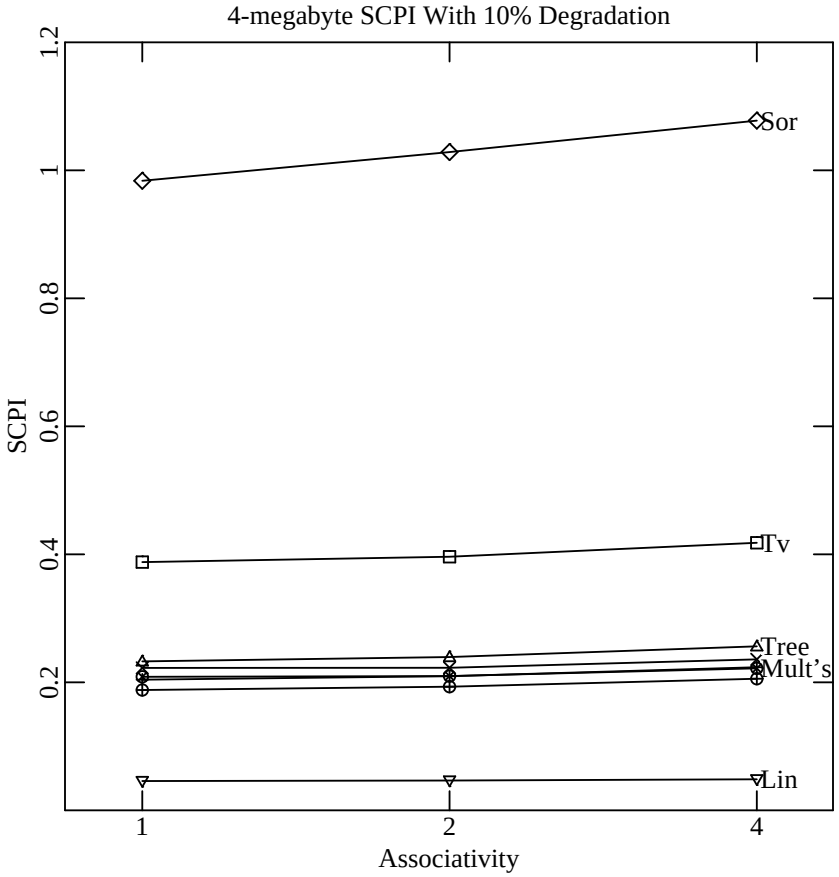


Figure 5.9. Associativity Performance With 10% Speed Degradation.

This figure shows the *SCPI* of 4-megabyte set-associative secondary caches with a 10% degradation in secondary cache access time ($Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$) for every doubling of associativity.

The break-even penalty vanishes with increasing secondary cache size. Direct-mapped caches provide good performance when the cache size is large compared to the size of the CPU cache(s) closer to the processor. Although associativity eliminates many misses, the *MPI* of the direct-mapped cache is already so low that the *SCPI* can only be decreased by a small amount. In effect, there are decreasing associativity returns with increasing scale (cache size). These results show that, for these parameters, it is probably not worth any extra design effort to implement a 16-megabyte 2-way set-associative cache rather than a direct-mapped cache of the same size.

The multiprogrammed traces have break-even curves that are smoothly decreasing with increasing secondary cache size in Figure 5.10a. Note that traces with the shorter process switch intervals (Mult1 and Mult2) tend to have larger break-even penalties, particularly for the smallest caches. This is a result of conflicts among the different processes using the cache. More frequent switching exacerbates the conflicting accesses of different processes. Associativity can reduce *MPI* in the presence of these conflicts.

Sor shows the most anomalous behavior in Figure 5.10a. It has negative break-even penalties in some cases because the *MPI* actually *increases* with associativity. This occurs because direct-mapped caches are the best for reference patterns that loop [SMIG85]. Tree shows smoothly decreasing break-

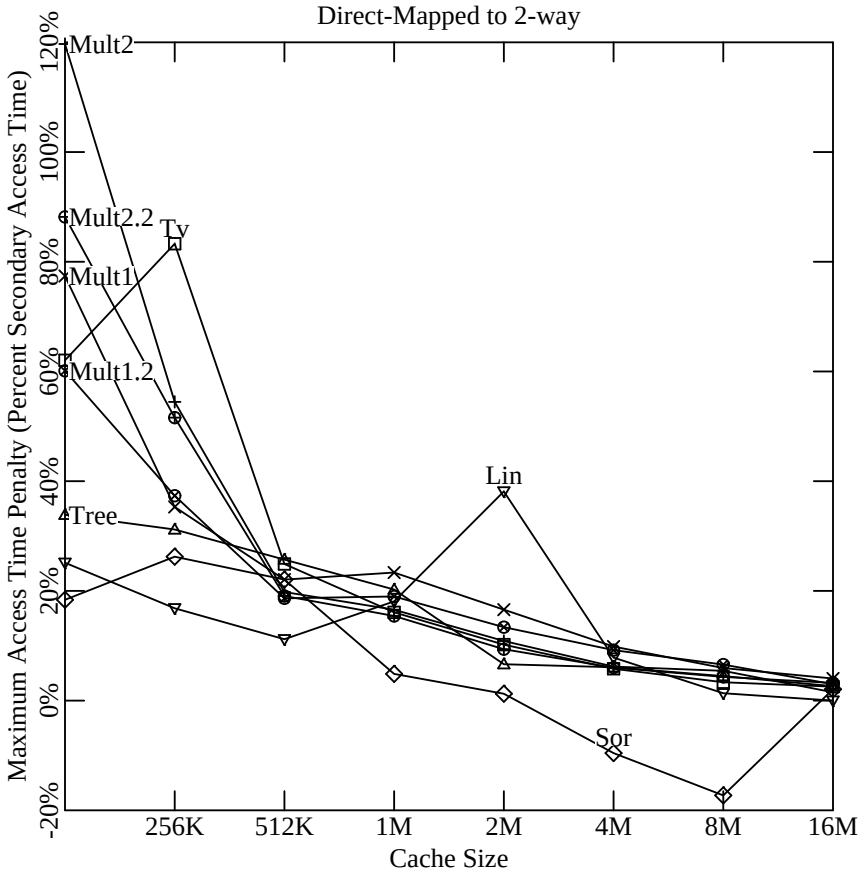


Figure 5.10a. Direct-Mapped to 2-Way Break-Even Implementation Penalties.

This figure shows the direct-mapped to 2-way break-even access time penalty ($T_{breakeven}$) for a range of cache sizes.

even penalties. Tv and Lin both have break-even penalties that are maximal at an intermediate cache size. This is abnormal and shows that associativity is most advantageous at the intermediate cache sizes, just like a cache size doubling was most advantageous with the intermediate-sized caches for these traces.

5.6.4. Inexpensive Associativity Implementations

Associativity can be expensive. Since any block in a set may be referenced on a secondary cache access, associativity requires a search. Traditional associativity implementations execute this search in parallel, as shown in Figure 2.2 and Figure 5.11: they read all cache tags of a set in parallel and compare them against the incoming tag. The required tag memory bandwidth and comparators are proportional to the associativity. Alternatively, inexpensive associativity implementations require a bandwidth of only a single tag [KEJL89]. Figure 5.11 depicts this option. Inexpensive implementations may be appropriate for secondary caches since secondary caches are accessed much less frequently than primary caches, and a slower access time may be tolerable. Inexpensive implementations provide the reduced *MPI* of set-associativity at the same cost as a direct-mapped cache. The slower access time of an inexpensive set-associativity implementation may be tolerable if the cache miss penalty is large or

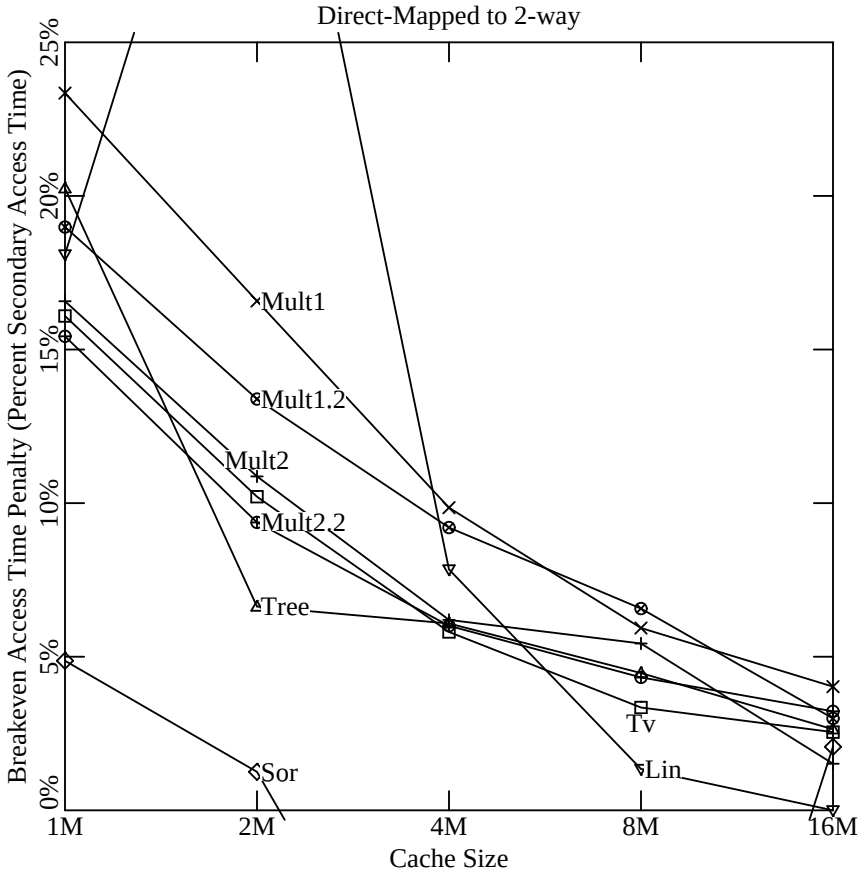


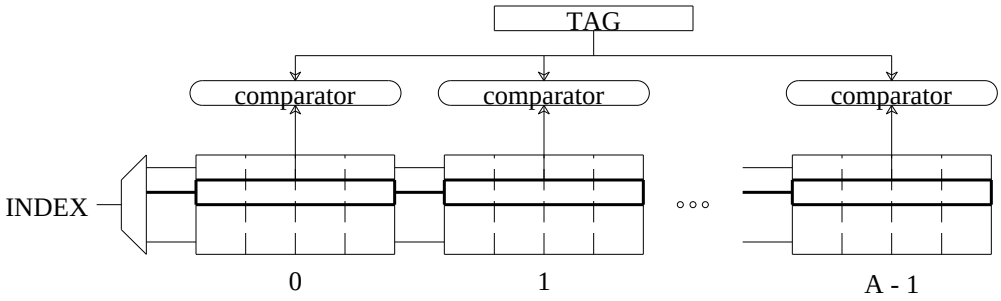
Figure 5.10b. Direct-Mapped to 2-Way Break-Even (Blow-up).

This figure shows a blowup of the results shown in Figure 5.10a.

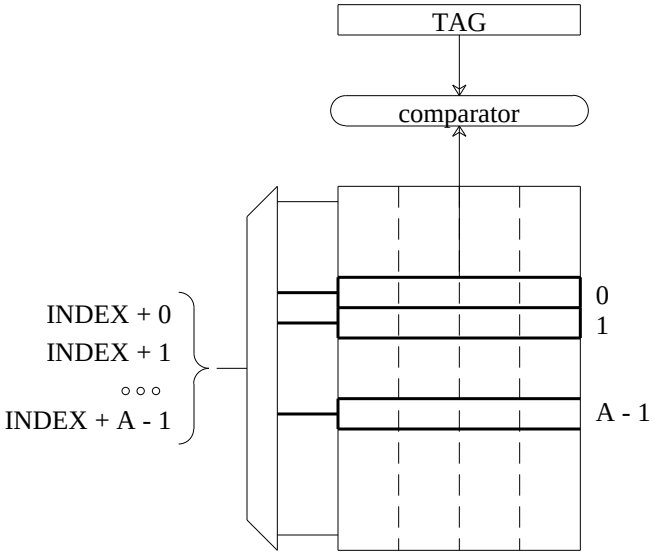
sensitive to increased load, and if a traditional set-associativity implementation cannot be afforded.

This section examines the three inexpensive implementation alternatives described by Kessler, et. al [KEJL89]. The first is a *naive* scheme that sequentially scans the tags in a set until it finds a match, starting at a random location. Figure 5.12 depicts two improved inexpensive associativity implementations. Chang, et al., [CHCS87] describe an implementation of the MRU scheme. It scans the tags from the most-recently-used to the least-recently-used. Of course, the MRU scheme requires information about the ordering of the blocks to complete its scan. This MRU information is similar (and perhaps the same) information required to implement the LRU replacement policy[21]. The final scheme is called *partial match*. Using a partitioning of the comparator bits with a tag memory addressing trick, it first compares pieces of each tag with the corresponding part of the incoming tag. In effect, the incoming tag is *partially compared* to each tag. When the partial compare does not match, the full compare will not match, so the tag need not be further examined. Each tag that partially matched is subsequently compared to the incoming tag to find if there is a full tag match. When a partial match does not give a full tag match, it is called a *false partial match*.

21. Note that the MRU information is local to the secondary cache, as with LRU in Section 5.6.2.



(a) Traditional



(b) Serial (Using Naive Approach)

Figure 5.11. Implementing Set-Associativity.

Part (a) of this figure (top) shows the traditional implementation of the logic to find a hit or miss in a *A*-way set-associative cache. This logic uses the set-indexing (‘‘INDEX’’) field of the reference to select one *T*-bit tag from each of *A* banks. It compares each stored tag to the incoming tag (‘‘TAG’’). It declares a hit when a stored tag matches the incoming tag, a miss otherwise.

Part (b) (bottom) shows a serial implementation of the same cache architecture. Here the implementation reads *a* stored tags in a set from one bank and compares them serially (the tags are addressed with "INDEX" concatenated with 0 through $A - 1$).

To compare these schemes, this section counts the number of *probes* required by each. A probe is a reading of the tag or MRU memories. Because the traditional associativity implementation has a high tag memory bandwidth, it requires only a single probe to decide a hit or miss. In the best case, a naive scan finds a hit in the first entry, so it needs only a single probe. In the worst case for a hit, or else for a miss, naive must probe through each tag to make the hit/miss determination. MRU requires at least two probes in any case since both the MRU and tag memories must be cycled: the MRU cycle selects the MRUth block, and then the tag memory cycle compares the tags. In the worst case for a hot, or for a miss, MRU must probe the MRU memory once (at least), and it also must probe the tag memory once for each tag. The partial match scheme requires at least two probes to find a match (one for the partial compare and one for the full comparison), but a miss can be determined with only a single partial

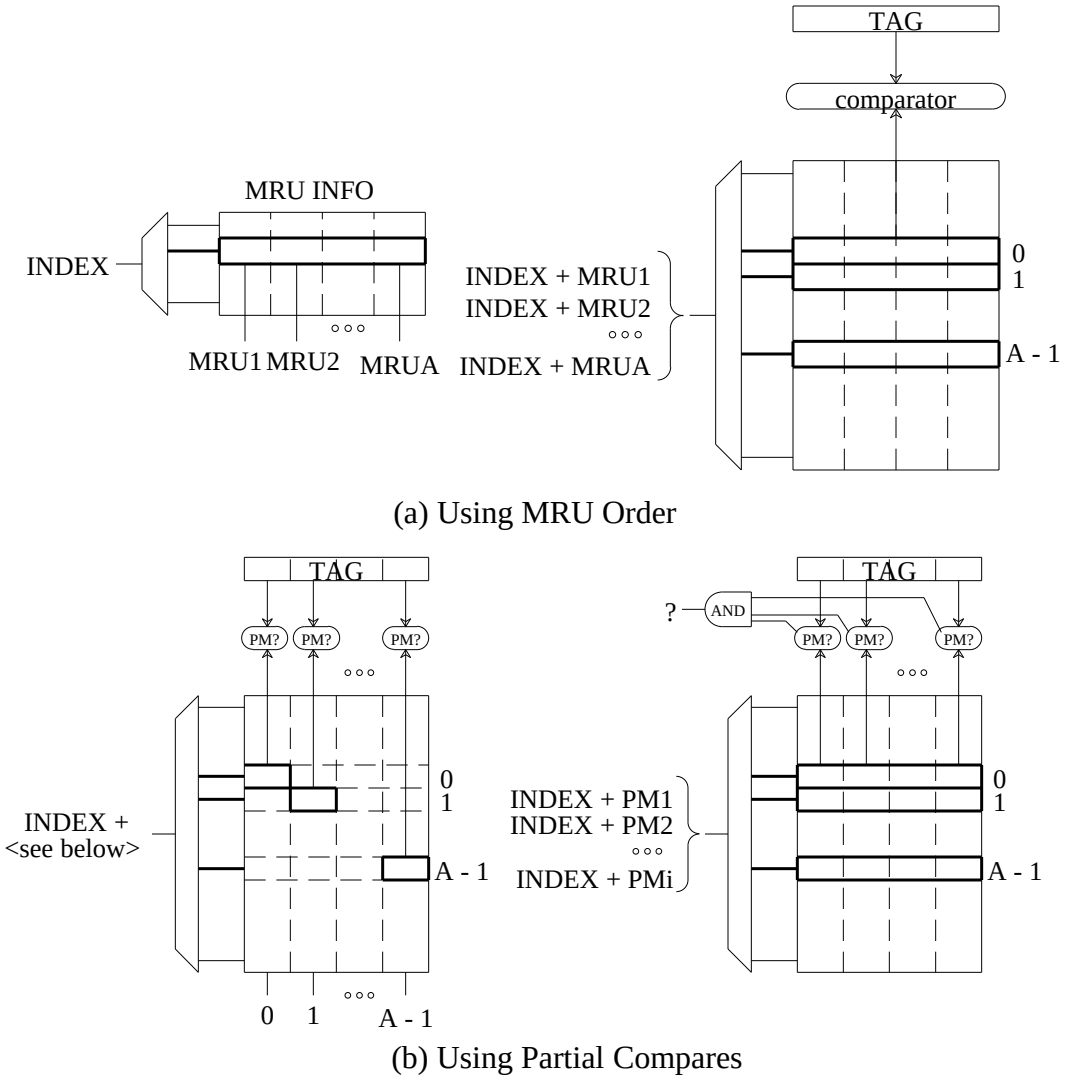


Figure 5.12. Improved Implementations of Serial Set-Associativity.

Part (a) of this figure (top) shows an implementation of serial set-associativity using ordering information. This approach first reads MRU ordering information (left) and then probes the stored tags from the one most-likely to match to the one least-likely to match (right). Note ‘‘+’’ represents concatenate.

Part (b) (bottom) shows an implementation of serial set-associativity using partial compares. This approach first reads K ($K = \lfloor T/A \rfloor$) bits from each stored tag and compares them with the corresponding bits of the incoming tag. In the second step, this approach serially compares all stored tags that partially matched (‘‘PM’’) with the incoming tag until it finds a match or it exhausts the tags (right).

compare of all tags, provided there are no false matches. This is a significant advantage of partial: it requires few probes to find a cache miss.

Table 5.4 shows the expected probes (assuming random tags) for the different schemes for the associativity A, and the tag memory width requirements. The entries show the speed superiority of the single probe required by the traditional associativity implementation, but it also shows the high tag and comparator width requirements needed to support this. For naive, the expected probes for a hit is the mean of the numbers 1, ..., A, since each tag is equally likely to hit. The MRU probes for hits depends entirely on the probability that the early entries in the MRU list are used. Since this probability is not

known in general, the table gives bounds for a 4-way set-associative example. The partial results assume random tags, that is, that the probability of a false partial match of *K*-bits is $\frac{1}{2^K}$. The expected partial probes on a hit includes the minimum two probes plus those due to false partial matches: the $A-1$ non-matching tags falsely match with probability of $\frac{1}{2^K}$, but only 1/2 of them will be probed before the match is found (on average). On a miss, the partial compare approach has superior performance compared to the other inexpensive associativity implementations. Naive and MRU require *A* and $A+1$ probes on a miss, but partial expects only a single probe plus $\frac{A}{2^K}$ probes from false partial matches.

Configuration			Expected Probes	
Method	Associativity	Tag Memory Width (bits)	Assume Hit	Assume Miss
Traditional	A	$A \times T$	1	1
	4	64	1	1
Naive	A	T	$(1/2)(A-1)+1$	A
	4	16	2.5	4
MRU	A	T	$1+\sum_{i=1}^{A} i\, f_i$	$1+A$
	4	16	[2,5]	5
Partial	A	$\max(T, A \times K)$	$2+\frac{(A-1)}{2^{K+1}}$	$1+\frac{A}{2^K}$
	4	20	2.05	1.13

Table 5.4. Expected Probes of Associativity Implementations.

For various associativity implementations, this table gives the tag memory width required, and the expected probes for hits and misses. The table assumes *T*-bit tags ($T = 16$), *K*-bit partial compares ($K = 5$), and the *i*th most-recently-used tag matches with probability f_i on a hit.

While Table 5.4 shows the expected probes, Table 5.5 shows the actual number required for different inexpensive 4-way set-associative cache implementations. The results are only for read requests to the secondary cache. The performance for write-backs is less important because they can be done in the background, and because write-back cache lookups can be eliminated if the primary cache maintains a pointer indicating where (within a set) a block resides in the secondary cache (the write-back optimization of [KEJL89]). The required naive hit probes is precisely as expected because the traversal order is random, and the total for hits and misses is high. The MRU hit probes are usually lower than naive, and they also tend to decrease with cache size because the MRUth block in each set satisfies a larger portion of the secondary cache references. The Sor, Lin, and Tv traces are exceptional with MRU because there are cases where increases in cache size decrease the probability that hits come early in the MRU list, increasing the required probes. Loops in these applications cause this. Loops can make MRU less effective because there is less locality.

Read Probes (4-Way)								
		Inexpensive Scheme						
Trace	Cache Size	Naive		MRU		Partial Match		
		Hits	Total	Hits	Total	Hits	Misses	Total
Mult1	256K	2.50	2.75	2.33	2.79	2.06	1.13	**1.90**
	1M	2.50	2.58	2.24	2.39	2.11	1.13	**2.06**
	4M	2.50	2.54	2.09	2.18	2.16	1.12	**2.13**
	16M	2.50	2.52	2.04	**2.08**	2.34	1.49	2.33
Mult1.2	256K	2.50	2.74	2.31	2.74	2.06	1.13	**1.91**
	1M	2.50	2.60	2.22	2.40	2.11	1.13	**2.04**
	4M	2.50	2.55	2.09	2.19	2.14	1.11	**2.11**
	16M	2.50	2.52	2.04	**2.09**	2.34	1.48	2.33
Mult2	256K	2.50	2.69	2.38	2.71	2.06	1.13	**1.94**
	1M	2.50	2.58	2.19	2.34	2.10	1.13	**2.05**
	4M	2.50	2.55	2.07	2.16	2.15	1.13	**2.12**
	16M	2.50	2.52	2.04	**2.08**	2.35	1.61	2.34
Mult2.2	256K	2.50	2.70	2.34	2.68	2.05	1.13	**1.93**
	1M	2.50	2.59	2.18	2.35	2.09	1.12	**2.04**
	4M	2.50	2.55	2.07	2.17	2.14	1.13	**2.11**
	16M	2.50	2.52	2.04	**2.08**	2.34	1.65	2.33
Tv	256K	2.50	2.77	2.74	3.14	2.03	1.07	**1.86**
	1M	2.50	2.66	2.26	2.56	2.03	1.06	**1.92**
	4M	2.50	2.63	2.06	2.32	2.02	1.04	**1.94**
	16M	2.50	2.57	2.09	2.23	2.01	1.00	**1.96**
Sor	256K	2.50	3.50	2.24	4.07	2.03	1.09	**1.40**
	1M	2.50	3.40	2.36	3.94	2.02	1.02	**1.42**
	4M	2.50	3.00	2.79	3.53	2.01	1.03	**1.68**
	16M	2.50	2.60	2.34	2.52	2.00	1.00	**1.93**
Tree	256K	2.50	2.85	2.53	3.10	2.03	1.06	**1.81**
	1M	2.50	2.63	2.33	2.56	2.04	1.04	**1.95**
	4M	2.50	2.54	2.12	2.19	2.04	1.02	**2.01**
	16M	2.50	2.52	2.03	2.07	2.05	1.01	**2.04**
Lin	256K	2.50	3.02	2.19	3.17	2.02	1.02	**1.67**
	1M	2.50	2.99	2.09	3.04	2.02	1.00	**1.68**
	4M	2.50	2.52	2.55	2.57	2.02	1.00	**2.01**
	16M	2.50	2.51	2.02	**2.03**	2.05	1.00	2.05
Theory		2.50		[2,5]		2.05	1.13	

Table 5.5. Read Probes of Inexpensive Associativity Implementations.

For various 4-way set-associative secondary caches, this table shows the probes required for a secondary cache read access. It shows results for Naive hits and total (hits + misses), MRU hits and total, and partial match hits, misses, and total. The partial compares of the virtual tags, including PID's, use 5-bits and the improved transformation of [KEJL89]. This constant partial compare width may be inappropriate since the tag width may be reduced with increasing cache size. The table shows the expected results (theory) from Table 5.4 at the bottom. The best total technique for each row is in **Bold**.

Because the assumption of random tags is invalid with virtual tags, the 5-bit partial compare results in Table 5.5 sometimes differ from the expected results in Table 5.4, even though partial hashes the virtual address bits to make them more random. For the multiprogrammed workloads, the small caches have nearly the expected performance. For the larger caches, however, the partial tags from the different address spaces in the multiprogrammed workloads tend to be similar because most programs use the same portions of the virtual address space, so partial has more false matches and requires more probes than expected. For the single-process workloads, the partial tags from the single address space

are more likely to differ than random, so the required probes are fewer than expected. This occurs because the dispersion of a single, largely contiguous, address space throughout a large cache leads to differing tag values since sequential virtual addresses have sequential virtual tag values. The non-randomness of the partial compare results for the different workloads suggests that one should take care when deciding which, and how many, bits to use in the partial compares. Real tags would likely be more random than the virtual tags used here (provided virtual pages are stored in random page frames), so real-tag results probably would closely follow those shown in Table 5.4.

Partial does the best for all but the largest (16-megabyte) caches. Partial is fast on misses, which is important because secondary caches have higher (local)miss ratios than do primary caches of the same size. The superiority of the partial technique with the smaller secondary caches is evidence supporting the importance of fast misses. MRU performs well for the large caches because a large portion of the hits are to the MRUth block in the secondary cache. This agrees with the observations of So and Rechtschaffen on primary caches [SORE88]. MRU is effective when there is locality of reference. It is less effective when the secondary cache is close in size to the primary caches, as in the 256-kilobyte case in Table 5.5, because the primary cache steals much of the locality from the secondary cache. Unless an MRU memory can be made particularly fast, partial performs better for the smaller secondary caches. MRU is often better for the 16-megabyte caches, however. As the cache size increases, MRU can successfully extract locality from the stream of primary cache misses, so the number of probes on cache hits is low. Also, since the *MPI* of the larger caches is lower, the poor performance of MRU on misses is less important. If the partial compare width decreases with cache size, as may be the case, MRU can be even more appropriate for larger caches since the partial performance will be degraded. On the other hand, larger tags, such as those required for 64-bit addresses, may allow a partial compare width of more than five bits. Then, partial may again be more appropriate.

5.7. Secondary Cache Block Size Alternatives

Another important cache design choice is the block size. Since the secondary caches are the sizes that main memories have previously been, one might mistakenly expect that secondary cache block sizes should be the size that pages previously were. The analogy to main memory pages does not hold because the backing store for the secondary cache (main memory) is orders of magnitude faster than the backing store for main memory (disk). Page sizes should be larger because once the disk is accessed, there is only a small additional latency to bring in more data. A larger block size will typically reduce the *MPI* because of spatial locality [SMIT82], and it also will reduce the tag memory requirements of the cache. But since main memory access times are much smaller than disk access times, the cache miss time can be greatly increased with a larger block size. Furthermore, larger blocks require more memory bandwidth: if contention for memory resources is a problem, smaller block sizes might be preferred. The optimal block size is in the middle, between the higher *MPI* with smaller block sizes and the larger miss penalty of larger block sizes.

To measure this effect, $Tmiss_{SCACHE}$ is a linear function of the block size that counts both *latency* (fixed) and *transfer time* (variable) components, as in Smith's model [SMIT87]. The fixed latency component is independent of the block size. It counts the fixed latency component of block transfer, such as reaching and cycling the main memory. The transfer time component is directly proportional to the block size and counts the time to transfer each byte of the block. The formula for the block size latency model is:

$$Tmiss_{SCACHE} = T_{latency} + T_{transfer} \times B. \quad (5.2)$$

$T_{latency}$ is the latency component, $T_{transfer}$ is the transfer time required for each byte of the block, and B is the block size in bytes.

The best block size depends on the values of $T_{latency}$ and $T_{transfer}$. The architecture of the cache-main memory communication decides these values. $T_{transfer}$ can be reduced by loading the bytes of a cache block in parallel rather than serially [MATI89]. This may involve having multiple memory banks operate in parallel, or having a single, wider memory bank. $T_{transfer}$ also could be reduced by completing the fetch of a block in the background after first returning the requested data. While $T_{transfer}$ may be changeable, it is more difficult to reduce $T_{latency}$ since a substantial portion of it is the main memory access time.

The equation for the block size with equal latency and transfer time is:

$$B = \frac{T_{latency}}{T_{transfer}}.$$

This block size is an important design option [PRZY90]. For smaller B, the latency can dominate the time for a cache miss. Sequential memory locations could be loaded into the cache considerably faster with a larger block size. For larger B, the transfer time can dominate. This can lead to poor performance for workloads that tend to reference small portions of many different cache blocks. A designer might choose the block size where latency equals transfer time since data can be retrieved from the memory system with no more than twice the minimum time caused by either the fixed latency or transfer time. This can ensure that the block size is close to optimal for workloads where either latency or transfer time (or both) is essential to memory system performance.

Figure 5.13 shows *SCPI* versus the block size with $T_{latency} = 64$ and $T_{transfer} = 0.5$ for the Mult1.2 trace (the latency equals the transfer time at 128-byte blocks). It shows that 64-byte blocks are best with the smaller caches, 128-byte blocks are best for a 1-megabyte cache, and 256-byte blocks are best for the 16-megabyte cache (though it is hard to see). Cache contention in the smaller caches made the smaller block sizes better; the smaller caches were unable to take advantage of enough spatial locality, so a large block size resulted in significant amounts of unnecessary data transfer. In the largest cache, the 256-byte blocks were the best since there was less cache contention. The bigger caches preferred the increased transfer times of larger blocks so that the fixed latency could be minimized. Except for the smallest caches, 128-byte blocks perform quite well with the Mult1.2 workload, showing the usefulness of the heuristic equalization of latency and transfer time.

To examine the block size alternatives for other workloads, Table 5.6 lists the *SCPI* of direct-mapped secondary caches with varying block sizes. The results show the same tradeoff between the *MPI* reduction and unnecessary data transfer of larger block sizes. For smaller caches, smaller blocks are preferred. As the cache size increases, there is less contention so the fixed latency of a memory operation can be amortized over larger data transfers: larger blocks will give lower *SCPI*'s. The lowest *SCPI* occurs with block sizes of 32-bytes or 64-bytes with a 256-kilobyte cache, 64-bytes or 128-bytes with a 1-megabyte cache, about 128-bytes with a 4-megabyte cache, and 128-bytes or 256-bytes with a 16-megabyte cache.

The multiprogrammed traces have similar behaviors while the uniprogrammed trace results vary. The Sor trace is especially interesting. One might expect a scientific program to prefer large block

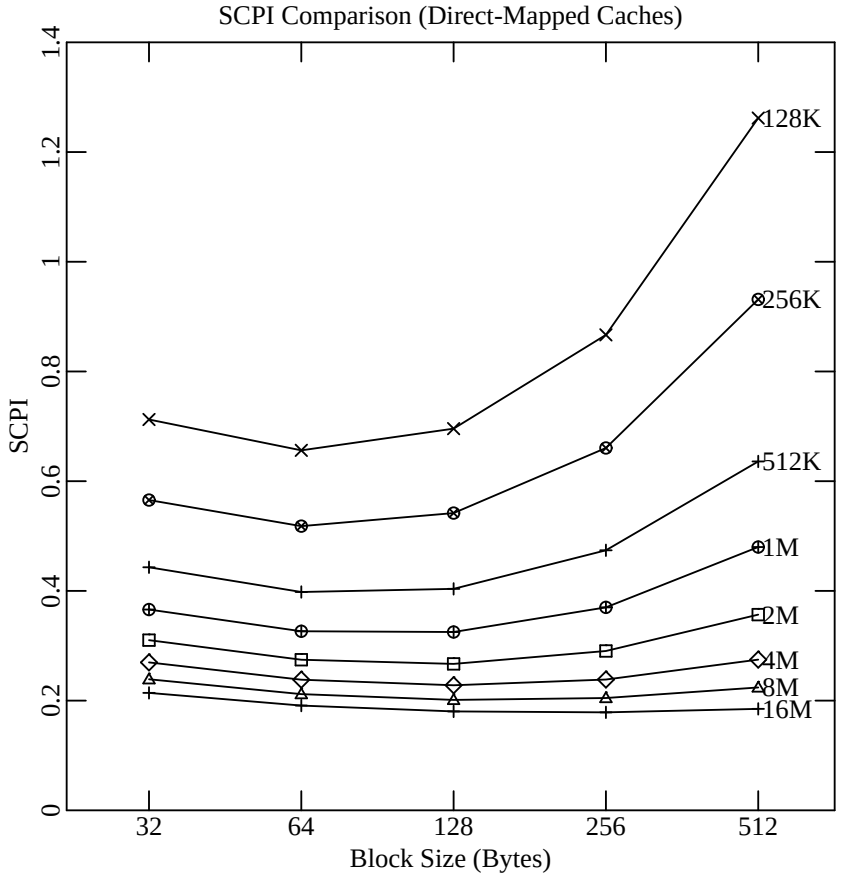


Figure 5.13. Performance Over a Range of Block Sizes.

For the Mult1.2 trace, this figure shows the *SCPI* versus the block size for a range of direct-mapped secondary caches. $T_{latency}$=64 and $T_{transfer}$=0.5.

sizes, particularly when traversing through large arrays. Except for the 16-megabyte cache, smaller 32-byte and 64-byte blocks give the minimum *SCPI* for Sor. The Sor sparse array representations result in less spatial locality and cause this preference for smaller blocks. However, sparse representations do not automatically imply that smaller blocks are better. Lin uses sparse arrays but slightly prefers larger block sizes.

For all the workloads, a 64-byte or 128-byte block gives the minimum *SCPI* for the 1-megabyte cache. Considering that larger block sizes require smaller tag memories, a block size of 128-bytes seems like a reasonable choice, precisely the block size where the latency equals the transfer time. On the other hand, if memory bandwidth and contention are a concern, a block size of 64-bytes might be preferred.

The results in Figure 5.13 and Table 5.6 agree with the results of Smith [SMIT87] and Przybylski [PRZY90] in that the block size with least *SCPI* is largely a function of the cache-main memory communication parameters. The secondary cache size is a substantial factor, but the *SCPI*-minimal block size varies surprisingly little over cache sizes from 256-kilobytes to 16-megabytes (a factor of 64!). Smaller block sizes are preferred with smaller caches, but the *SCPI*-minimal size is usually no more than a factor of two from the block size with equal latency and transfer times (128-bytes). Except for

SCPI For Different Traces (Direct-Mapped)						
Trace	Cache Size	Secondary Block Size (Bytes) 32	64	128	256	512
Mult1	256K	0.61	**0.56**	0.60	0.74	1.05
	1M	0.39	**0.35**	**0.35**	0.41	0.55
	4M	0.28	0.25	**0.24**	0.26	0.30
	16M	0.23	0.20	**0.19**	**0.19**	0.20
Mult1.2	256K	0.57	**0.52**	0.54	0.66	0.93
	1M	0.37	**0.33**	**0.33**	0.37	0.48
	4M	0.27	0.24	**0.23**	0.24	0.27
	16M	0.21	0.19	**0.18**	**0.18**	**0.18**
Mult2	256K	0.58	**0.53**	0.55	0.65	0.96
	1M	0.36	0.31	**0.30**	0.33	0.40
	4M	0.27	0.23	**0.22**	**0.22**	0.25
	16M	0.21	0.19	0.18	**0.17**	0.18
Mult2.2	256K	0.53	**0.48**	0.49	0.58	0.82
	1M	0.34	0.29	**0.28**	0.30	0.37
	4M	0.25	0.22	**0.20**	0.21	0.23
	16M	0.19	0.17	**0.16**	**0.16**	**0.16**
Tv	256K	0.91	**0.90**	0.98	1.34	2.00
	1M	0.58	**0.54**	**0.54**	0.60	0.73
	4M	0.49	0.45	**0.44**	0.46	0.50
	16M	0.35	0.34	**0.33**	0.34	0.36
Sor	256K	**1.95**	2.08	2.72	4.24	7.47
	1M	1.85	**1.80**	2.12	3.05	5.24
	4M	1.39	**1.18**	1.19	1.47	2.17
	16M	0.85	0.61	0.48	0.42	**0.39**
Tree	256K	**0.69**	0.70	0.83	1.18	1.84
	1M	0.46	**0.42**	0.45	0.56	0.80
	4M	0.28	**0.25**	**0.25**	0.27	0.31
	16M	0.25	0.22	**0.21**	**0.21**	0.22
Lin	256K	0.27	0.22	0.22	**0.20**	0.21
	1M	0.19	0.18	0.19	0.18	**0.17**
	4M	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**
	16M	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**

Table 5.6. Block Size Alternatives.

This table shows the *SCPI* for direct-mapped secondary caches with different block sizes. $T_{latency}$=64 and $T_{transfer}$=0.5. The bold entries show the lowest *SCPI* in each row.

the smallest secondary caches, the 128-byte block *SCPI* is near the minimal *SCPI*.

The *SCPI*-minimal block sizes are larger than those found in [SMIT87], and [PRZY90]. For example, Przybylski finds block sizes of 16-bytes to 32-bytes to be optimal for most caches. The smaller caches and different $T_{latency}$ and $T_{transfer}$ values considered in those studies are the cause of this discrepancy. For the large $T_{latency}$ values and multi-megabyte caches examined here, secondary cache block sizes of 64-bytes or 128-bytes were better.

5.8. Inclusion Design Alternatives

An important design choice with multi-level cache configurations is whether the hierarchy will maintain inclusion. For the two-level hierarchical system examined in this dissertation, inclusion implies that any memory location held in either of the primary caches also must be in the secondary cache [BAEW88].

Inclusion can be useful when hardware maintains cache data consistent in the presence of external accesses to the memory locations. In a uniprocessor system, inconsistencies result from writes to cached locations by I/O devices. In a shared-memory multiprocessor, inconsistencies result from updates of cached locations by other processors. Hardware monitors resolve these inconsistencies by invalidating or updating cached locations when necessary. When maintaining inclusion, a memory location that is not in the secondary cache is also not in the primary cache. Thus, if a secondary cache invalidate (or update) was unnecessary, it is *guaranteed* that a primary cache invalidate (update) also will be unnecessary. The effect of inclusion is that the secondary cache can filter out many consistency operations, reducing the communication and interference with the primary caches. Without inclusion maintenance, any consistency operation could affect the primary caches, so all them may have to be passed on.

Inclusion is desirable for hardware consistency management, but it has an implementation cost. While it may seem intuitively clear that a smaller cache should contain a subset of a larger cache, inclusion may be nullified because of block size, associativity, and indexing differences between the primary and secondary caches. Baer and Wang outline the configuration requirements for a restricted form of inclusion, where primary cache locations would never have to be invalidated to ensure inclusion [BAEW88, WANG89]. Unfortunately, the requirements for this form of inclusion are strict; often it requires extremely high associativity in the secondary cache. More general circumstances occasionally require primary cache invalidations to maintain inclusion.

Wang described two invalidation techniques [WANG89]. *Implicit* inclusion invalidates the primary caches each time the secondary cache replaces a block. *Explicit* inclusion eliminates all unnecessary primary cache invalidations since it maintains information denoting whether the primary caches hold a (perhaps dirty) copy of a replaced block. Both techniques guarantee inclusion by invalidating the primary cache on secondary cache misses. The difference between the two is that implicit inclusion requires an invalidation on *every* secondary cache miss, while explicit inclusion executes only the invalidations that are *required*. Since the primary cache block size is smaller than the secondary cache block size, each primary block can hold only a portion of a secondary block. The extra information needed by explicit inclusion is a valid bit (or two bits, one each for the instruction and data caches) for each portion of each secondary cache block frame. When the valid bits are not set, as is often the case, primary cache invalidations are unnecessary when a secondary cache block is replaced.

Table 5.7 shows the frequency of primary cache invalidations required by implicit and explicit inclusion with 4-megabyte secondary caches. Each primary cache invalidation applies to both the instruction and data caches. The implicit invalidation frequency is simply four times the secondary cache *MPI* since there are four different 32-byte primary cache blocks contained in each 128-byte secondary block; only some implicit invalidations actually remove a primary cache block. Not all secondary cache misses require explicit invalidations; each explicit invalidation removes a primary cache block. For explicit inclusion, Table 5.7 shows results with random and inclusion-random replacement. Inclusion-random (INCRAND) replaces secondary cache blocks to minimize the primary cache invalidation frequency. Since the explicit inclusion information is available, replacement decisions may be improved by taking inclusion information into account. Table 5.3 showed that inclusion-random replacement reduces the secondary cache *MPI*. These results show that it also reduces the explicit inclusion primary cache invalidation frequency. This replacement policy is not an option with implicit inclusion because the inclusion information is not available.

Primary Cache Invalidates Per 1000 Instructions				
Trace	Assoc.	Implicit	Explicit	
		RAND	RAND	INCRAND
Mult1	1	2.924	0.214	0.214
	2	2.205	0.056	0.007
	4	2.062	0.035	0.000
Mult1.2	1	2.863	0.189	0.189
	2	2.239	0.051	0.005
	4	2.116	0.035	0.000
Mult2	1	2.516	0.185	0.185
	2	2.062	0.048	0.004
	4	1.992	0.033	0.000
Mult2.2	1	2.426	0.169	0.169
	2	2.025	0.045	0.004
	4	1.957	0.031	0.000
Tv	1	7.588	0.162	0.162
	2	7.046	0.050	0.002
	4	7.041	0.044	0.000
Sor	1	29.616	0.263	0.263
	2	30.630	0.191	0.005
	4	31.626	0.179	0.000
Tree	1	2.533	0.142	0.142
	2	1.944	0.028	0.001
	4	1.859	0.021	0.000
Lin	1	0.405	0.025	0.025
	2	0.232	0.003	0.000
	4	0.143	0.001	0.000

Table 5.7. Implicit and Explicit Inclusion Invalidate Frequency.

This table shows the frequency of implicit and explicit primary cache invalidates for random replacement (RAND) and inclusion replacement (INCRAND). The primary caches are split direct-mapped 32-kilobyte with 32-byte blocks. The secondary cache is 4-megabytes with a block size of 128-bytes. INCRAND is random replacement except it replaces blocks with copies in the primary cache only when there is no other choice, as in Table 5.3.

The results in Table 5.7 show that the invalidation frequency is much lower with explicit inclusion. This agrees with Wang's results for smaller caches [WANG89]. Although secondary cache replacement is infrequent, required invalidations are at least an order of magnitude more rare, and inclusion-random replacement almost eliminates them. With associativity and inclusion-random replacement, invalidations are extremely rare since the secondary cache is so large compared to the primary caches. There are considerably more explicit invalidations with the direct-mapped secondary caches. Direct-mapping increases the likelihood of a primary cache invalidation caused by conflicting secondary cache blocks.

Since a primary cache block may be dirty, primary cache invalidations may have to be executed before the secondary cache replaces a block. Consequently, the extra implicit invalidations may increase $Tmiss_{SCACHE}$ and *SCPI*. Explicit inclusion eliminates many invalidations, but maintaining the extra inclusion information can be costly. For it to be precise, the information should be updated each time a block is replaced or loaded from a primary cache. This means that explicit inclusion may increase $Tmiss_{ICACHE}$ and $Tmiss_{DCACHE}$, and consequently also increase *SCPI*.

Maintaining hardware inclusion has a cost, for one thing a more complex implementation, and also perhaps a higher *SCPI*. For multiprocessors with sophisticated hardware cache consistency mechanisms, inclusion may be necessary. For uniprocessors, however, it may be more appropriate to maintain cache consistency in software. The operating system can invalidate cached locations when there is a potential conflict. Then, if memory accessed by I/O or other devices is not in the cache, there can be no inconsistencies. Provided that software invalidations are fast, software consistency maintenance can improve system performance and reduce hardware complexity. If software invalidations are too slow, or if they have to be executed too often because the operating system must conservatively invalidate, hardware inclusion and consistency may be a better alternative.

5.9. Conclusions

This chapter discusses the motivation for multi-megabyte secondary caches and analyzes design tradeoffs in these large caches. The two key trends discussed in Chapter 1 motivate multi-megabyte caches: increasing processor speeds and larger main memories. Without an adequate cache, faster processors may wait for main memory too often because each main memory access takes too long. Multi-megabyte caches are more essential with faster processors because they eliminate much of the main memory access penalties. The bigger workloads that come with faster processors and larger main memories even further motivate multi-megabyte caches; Smaller caches may not have high enough hit ratios with these workloads. CPU cache hierarchies combine the fast access time of a smaller cache with the storage capabilities of a multi-megabyte cache. A multi-megabyte secondary cache allows the hierarchy to satisfy memory requests at processor speeds, even when each main memory access is 100 cycles or more.

Several factors determine the best size for a secondary cache. As the cache miss time increases, a larger cache size is needed for good performance. The results of this chapter show that multi-megabyte caches reduce the *SCPI* of smaller caches, even if each cache size doubling increases the cache access time by 10% or more. The target workload of the system is also an important factor in the cache size choice. One workload may perform well with one cache while another does not. Larger caches give good performance for a wider variety of applications.

Associativity is another major design consideration. This chapter shows that an associativity doubling improves direct-mapped multi-megabyte cache performance less than a cache size doubling does, which differs from the 2:1 rule that says each doubling should have about the same effect in smaller caches. This chapter also shows that higher associativity is particularly useful only when the secondary cache is not a lot larger than the primary caches. Direct-mapping performs very well for the largest secondary caches; associativity increases in the largest secondary caches may not be justified if they cause any increases in design time, access time, or cost. This chapter shows that LRU reduces *MPI* for most workloads. The increased complexity of LRU may not be worthwhile, however, since some workloads do not benefit much from it. This chapter also shows that the partial match technique looks like a primising inexpensive associativity implementation, especially for the smaller secondary caches. MRU also performs well for the largest secondary caches.

This chapter finds that block sizes of 128-bytes perform well for the traces and parameters in this study, validating a design choice of equal latency and transfer times that previous studies have advocated. The *SCPI*-minimal block size is often within a factor of two of the 128-byte block size, and the 128-byte *SCPI* is usually near the minimum *SCPI*. This emphasizes that the best block size for a given

cache is heavily dependent on the parameters of the cache-main memory intercommunication. Larger fixed latencies imply that larger blocks are preferred while larger transfer times imply that smaller block sizes are preferred. Multi-megabyte secondary caches prefer bigger blocks because they can better exploit spatial locality and mitigate fixed latencies since they have less contention for cache locations.

Inclusion simplifies a hardware consistency mechanism, but its cost can be high, both in terms of implementation complexity and *SCPI* increase. Cache replacement decisions can significantly affect the number of the primary cache invalidations needed to enforce consistency.

Chapter 6

Page Placement Algorithms for Real-Indexed Caches

6.1. Introduction

This dissertation chapter exposes the affect of virtual memory page mapping on real-indexed multi-megabyte cache performance. The associativity difference between (virtual memory) address translation and (set-associative) cache set-indexing implies that the page placement (or page mapping) in the main memory determines the data placement in a real-indexed multi-megabyte cache. Figure 6.1 shows the combined stages (both virtual memory and cache) that index a virtually addressed memory location into a multi-megabyte set-associative cache. Multi-megabyte caches require so many set-indexing bits that some of them must come from above the page boundary. This creates what are called cache *bins*. The upper index bits that come from above the page boundary are the *bin index* and the group of sets that they select is a bin[22]. Address translation selects cache bins only when a cache is both large and real indexed: small caches have no bins because they get their index bits from below the page boundary, and virtual-indexed caches get all their index bits directly from the virtual address.

Multi-megabyte secondary caches are likely to be real indexed because the address translation cost of real indexing can be small enough in secondary caches that the benefits of real indexing exceed its cost [TADF90, WABL89]. The previous chapters do not consider real-indexed caches; instead, they use virtual-indexed caches (with PID-hashing) to approximate real-indexed caches. This chapter examines the differences between virtual-indexed caches and real-indexed ones, and determines how closely

22. Figure 6.1 depicts bins that are four cache sets. Note that the *superset* of Goodman [GOOD87] is different from a bin. Pieces of a superset span *all* bins.

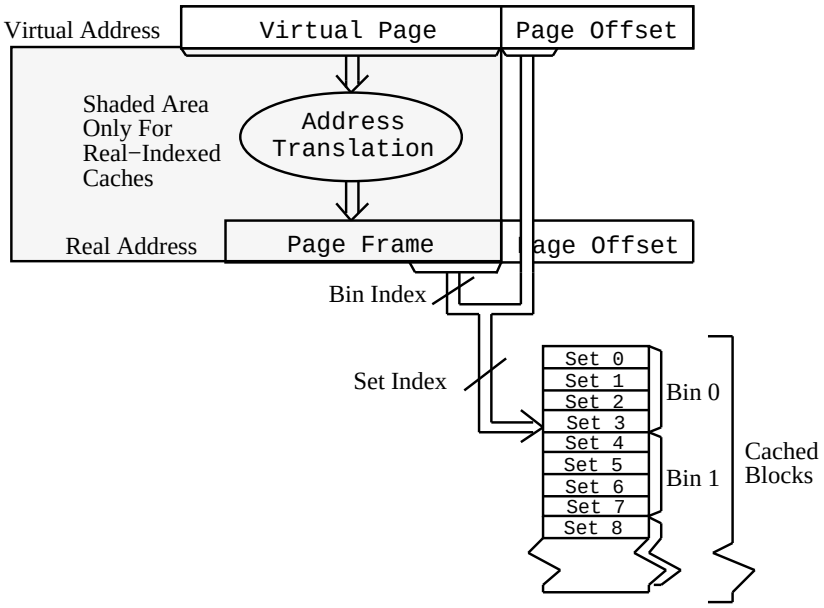


Figure 6.1. Virtual Memory and Set Indexing Interaction.

This figure shows translation stages that index a virtually addressed memory location into a large set-associative CPU cache. The cache takes index bits directly from the virtual address for a virtually-indexed cache, and it takes the bits from the real address for a real-indexed cache. Any of the *A* (*A* is the associativity) cache block frames in the set chosen by the *index* bits can contain the addressed memory location. This chapter only considers multi-megabyte caches with bit-selection indexing [SMIT82].

virtual-indexed cache performance approximates real-indexed performance. The page mapping can greatly affect real-indexed multi-megabyte cache behavior because address translation places each page in any of a hundred or even several thousand different cache bins. A poor page placement reduces real-indexed multi-megabyte cache efficiency because it overutilizes some cache bins and underutilizes others. This chapter examines the interaction of page placement with real-indexed multi-megabyte caches, and develops software techniques that avoid poor page mappings, and consequently increase hardware cache performance.

6.1.1. Previous Work

Few previous studies examine the page mapping real-indexed cache performance effect because it is only recently that real-indexed caches have started taking set-indexing bits from above the page boundary. Sites and Agarwal [SITA88] compare the performance of virtual-indexed caches to real-indexed caches. They find that real-indexed caches perform worse than the corresponding virtual-indexed caches on the same workloads unless there is frequent context switching.

Only a few systems optimize the page mapping for cache performance. In the Sun 4/110, the operating system maps instruction pages and data pages [KELL90] to even and odd page frames to partition the instruction and data references into the 4/110's SCRAM cache [GOOC84]. MIPS uses a variant of *Page Coloring*, described in Section 6.3, to improve and stabilize its real-indexed cache performance [TADF90].

Many studies optimize the addressing of programs with compiler or user-level optimizations. Ferrari [FERR76] surveys many schemes to improve virtual memory performance by referencing fewer

pages. Stamos tries similarly to cluster objects and improve virtual memory performance [STAM84]. McFarling [MCFA89] and Hwu and Chang [HWUC89] introduce schemes to scatter commonly used instructions across direct-mapped caches to reduce instruction cache misses. Cache performance also improves with algorithm modifications such as matrix blocking [LARW91]. This chapter instead introduces improvements to the operating system memory management software. Though the virtual memory references cannot be modified by the memory management, the real addresses can.

6.1.2. Contributions of this Chapter

This chapter examines the software page placement effect on multi-megabyte real-indexed cache performance. It develops software page placement algorithm improvements that can increase multi-megabyte real-indexed cache performance. The trace-driven simulation results of this chapter suggest that these software page placement improvements can make a hardware direct-mapped cache appear about 50% larger, at no hardware cost.

The improved placement algorithms are called *careful page mapping* algorithms. They are small modifications to the existing operating system virtual memory management software that improve *dynamic* cache performance with better *static* page bin placement decisions. Most operating systems map (place) new pages into the main memory by selecting from a pool of available page frames. Normally the operating system selects an arbitrary page frame from the pool, probably the first one that is available. In the terminology of this chapter, this arbitrary selection is called *random* (or naive) page mapping because the operating system does not know (or care) *where* it places the page in the main memory when it chooses an available page frame. Careful page mapping algorithms don't just map pages to arbitrarily available page frames. Instead, when the pool allows some mapping flexibility, they choose one of the available page frames in the pool that best suits some simple static cache bin placement heuristics. These algorithms are low overhead, particularly since they may only execute once when each page is mapped, while they may improve cache performance each time the processor references the pages.

Section 6.2 motivates the static improvements that these careful mapping algorithms rely on, first qualitatively, then quantitatively. It first shows how the operating system can improve static page bin placement, and extract many of the advantages of virtual-indexed caches. A simple static analysis then shows the potential careful mapping static improvements: as many as 30% of the pages from an address space are unnecessarily in conflict in a direct-mapped cache when using random mapping. This analysis also correctly predicts that the largest gain from careful page mapping comes when the cache is direct-mapped.

Section 6.3 describes Page Coloring and introduces several more careful page mapping algorithms. *Page Coloring* matches the real-indexed bin to its corresponding virtual-indexed bin. *Bin Hopping* places successively mapped pages in successive bins. *Best Bin* selects a page frame from the bin with the fewest previously allocated and most available page frames. *Hierarchical* is a tree-based variant of Best Bin that executes in time logarithmic of the number of bins, and is cache size independent (i.e. it improves static placement in many different cache sizes simultaneously).

Section 6.4 uses trace-driven simulation simulation results to show that the static bin placement improvements of these policies eliminate cache misses. It shows that careful page mapping eliminates 10-20% of the direct-mapped cache misses from the Chapter 3 traces. This is about half of the improvement from either doubling the size or the associativity of a direct-mapped cache.

Section 6.4 correlates the dynamic cache miss reductions with static page placement improvements. It also compares the performance of the different careful page mapping implementations for one multiprogrammed trace. This comparison shows that Best Bin eliminates the most misses but that Bin Hopping and Hierarchical also perform well with a large pool of available page frames, and that Hierarchical's size-independence optimizes for many caches. Section 6.4 then shows that virtual indexing (with PID-hashing) optimistically estimates real-indexed cache performance, that careful mapping is effective over a range of page sizes, and that a pool of available page frames that is too large (relative to the main memory size) can cause more page misses.

6.2. Motivation for Careful Page Mapping

This section motivates the careful page mapping algorithms by demonstrating, first qualitatively and then quantitatively, the potential static real-indexed page placement improvement of careful mapping over random mapping.

6.2.1. Page Placement in Cache Bins

To fully understand how to improve real-indexed cache page placements, it is useful to first understand how (and why) virtual-indexed caches can perform better than real-indexed ones. The placement of a page in a virtually-indexed cache bin depends only on the placement of the page in the virtual address space. Figure 6.2 depicts a typical virtual address space and its virtual-indexed bin placement. The code area, holding the (read only) executable instructions, resides in the lower portion of the address space, followed by the data area, a modifiable working space of the process. The procedure call stack grows from the top of the virtual address space downward, and there is a wide separation between the stack and data areas. The lowest address is zero and the highest address bits are all ones. The virtual indexing in Figure 6.2 places the contiguous code and data pages in the lower bins, and it places the stack pages in the upper bins. The address space is large enough in this example so that the data addresses *wrap around*, and two pages index to some bins. Since virtual indexing places sequential virtual pages in sequential cache bins [SITA88], virtual address spaces that have contiguously-allocated pages will evenly utilize the cache.

Real indexing is much different from virtual indexing since the address translation shown in the shaded area of Figure 6.1 determines bin placement. Since the mapping of pages to page frames is usually fully associative, there may be no relationship between the real-indexed cache bin placement and the virtual address of a page. Once the operating system places a page in a page frame, the page will index to the bin determined by the bottom bits, or bin bits, from the page frame number. If the operating system ignores the values of these bits when it maps virtual memory pages to main memory page frames, address translation will cause cache bin randomization. Figure 6.3 depicts an example of this randomization on the left, where pages from the address space in Figure 6.2 are randomly placed in the real-indexed cache.

The left (random) placement in Figure 6.3 is poor because it puts many pages in the same cache bins, where they compete for the same cache locations. Competition is undesirable since it can cause more cache *conflict misses* [HILS89]. Careful page mapping policies can produce page placements more like the right side of Figure 6.3 than the left, so that cache bins are evenly utilized and many unnecessary conflict misses are eliminated. Virtual indexing may have more even bin utilization than random real-indexing, but a careful page placement makes a real-indexed cache perform more closely

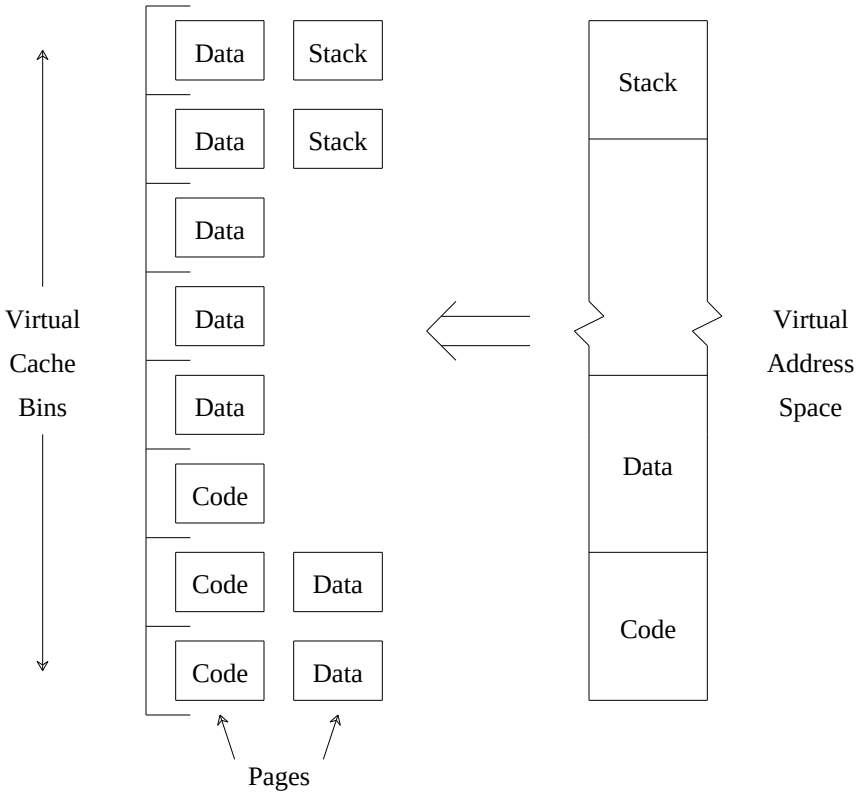


Figure 6.2. A Virtual-Indexed Bin Placement.

This figure shows a placement of the pages from a virtual address space in a virtually-indexed cache. The pages horizontally stacked in a bin are the ones that the virtual-indexing indexes to the bin.

like a virtual-indexed one because it more evenly utilizes the cache bins like virtual indexing does.

6.2.2. A Simple Static Page Conflict Analysis

The difference in the two mappings in Figure 6.3 shows the possible static placement improvements of careful page mapping visually. This section develops a simple model to measure the potential improvements quantitatively. The metric calculated by the model is *page conflicts*, *C*, which is the number of pages in the cache bins above the cache associativity (*A*). Once there are more than *A* pages in a bin there can be cache contention because the cache can only store *A* cache blocks in each set. Thus, when *u* pages land in a cache bin, there are said to be $max(0, u - A)$ page conflicts for that bin[23], and the total conflicts in a page mapping is the sum of the conflicts from all the pages in each bin. As an example, with a direct-mapped cache there are six conflicts in the left placement of the address space shown in Figure 6.3, and there are four conflicts in the right placement of the same address space. Similarly, there are three and zero conflicts for a 2-way set-associative cache.

23. This chapter uses *u-A* rather than *u* because it more correctly predicts the magnitude of the conflict. If *u* was used instead, then conflicts would be minimized by first filling the cache (placing *A* pages in each bin) and then placing all the rest of the pages in a single bin; it is unlikely that this mapping would minimize dynamic cache misses. Similarly, with many pages, static conflicts would be maximized when the pages were evenly spread across the cache; this would much more likely minimize than maximize dynamic misses.

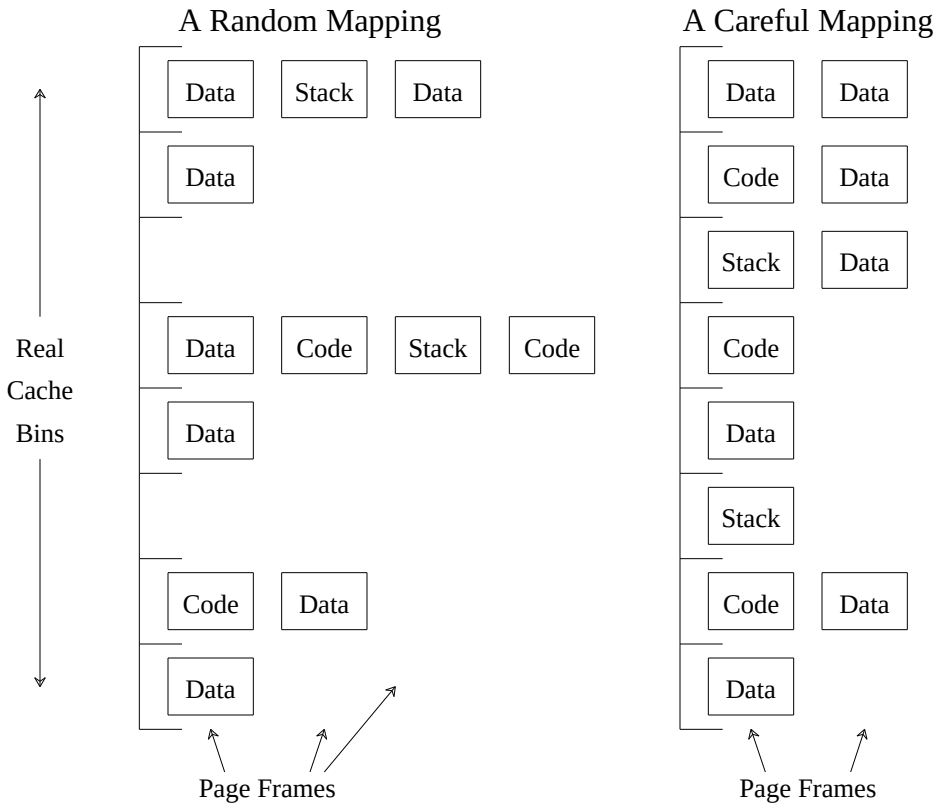


Figure 6.3. Random and Careful Real-Indexed Bin Placements.

This figure shows two placements of the same address space in Figure 6.2 into a real-indexed cache. The left bin placement might result from random mapping and the right one might be a careful mapping. The horizontally stacked pages are the (competing) ones that the virtual to real address translation indexes to the same bin.

The average page conflicts from random mappings can be calculated to quantify the potential static mapping improvement of careful page mapping. The number of bins of a set-associative cache, *B*, is the cache size in pages, *N*, divided by the cache associativity, *A*: $B = \frac{N}{A} = 2^{depth}$, where *depth* is the number of bin-index bits. With random mapping, the operating system maps the *U* pages from an address space to page frames by randomly sampling (without replacement!) from a finite population of page frames (i.e. the main memory). There are *P* available page frames, and exactly $\frac{1}{B}P$ (an integer) page frames are in each of the *B* bins. A hypergeometric distribution gives the probability that exactly *u* of the *U* pages randomly fall in one of the *B* bins [MILF77]:

$$P(u \text{ in bin}) = \frac{\binom{\frac{1}{B}P}{u}\binom{\frac{(B-1)}{B}P}{U-u}}{\binom{P}{U}}. \qquad (6.1)$$

The binomial coefficient

$$\binom{a}{b} = \frac{a!}{(a-b)!\,b!}$$

represents the number of ways to choose *b* elements from *a* elements (without replacement).

When *P* is large, the hypergeometric distribution in Equation 6.1 can be approximated by a binomial distribution[24]:

$$P(u \text{ in bin}) = \binom{U}{u}(\frac{1}{B})^u(1-\frac{1}{B})^{U-u}. \tag{6.2}$$

For many realistic cases, *P* is sufficiently close to infinity that Equation 6.2 is the same as Equation 6.1, so it may be used because it is simpler.

Equation 6.3 calculates the expected conflicts of a random placement of *U* pages using Equation 6.1 (Equation 6.2 also could be used):

$$C_{avg} = B \sum_{u=A+1}^{min(\frac{P}{B}, U)} (u-A)P(u \text{ in bin}). \tag{6.3}$$

It multiplies $u-A$, the conflict pages, times the probability of $u > A$ pages in a bin to produce the expected conflict pages in each bin. This times the number of bins gives the total expected conflict pages.

The bounds on *C* are also important to know because they gauge the potential static conflict variability of different placements. The maximum conflicts in a placement of *U* pages into a real-indexed cache, $C_{\max}$, is:

$$C_{\max} = U - A\left\lfloor U\frac{B}{P}\right\rfloor - min(U-\frac{P}{B}\left\lfloor U\frac{B}{P}\right\rfloor, A) \tag{6.4}$$

where $\left\lceil U\frac{B}{P}\right\rceil$ is the minimum number of bins that the *U* pages can be placed in. The complexity of this equation occurs since it may not be possible to place all *U* pages of an address space in a single cache bin when *P* is finite. As $P \rightarrow \infty$ Equation 6.4 simplifies to

$$C_{\max} = max(0, U-A)$$

since all *U* pages can be placed in the same bin. The minimum conflicts, $C_{\min}$, occurs when the pages of the address space evenly distribute across the bins of the cache, so

$$C_{\min} = max(0, U-N). \tag{6.5}$$

Figure 6.4 plots C_{avg} and its bounds for various address space sizes (*U*). (Note that the address space size is the number of pages pages referenced in an address space.) In the worst case, the operating system places all pages in the same bin, so $C_{\max}$ is linear with the address space size. In the best case, the operating system evenly utilizes the cache, so no conflicts occur until the cache is full of pages, and then each additional page adds a new conflict.

Figure 6.5 illustrates the difference between C_{avg} and $C_{\min}$ more dramatically for direct-mapped, 2-way, and 4-way set-associative caches. It plots the random conflicts less the minimum, or the

24. Both Thiebaut and Stone [THIS87] and Agarwal, et al. [AGHH89], also use binomial distributions for similar analysis.

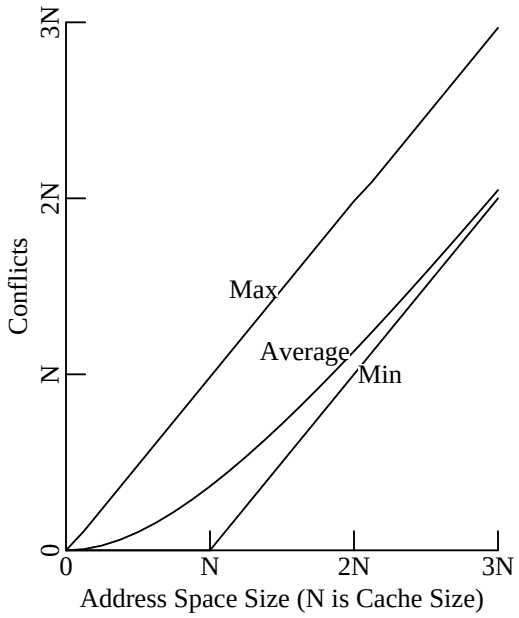


Figure 6.4. Cache Placement Conflicts.

This figure shows C_{avg}, $C_{\max}$, and $C_{\min}$ (as calculated by Equations 6.3, 6.4, and 6.5) for various address space sizes (U). The modeled system is a 1-megabyte direct-mapped ($A=1$) cache ($N=B=64$) backed up by a 128-megabyte main memory, with a page size of 16-kilobytes.

mapping conflicts ($C_{avg}-C_{\min}$), which is the potential static conflict savings of careful page mapping. While random page mapping would average C_{avg} conflicts, a careful mapping of the pages could reduce the conflicts close to $C_{\min}$. The figure shows that as many as 30% of the pages from an address space are unnecessarily in conflict with random mapping. This is the potential static improvement available to careful mapping.

Figure 6.5 shows that the mapping conflicts are maximized at the point where the address space size equals the cache size. This indicates that the largest potential gain from careful mapping is when the (active) address space size is about equal to the cache size. Other address spaces also will benefit from careful mapping, but this is the best case. When the cache is underutilized or highly utilized, careful mapping becomes less useful and necessary because it can remove fewer conflicts.

Figure 6.5 also shows that $C_{avg}-C_{\min}$ decreases with increasing associativity. This shows that careful mapping is most useful with a direct-mapped cache. In the extreme case, careful mapping cannot help fully-associative caches because they have no mapping conflicts.

Consider the peak values of the curves in Figure 6.5. This value is interesting because it shows the potential of careful mapping for a given cache. If a simple computation could determine this value, it would be an easy way to learn the maximum potential of careful mapping for a given cache.

Let $max(C_{avg}-C_{\min})$ be the value for $C_{avg}-C_{\min}$ when $U=N=B\times A$ and $P\to\infty$ (Equation 6.2 is used rather than Equation 6.1). It can be shown that

$$\lim_{B\to\infty}\frac{max(C_{avg}-C_{\min})}{N}=e^{-A}\sum_{u=0}^{A-1}(A-u)\frac{A^{u-1}}{u!} \tag{6.6}$$

where e is the natural exponential [ELLG82]. For direct-mapped caches, this is simply e^{-1}. This means

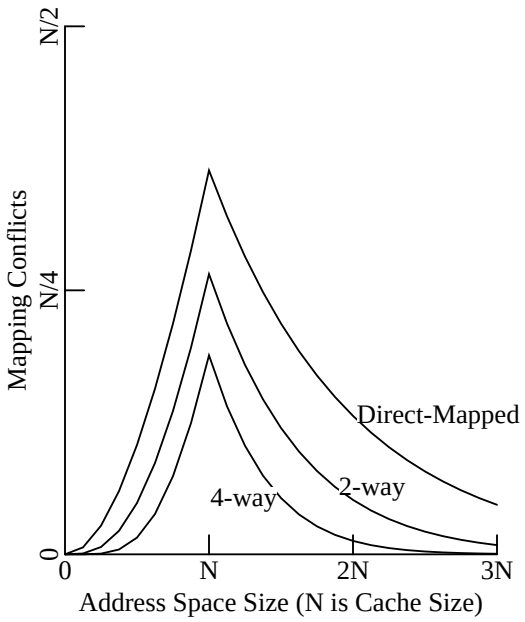


Figure 6.5. Random Mapping Conflicts.

This figure shows the difference between the conflicts resulting from a random mapping and the minimum conflicts, the mapping conflicts ($C_{avg} - C_{\min}$), for different address space sizes and cache associativities ($A = 1, 2, 4$). The configuration is a 128-megabyte main memory in 16-kilobyte pages with a 1-megabyte cache ($N = 64$, $B = N/A$).

that up to e^{-1}, or 36.7%, of pages can be unnecessarily in conflict in large caches when the main memory size is large. An empirically found closed-form approximation of Equation 6.6 is:

$$\frac{max\,(C_{avg} - C_{\min})}{N} \approx Ae^{-(\log_2(A)+1)}. \tag{6.7}$$

This formula is extremely simple, yet it contains much useful information. Table 6.1 lists actual maximum conflict fractions for various configurations, and compares them to the estimations given by Equation 6.6 and Equation 6.7.

In general, the approximations are accurate. Both Equation 6.6 and 6.7 consistently overestimate the actual maximum fraction of conflict pages over the range of caches studied. Equation 6.6 is slightly more accurate than Equation 6.7, the empirical equation. Both approximations are more accurate for the lower associativities.

Usually, the simple closed-form formula given by Equation 6.7 is a good approximation of the maximum average static mapping gain from careful mapping. Since it depends only on the associativity of the cache, it shows that the usefulness of careful mapping is more dependent on associativity than the cache size, page size, or main memory size.

6.3. Careful Mapping Implementations

This section presents four careful page mapping implementations that try to realize the potential improvements shown in the previous section. The careful mapping algorithms use any pool of available page frames to utilize the cache bins more fully and minimize the cache contention by the pages from multiple address spaces. They do not require a pool for correctness, but only to give better placements.

Maximum Conflict Fraction						
B	A	Actual	Equation 6.6		Equation 6.7	
			Value	Error	Value	Error
64	1	0.36	0.37	1.2%	0.37	1.2%
32	2	0.27	0.27	2.0%	0.27	2.0%
16	4	0.19	0.20	3.7%	0.20	5.7%
8	8	0.13	0.14	7.3%	0.15	12.7%
4	16	0.09	0.10	16.0%	0.11	26.0%
256	1	0.36	0.37	1.8%	0.37	1.8%
128	2	0.27	0.27	2.0%	0.27	2.0%
64	4	0.19	0.20	2.4%	0.20	4.4%
32	8	0.14	0.14	3.2%	0.15	8.3%
16	16	0.09	0.10	4.9%	0.11	14.0%

Table 6.1. Maximum Conflict Fraction Approximations.

This table compares the maximum fraction of mapping conflict pages approximations in Equation 6.6 and Equation 6.7 with actual values (as calculated by Equation 6.3 with $U = N = B \times A$). The ''actual'' data in this table corresponds to a 128-megabyte main memory with 16-kilobyte pages ($P = 8192$). The top half corresponds to a 1-megabyte cache and the bottom half corresponds to a 4-megabyte cache.

Of course, a careful page mapping algorithm should preferably produce better placements with a smaller pool. That way, careful mapping is more adaptable to all pools, small or large, and it can produce static mapping improvements with minimal page replacement effects (perhaps none).

The algorithms most importantly minimize intra-address-space cache contention, and only secondarily do some of them try to minimize inter-address-space contention[25]. They improve the page placement of a single address space in a (unified) multi-megabyte real-indexed cache based solely on the static criteria shown in the previous section; they do not use any dynamic referencing information for placement improvements[26]. Section 6.4 shows that the static placement improvements of these algorithms eliminate many dynamic cache misses.

6.3.1. Page Coloring

The simplest way to map pages carefully is to force a real-indexed cache to work as if it were a virtual-indexed cache. This is called *Page Coloring*. MIPS uses a variant of it [TADF90]. Page Coloring minimizes cache contention because successive virtual pages do not conflict in a virtual-indexed cache, and mostly contiguous address spaces that are smaller than the cache often index into cache bins without conflict. Given the complexity of current memory systems, the simplicity of Page Coloring is desirable. The page mapping function simply (equality) matches the bin bits of a virtual page to a real page frame. When it cannot find a page frame that matches the bits, it chooses any page frame, probably the first available. Figure 6.6 shows a code fragment that could be used to implement Page Coloring. The `Careful_Coloring()` function chooses the bin each time Page Coloring maps a page. Not shown are more difficult portions of the memory management code that would extract an available

25. Intra-address space conflicts are likely to cause the most cache misses in future processors that execute hundreds of thousands, or even millions, of instructions within a single address space before each process switch.

26. Static placement decisions are appropriate because dynamic information, such as cache references (misses) or page references (faults), may not be available to the page mapper.

page from the chosen bin and update system state information accordingly.

```
VALUE free[NUMBER_BINS];

BINTYPE Careful_Coloring(vp, pid)
VIRTUALPAGE vp;
PIDTYPE pid;
{
    BINTYPE bin;

    bin = vp % NUMBER_BINS;

    if(free[bin] > 0)
        return(bin);
    else
        return(BIN_OF_ANY_AVAILABLE_PAGE);
}
```

Figure 6.6. Bin Choice Code for Page Coloring.

This figure shows a ‘‘C’’ code fragment to implement Page Coloring. `Careful_Coloring()` places a page in a bin, given the virtual page, *vp*, to be mapped. The `free` array contains the number of page frames available in each bin. The ‘‘%’’ operator is the modulo operator. BIN_OF_ANY_AVAILABLE_PAGE would likely be the bin of the first available page, but could be the bin of any arbitrary available page.

More sophisticated Page Coloring implementations are better. As Figure 6.6 shows the implementation, Page Coloring places commonly used virtual addresses (like the stack, for instance) from different address spaces in the same real-indexed cache bins, which will lead to excessive inter-address space contention and, consequently, many cache misses. Page Coloring can solve this problem by offsetting each address space (or contiguous segment) differently in the cache [TADF90], instead of directly matching virtual and real bin bits. This chapter considers two versions of Page Coloring: (1) an exact match of bin bits (as shown in Figure 6.6), and (2) a match of bin bits exclusive-ored with a different PID (process identifier) for each address space. The precise form of Page Coloring used by MIPS was not known, but they probably don’t use a direct equality bin match.

6.3.2. Bin Hopping

Figure 6.7 shows Bin Hopping, another simple way to equalize cache bin utilization. It allocates sequentially mapped pages in sequential cache bins, irrespective of their virtual addresses. Bin Hopping reduces cache contention because it sequentially distributes the pages from an address space across the cache in different bins (until it must wrap around). It exploits temporal locality because the pages it maps close in time tend to be placed in different bins. If pages mapped together are also referenced together, this reduces contention.

Figure 6.7 shows a Bin Hopping example where it always finds available page frames in successive bins. This may not be a common scenario in many systems since there may not be page frames available for replacement in all bins. Bin Hopping skips some bins when it increments over bins that do not contain free page frames. This degrades the page placement.

Figure 6.8 shows a straight-forward ‘‘C’’ code Bin Hopping implementation that chooses the bin placement of a page. The implementation traverses the bins until it finds one with an available page frame. Then it remembers where it placed the last page, so the next traversal can start from there. Bin

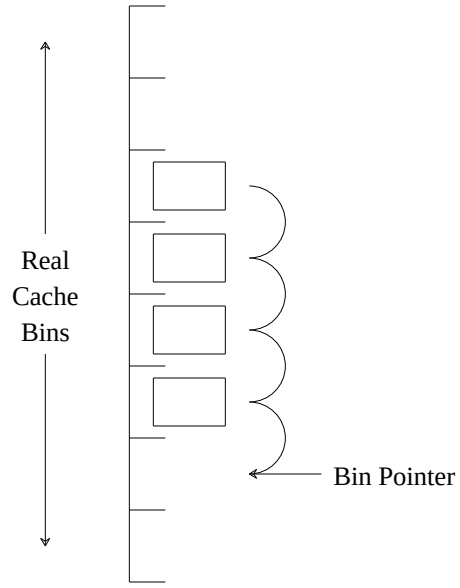


Figure 6.7. Page Placement By Bin Hopping.

This figure pictures a page placement that could occur with Bin Hopping, which maps pages to page frames in successive cache bins.

Hopping maintains a separate traversal path per address space, so it minimizes conflicts within each address space separately. The simplicity of Bin Hopping, like Page Coloring, is its largest asset. A small amount of state information is needed per address space, only the bin where the last page was placed.

6.3.3. Best Bin

The strength of Page Coloring and Bin Hopping may also be their weakness. The small amount of saved state information reduces the effectiveness of the mapping function because it precludes sophisticated decision-making. When there are not available pages in all bins, Page Coloring and Bin Hopping may not produce a good mapping. Instead, a page mapping function can use to advantage information indicating the previously placed pages in each cache bin. The complete mapping of virtual pages to page frames contains this information, but it can be costly to extract it. Since the page mapping function requires only counts of page frames in each bin, execution efficiency is greatly improved by storing these counts in a readily-available form. The storage space of the counts is modest compared to the space required for the complete page mapping information.

The count for each bin is the pair <*used, free*>. *Used* is the number of previously placed pages from a given address space, and *free* is the number of page frames available for placement in this bin. A straight-forward mapping algorithm sequentially looks at each of the <*used, free*> pairs (for each bin) and chooses the bin that best meets the standards of suitability. This algorithm is *Best Bin*. Like Bin Hopping, Best Bin ignores the virtual addresses when it maps a page. Best Bin maintains system-wide *free* information to show the number of available page frames in each bin, just as Page Coloring and Bin Hopping do. The added state of Best Bin is the *used* information per bin for each active address space. Best Bin reduces contention within each single address space by equalizing these *used* values across the different cache bins.

```
VALUE free[NUMBER_BINS];
BINTYPE next_bin[NUMBER_PIDS];

BINTYPE Careful_Hopping(vp, pid)
VIRTUALPAGE vp;
PIDTYPE pid;
{
    BINTYPE bin;

    bin = next_bin[pid];

    while(free[bin] == 0)
        bin = (bin + 1) % NUMBER_BINS;

    next_bin[pid] = (bin + 1) % NUMBER_BINS;
    return(bin);
}
```

Figure 6.8. Bin Choice Code for Bin Hopping.

This figure shows a ‘‘C’’ code fragment to implement Bin Hopping. `Careful_Hopping()` places page `vp` in a bin, given the process (or address space) identifier `pid` indicating the mapping process (address space). It will properly terminate provided there is an available page frame. The `free` array contains the number of available page frames in each bin and the `next_bin` array contains the bin position (initially random) for each process (address space).

The `Choice()` code in Figure 6.9 ranks two bins based on the <*used*, *free*> parameters. The specific rules used for ranking the <*used*, *free*> pairs (bins) are, in priority order: (1) the bin must have at least one free page frame (*free* ≥ 1), (2) the bin should have the fewest pages used by this address space (minimize *used*), and (3) the bin should have the most page frames available (maximize *free*). Rule (2) minimizes conflicts within the address space, and rule (3) minimizes conflicts between different address spaces since more available page frames implies that other address spaces underutilize the bin. If these rules fail to produce a preferable bin, an arbitrary choice can be made.

Figure 6.10 shows ‘‘C’’ code to implement Best Bin. The code is a loop calling the `Choice()` function that incrementally finds the bin that is best. This is a simple implementation, and it assumes that there is at least one available page frame.

Figure 6.11 depicts an example of a choice made by Best Bin. The <0, 0> bin has the least previously mapped pages, but it cannot be chosen since there are no available page frames. Of all the <*used*, *free*> pairs, Best Bin chooses the <1, 3> bin because there are several available page frames and only one previously placed page frame. After choosing the <1, 3> bin, the page mapping function extracts an available page frame from the bin, maps the virtual page to the real page frame, and changes the <1, 3> pair to <2, 2> to reflect the modified state.

Best Bin requires more memory than the previous mapping algorithms. It requires less storage with increasing associativity or page size, but more storage with larger caches. Since the number of page frames from each bin is likely to be small and exact *used* values are not required for correctness, short integers can be used to represent each value. An 8-bit byte may be sufficiently large, for example. Variable overflow on short integers can be easily handled by ‘‘pinning’’ a node at the maximum value, making the value at a node an approximation of the true value. When using one byte per entry, only 256-bytes would be required for a *used* array corresponding to the bins of a 4-megabyte direct-mapped

```
BOOLEAN Choice(left_used, left_free, right_used, right_free)
VALUE left_used, left_free, right_used, right_free;
{
    if(left_free == 0)
        return(CHOOSE_RIGHT);
    else if(right_free == 0)
        return(CHOOSE_LEFT);
    else if(right_used > left_used)
        return(CHOOSE_LEFT);
    else if(left_used > right_used)
        return(CHOOSE_RIGHT);
    else if(left_free > right_free)
        return(CHOOSE_LEFT);
    else if(right_free > left_free)
        return(CHOOSE_RIGHT);
    else
        return(ARBITRARY_CHOICE);
}
```

Figure 6.9. Code to Choose Among Bins.

This figure shows a ‘‘C’’ code function used rank alternative bins. Its arguments are the <*used*, *free*> pairs for each bin and it returns the preferred bin. `ARBITRARY_CHOICE` could be a constant or a random variable.

```
VALUE used[NUMBER_PROCESSES][NUMBER_BINS];
VALUE free[NUMBER_BINS];

BINTYPE Careful_Best(vp, pid)
VIRTUALPAGE vp;
PIDTYPE pid;
{
    BINTYPE best_bin, bin;

    best_bin = 0;
    for(bin = 1; bin < NUMBER_BINS; bin++)
        if(Choice(used[pid][best_bin],
                  free[best_bin],
                  used[pid][bin],
                  free[bin]) == CHOOSE_RIGHT)
            best_bin = bin;

    return(best_bin);
}
```

Figure 6.10. Bin Choice Code for Best Bin.

This Figure shows a ‘‘C’’ code fragment to implement Best Bin page mapping. `Careful_Best()` places page `vp` in a bin, given the process (or address space) identifier of the mapping process (address space). The `used` array contains the *used* state information for each process. The `free` array contains the number of available page frames in each bin. Figure 6.9 defines the `Choice()` function.

cache with 16-kilobyte pages.

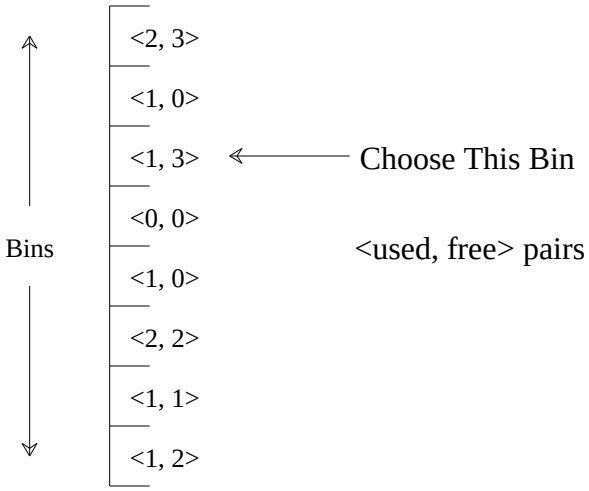


Figure 6.11. Page Placement by Best Bin.

This figure shows the bin that would be chosen as then best bin by Best Bin using `Choice()` from Figure 6.9. Pictured are the *<used, free>* pairs for each bin. Best Bin will look at all bins and decide that the <1, 3> bin is best.

6.3.4. Hierarchical

The Best Bin execution time may be linear in the number of bins, which may be too slow when there are many bins. Alternatively, Hierarchical can choose a bin more efficiently with the aid of a bin tree structure, as pictured in Figure 6.12. The time complexity of Hierarchical is logarithmic in the number of bins, much better than linear when there are many bins. Another important property of Hierarchical is its size independence. This means that it improves the static page placement in many caches *simultaneously*. With the proper association of tree leaves to cache bins, the different levels of the tree correspond to caches with more or less bins. By optimizing each tree level, Hierarchical improves the static page placement in each cache. This is particularly important with a hierarchy of real-indexed CPU caches since the page mapping function will eliminate conflicts in all the caches in the hierarchy simultaneously. It also simplifies software management. For example, it allows machines with different caches to use the same operating system executable; the same algorithm will eliminate conflicts in each cache without modification. Size independence also has a simulation advantage: a cache simulator can concurrently simulate multiple caches with the same size-independent virtual to real page mappings produced by Hierarchical.

Figure 6.12 shows that a bin tree is a fully balanced binary tree; the value of each leaf node is the number of page frames in its associated bin. The value of interior nodes is the sum of the values of its children. The value of the root node of a used (free) bin tree is the number of used (free) page frames. Hierarchical uses a single free bin tree system-wide, but it needs a used bin tree for each address space, just as Best Bin needs a bin array per address space and Bin Hopping needs a bin position for each address space. This chapter uses binary trees; higher branching factors also could be used, but they may not maintain size independence with all caches.

When Hierarchical maps a new virtual page to a page frame, it traverses the bin trees from the root to a bin (leaf) at the bottom of the trees. During a traversal, exactly $\log_2(B)$ choices decide the particular traversal path. The key determinant of the effectiveness of Hierarchical is the quality of these

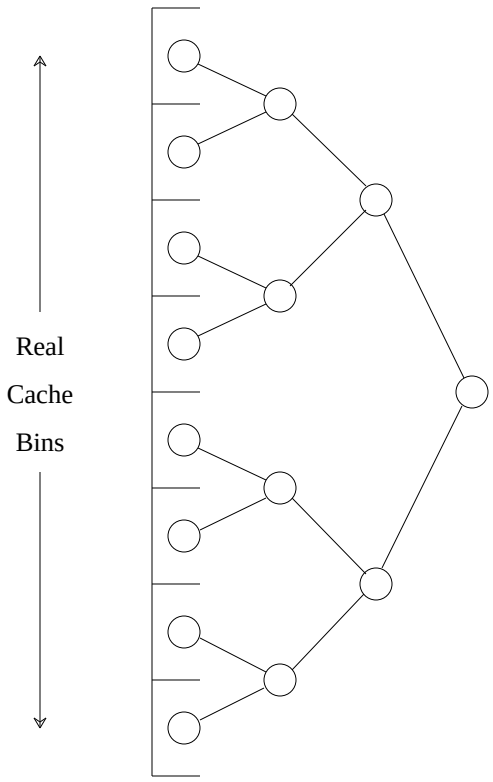


Figure 6.12. Page Placement Using Hierarchy.

This figure pictures cache bins structured as a hierarchical tree, called a bin tree. A tree traversal places the page in a bin. Decisions at each node, based on the node values, determine the tree traversal direction. This chapter assumes binary bin trees; trees with higher branching factors also could be used.

decisions. Hierarchical decides the traversal direction by looking at the <*used*, *free*> pairs corresponding to the two children of a node. It ranks bins (sub-trees) precisely as Best Bin does, as described in Figure 6.9. The <*used*, *free*> values passed to the `Choice()` function are interior tree values instead of the final bin values. Both Hierarchical and Best Bin use the same `Choice()` because the goals of the decision are the same in each case. They both minimize contention by equalizing the *used* values within each address space separately.

Figure 6.13 shows Hierarchical on a simple example. Traversal starts from the root of the bin trees. From there, Hierarchical decides based on the <4, 6> and <5, 5> pairs corresponding to its sub-trees. It chooses <4, 6> since it has less *used* pages. Next, Hierarchical chooses <1, 3> over <3, 3>, again because there are less *used* pages. Finally, it chooses <1, 3> over <0, 0> because <1, 3> has available page frames. Bin tree traversal is then complete since a leaf node is reached.

Figure 6.13 also shows how Hierarchical updates bin trees after placing the page in the chosen bin. Tree update, unlike tree traversal, proceeds from a leaf of the tree through the only path to the root of the tree, updating each node along the way. Similar to the $\log_2(B)$ decisions required to traverse a bin tree downward, Hierarchical modifies $\log_2(B)+1$ nodes during a tree update. Note that tree update could be done with tree traversal since Hierarchical knows the changes it will make on the path to the leaf.

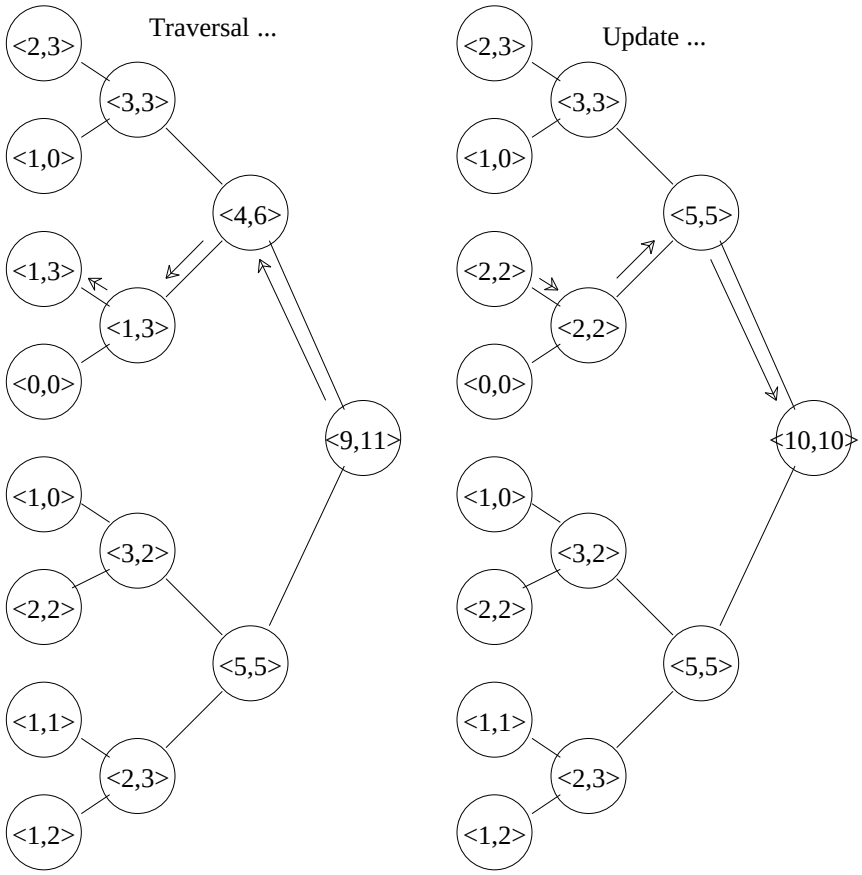


Figure 6.13. Bin Tree Traversal.

This figure shows an example of a bin choice when mapping a virtual page (using the decision heuristic in Figure 6.9) by <*used*, *free*> bin tree traversal with Hierarchical. The right half shows the corresponding updates that occur after the mapping traversal. The values of the tree nodes are given. The used values of the leaves show the number of pages already placed in the corresponding bin. The free values of the leaves show the number of page frames available for replacement in the corresponding bin.

Figure 6.14 illustrates the labeling of tree nodes in a bit-reversed manner. With this labeling, Hierarchical produces size independent mappings. The labels of the tree nodes are a reversal of the bit strings obtained from an increasing (binary) ordering of the nodes on a level from left to right. By definition, the value of a leaf node is the number of *used* (*free*) page frames whose bottom bits match the leaf node label. Since a parent node is the sum of its children values, Appendix A shows that the value of each tree node is the number of page frames whose bottom bits match the node label.

The low-order (right) digit of the nodes on the left half of the bin tree are all zero while the corresponding bit is always one for the nodes on the right half. Since the first decision during mapping decides whether the right or left sub-tree will be traversed, this decision chooses the bottom bit of the page frame number. This is exactly the conflict elimination decision required for a cache with two bins. Hierarchical reduces conflicts in a cache with two bins although the tree in Figure 6.14 is specifically targeted for a cache with eight bins (since there are eight leaves). The essence of cache size independence is that when eliminating conflicts in a cache with more bins, conflicts must also be eliminated in smaller caches.

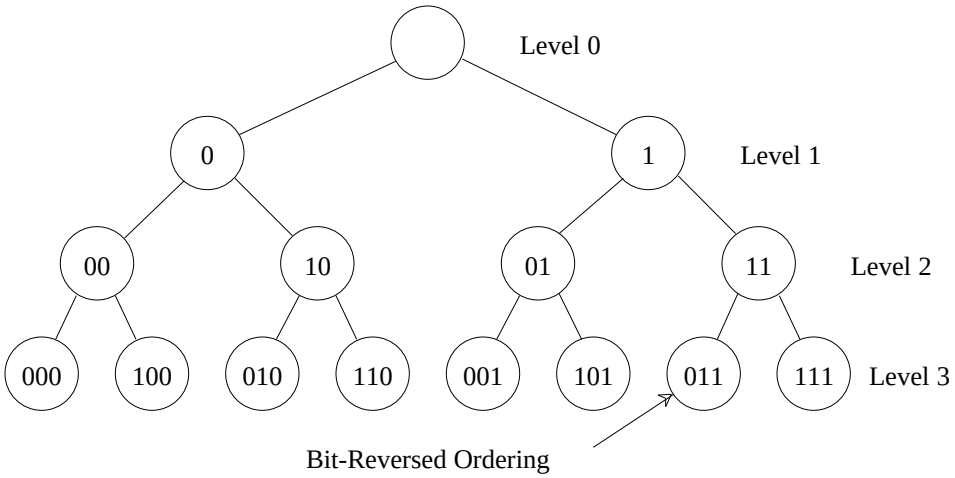


Figure 6.14. Mapping Tree Nodes to Bins.

This figure shows a relationship of bin numbers to tree nodes that allows Hierarchical to produce size independent mappings. It is the bit-reversal of numbering the nodes with labels in increasing order from left to right.

Bin trees labeled as in Figure 6.14 are called a Bit-Reversed Bin Tree (BRBT). As well as the number of *bin* bits, the depth of a cache is the BRBT level with the same number of nodes as there are bins in the cache. With a BRBT, Hierarchical improves the static page placement in all caches with depth less than or equal to the depth of the tree. Hierarchical makes the same choices at the upper levels of the bin tree whether the lower levels of the BRBT are filled out or not. To maintain size independence, the lower levels only refine the decisions made at higher levels. The mappings produced by Hierarchical are equivalent with different tree sizes provided the choice among multiple free page frames within a bin is arbitrary. Appendix A gives a formal proof that Hierarchical produces size independent mappings.

The implementation of a BRBT is efficient since the trees are fully balanced binary trees that can be represented using a simple array as a *heap* (as used by *heapsort*) [AHHU85] with only $2B-1$ entries required, indexed from 1 to $2B-1$. Each node is identified by its index into the array, the first entry is the root node. The content of the corresponding array entry is the node value. The power of the heap representation is that the parents and children of a node can easily be found by performing bit shift and logical operations on the node identifier. The left child of the node identified by i is $2i$ and the right child is $2i+1$. The parent of node i is $\lfloor i/2 \rfloor$. The implementation of these operations on a binary machine can be particularly efficient.

Figure 6.15 shows code to implement Hierarchical, which uses the decision heuristic in Figure 6.9 and a heap representation of the used and free BRBT's. Hierarchical requires only twice the memory of Best Bin. 511 nodes per address space (process) would be required for the *used* BRBT of a 4-megabyte direct-mapped cache with 16-kilobyte pages. Similar to Best Bin, each node requires little storage since its value need not be exact.

Though not explored further in this study, Best Bin and Hierarchical could be merged into a single algorithm. This algorithm would combine the speed of Hierarchical with the better decisions of Best Bin. Best Bin could be used to find the best bin at a certain level of the BRBT, then Hierarchical could further traverse the sub-tree of the BRBT chosen by Best Bin. For example, in a BRBT with 11 levels

```
VALUE used[NUMBER_PROCESSES][2 * NUMBER_BINS];
VALUE free[2 * NUMBER_BINS];

BINTYPE Careful_Hierarchical(vp, pid)
VIRTUALPAGE vp;
PIDTYPE pid;
{
    int level;
    NODE_IDENTIFIER node, left_node, right_node;
    BINTYPE bin;

    level = 1;
    node = 1; /* root node */

    while(level < NUMBER_TREE_LEVELS)
    {
        left_node = 2 * node;
        right_node = 2 * node + 1;
        if(Choice(used[pid][left_node],
                        free[left_node],
                        used[pid][right_node],
                        free[right_node]) == CHOOSE_LEFT)
            node = left_node;
        else
            node = right_node;
        level = level + 1;
    }

    bin = Bit_Reverse(node - NUMBER_BINS, NUMBER_BIN_BITS);
    return(bin);
}
```

Figure 6.15. Bin Choice Code for Hierarchical.

This figure shows a ‘‘C’’ code fragment to implement Hierarchical. `Careful_Hierarchical()` places the page `vp` in a bin, given the process (address space) identifier of the process (address space) requesting the mapping. The `used` array contains the used bin tree for each process and the `free` array holds the system free bin tree, each tree is stored as a heap. Figure 6.9 defines the `Choice()` function. The `Bit_Reverse()` function reverses the bits of its first argument (the second argument gives the number of reverses) so that the mapping of bins to leaves maintains size independence.

(of depth 10), Best Bin could choose the best of the eight sub-trees at level three in the tree. Hierarchical could then refine the decision made by Best Bin to level ten in the tree. This example algorithm could be particularly useful for a system with hierarchical caches of eight and 1024 bins each; Best Bin would give good mappings for the cache of eight bins, while the mappings would be size independent for this configuration because Hierarchical was used at the bottom of the tree.

6.4. Trace-Driven Simulation Performance Analysis

This section analyzes the careful page mapping algorithms described in the previous section using trace-driven simulation. Trace-driven simulation allows the usefulness of the different careful page mapping algorithms to be measured. This analysis uses the traces described in Chapter 3. It shows that careful static page placement improves dynamic cache performance.

6.4.1. The Simulator

Previous trace-driven simulation results analyzing cache performance assume that operating system policies are fixed and independent of the studied caches. This chapter eliminates this assumption. Since this section examines different virtual page to physical page frame mapping policies, the simulator incorporates memory management policies of the operating system along with hardware cache management implementations. The simulation results expose the interaction between hardware and software system components.

The simulator implements a global least-recently-used (LRU) page replacement policy by maintaining an exact ordering of the page frames from most recently used to least recently used (an LRU list). (It does not place pages in the pool when a process exits, only when they are least-recently-used.) The simulator maps a virtual page to a corresponding page frame at the point when the page is first referenced (demand loading). Though exact LRU may be infeasible to implement in practice, it is a replacement policy with desirable properties that is feasible to implement in a simulation environment. The page frame at the bottom of the LRU list, the least recently used one, would normally be the best candidate page frame for replacement. Rather than requiring that the simulator choose the last page on the LRU list for mapping, the careful page mapping policies have the freedom to choose among the page frames near the bottom of the LRU list. That is, the pool of available page frames consists of the frames at the bottom of the LRU list. Figure 6.16 shows this. The size of the available page frame pool is constant throughout the simulations of this chapter, even at the beginning of a simulation when the entire main memory is unmapped. The default pool size is 4-megabytes of the 128-megabyte main memory in 16-kilobyte pages. The default main memory size is large enough that the simulation required no process swapping, and the static process scheduling on each trace is used.

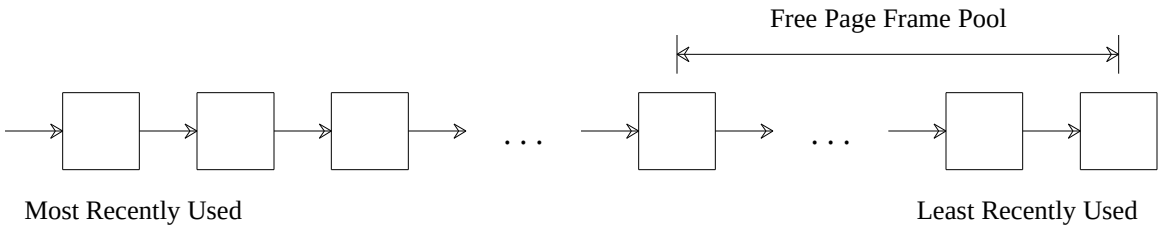


Figure 6.16. LRU List Implementation With Available Page Frames.

This figure shows the global LRU list of page frames and the free page frame pool. The simulator maintains an exact list of page frames from most recently used to least recently used during real-indexed cache simulations. When it maps a virtual page to a page frame, the bottom page frames on the list are available for mapping.

Before starting each real-indexed cache simulation, the simulator initializes the LRU list in a different random order. This models a system that does not differentiate among its page frames and has been executing for some time; eventually page frames are not in any significant order. Given this random ordering, the mapping algorithms have a random sampling of page frames available for mapping at the beginning of a simulation. As execution continues, the memory reference pattern of the traces and the previous mapping decisions determine the available page frames.

6.4.2. Results

6.4.2.1. The Usefulness of Careful Page Mapping

This section compares Hierarchical mapping to random mapping because Hierarchical's size-independence reduces simulation time, and because Hierarchical performs well compared to the other careful mapping schemes in Section 6.4.2.3. For 1-megabyte (top), 4-megabyte, and 16-megabyte caches (bottom), figure 6.17 plots 100 million instruction averages of the Mult2.2 misses per instruction (*MPI*) for four different simulations, each with a different virtual to real page mapping. The cache performance is different because each simulation has a different pool of available page frames, so each simulation places the same pages in different page frames, and consequently different cache bins. Note that random mapping *does not* imply random page replacement; it means that the simulation places the page in the page frame at the bottom of the LRU list. Thus, the simulator places the page in a random (arbitrary) bin, depending only on the page frame number at the bottom of the LRU list.

Figure 6.17 shows, for the Mult2.2 trace, that the careful (Hierarchical) mappings have consistently lower *MPI* than random (since the solid lines are usually below the dotted lines). This shows that the static page placement improvements of careful mapping reduce dynamic cache misses for this trace. Over most of the trace, the careful mapping *MPI*'s are only a few percent better than random. But in addition to this small and constant improvement, there are bursts of cache misses from poor random page mapping that occasionally double or triple the *MPI* for the 100 million instruction intervals[27]. These bursts increase the mean improvement of careful mapping well beyond only a few percent.

To get a better idea of the quantitative dynamic cache performance improvements of careful mapping, the average cache *MPI* when using random and Hierarchical mapping should be determined. To do this, the results from multiple simulation runs that use different page mappings were averaged. While the *MPI* of one simulation is an unbiased estimate of the true mean *MPI*, Figure 6.17 suggests that the *MPI* results from simulations can substantially vary, so it is hard to say how close one simulation result is to the true mean *MPI*. After running four simulations, most of the four-sample means were found to be accurate (with 90% confidence) within a few percent, so no more simulations were run. Fortunately, each trace was long enough so that the individual simulations averaged out much of the variance in cache performance, and too many long-trace simulations were not needed (which is important since each simulation takes days). With shorter traces, many more samples would have been required for the same statistical significance. Appendix B is provided for those who wish to check the statistical significance of the results. This is particularly important given that the sample size is so small. This dissertation chapter uses the four-sample means to quantitatively evaluate the improvement of careful mapping for 1-megabyte, 4-megabyte, and 16-megabyte secondary caches with direct-mapped, 2-way, and 4-way set-associativity.

Figure 6.18 shows the direct-mapped Hierarchical miss reductions. The Hierarchical reductions are the relative amount that the four-sample mean Hierarchical *MPI* is lower than the four-sample mean random mapping *MPI*. The figure shows that careful page mapping is useful for a variety of workloads. Careful mappings eliminate up to 55% of the direct-mapped cache misses from random mapping. On

27. Note that some bursts appear in the random mappings of both the 1-megabyte and 4-megabyte cache simulations. This occurs since the simulations of different cache sizes are of the same virtual to real page mapping, and the results are correlated. The results for each individual cache are not correlated, however.

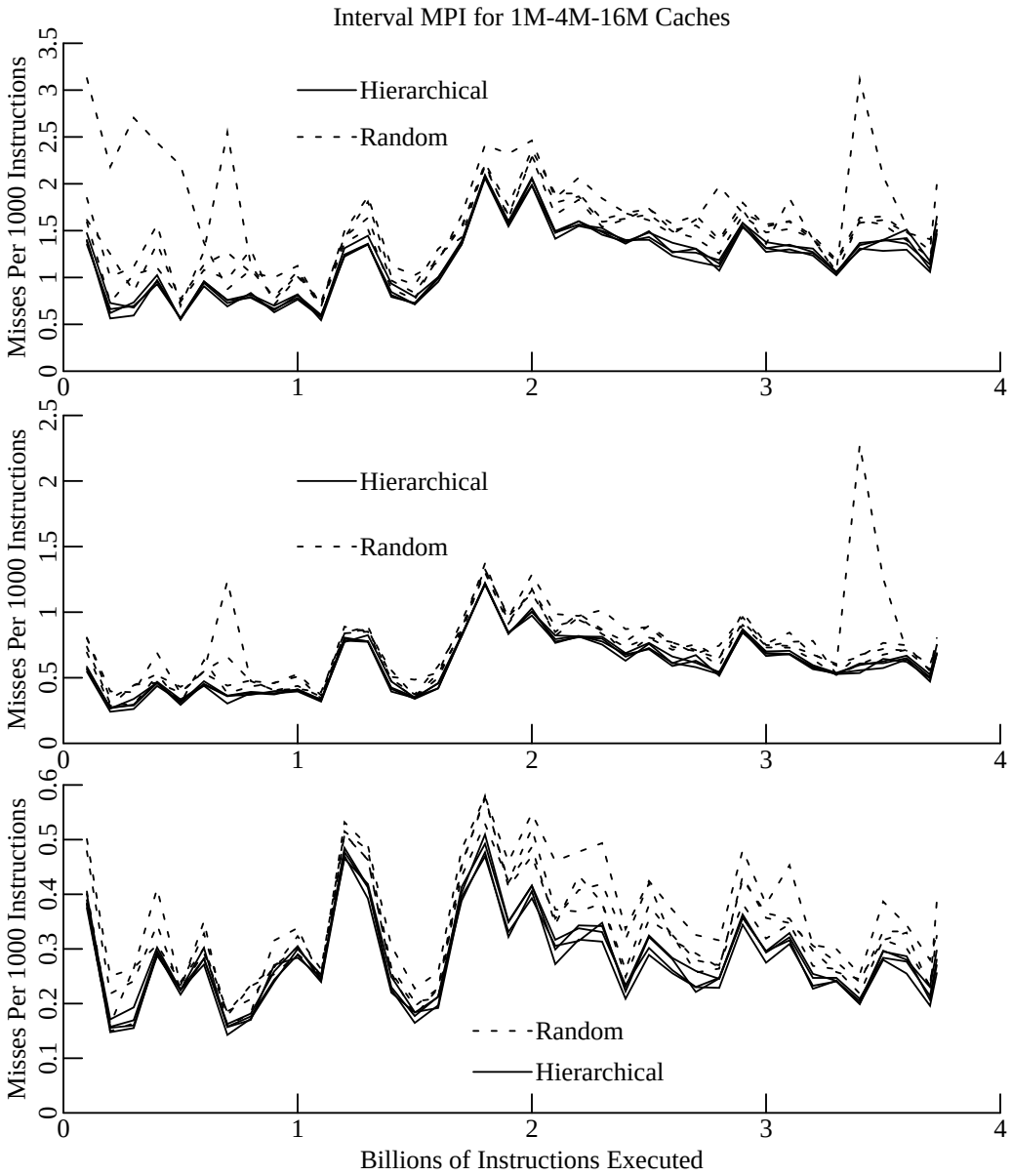


Figure 6.17. Mult2.2 Results for Different Page Mappings.

This figure shows the *MPI* effect of four different virtual to real mappings generated by Hierarchical (solid lines) and generated randomly (dotted lines) on the misses of three real-indexed secondary caches. It shows the average *MPI* for the previous 100 million instructions of the Mult2.2 trace for 1-megabyte (top), 4-megabyte (middle), and 16-megabyte (bottom) secondary direct-mapped caches. Note that the scales are different for each graph.

the average over all the traces, careful mapping eliminates about 10-20% of the random-mapping direct-mapped cache misses. A similar look at the results for the other associativities suggests that careful mapping eliminates about 4-10% of the misses in a 2-way set-associative cache with random replacement, and about 2-5% of the misses in a 4-way set-associative cache with random replacement (on average). (The Hierarchical reductions for the higher associativities are given in Appendix B.) These results verify that the largest gain from careful mapping comes with a direct-mapped cache, as

predicted by the simple static analysis of Section 6.2.2. Higher set-associativities can more easily eliminate contention without static placement improvements because they spread contending blocks across the cache block frames in a set. Static improvements that avoid contention are more valuable with direct-mapping because direct-mapped caches cannot eliminate contention for the same set (block frame).

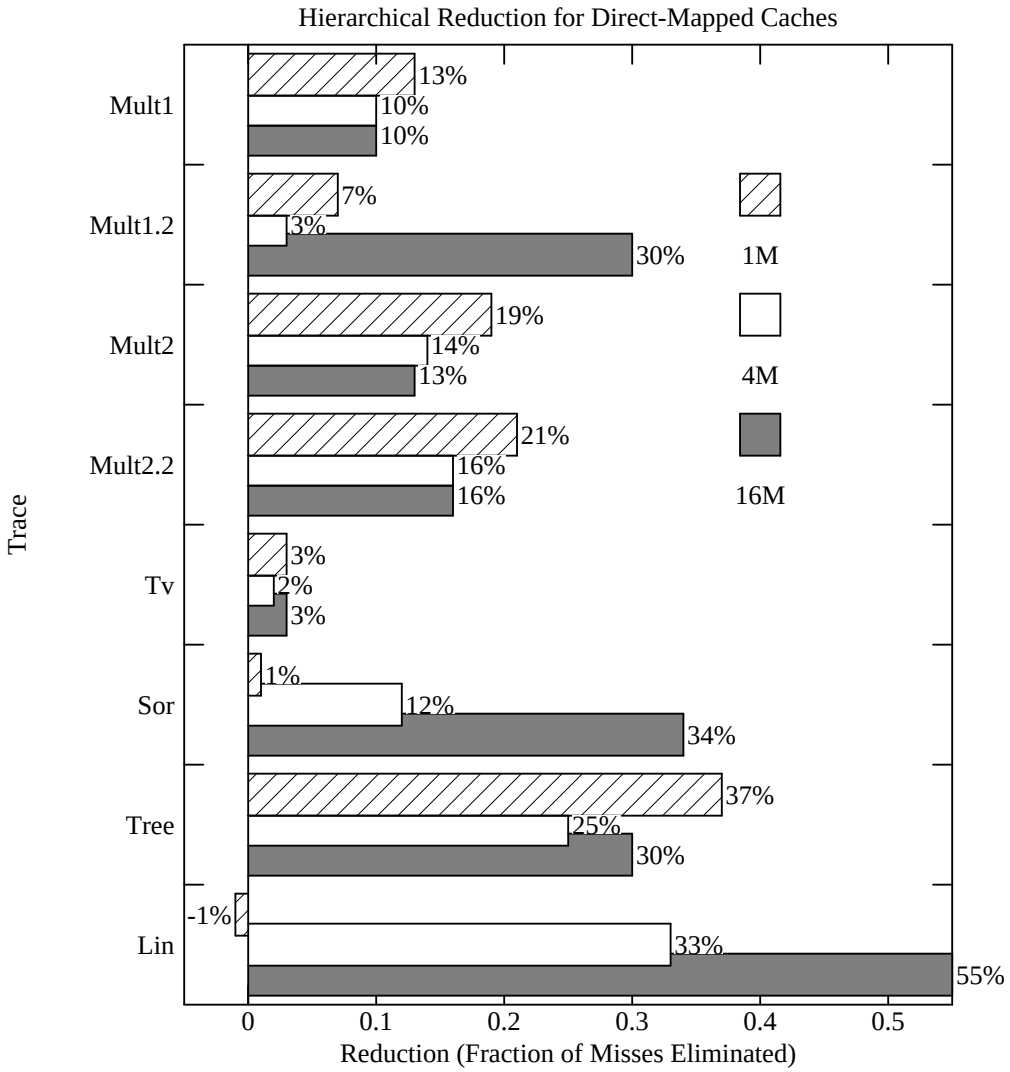


Figure 6.18. Direct-Mapped Hierarchical Miss Reductions for All Traces.

This figure shows, for different direct-mapped secondary caches and traces, the Hierarchical reduction, which is the fraction of the misses produced from random mapping that the careful mapping of Hierarchical eliminated.

The direct-mapped Hierarchical miss reduction is positive in all cases except once with Lin, where the *MPI* slightly increases with careful mapping. The multiprogrammed trace behaviors are similar, except for Mult1.2, where one bad page placement in one simulation biased the results and cut the direct-mapped Hierarchical reduction in about half (for the 1-megabyte and 4-megabyte caches). Careful mapping doesn't help Tv much because most of its misses occur while it traverses a large data

structure with little locality; careful mapping does not eliminate many misses during this traversal because the memory locations tend to be randomly accessed, and the Tv address space is too large for the static placement improvements to eliminate much contention. (Figure 6.5 shows how there is little room for static mapping improvements when the active address space size is large relative to the cache size.) The improvements of both Sor and Lin relate closely to the benefits of associativity. There is little (or no) careful mapping (associativity) improvement with the 1-megabyte cache, but careful mapping (associativity) improves performance greatly for the 16-megabyte cache because some arrays fit fully in the cache. Careful mapping also gets much of the associativity benefits with Tree, averaging about a 30% miss reduction. Though the results in Figure 6.18 vary for different traces, most of the rest of this chapter examines Mult2.2 in detail because of its ‘‘average’’ behavior; the Mult2.2 Hierarchical reductions are close to the averages for all the traces.

Figure 6.19 shows the Mult2.2 results in another graphical way to put the careful mapping miss reduction in its proper perspective. By interpolating between the simulation results for different cache sizes and associativities, it estimates the effect of software careful mapping relative to hardware cache design changes. The dashed line in the left graph finds the equivalent associativity change of Hierarchical with a 1-megabyte direct-mapped cache. Careful mapping eliminates more than half the misses of a random-mapping associativity increase; it is about equivalent to an associativity increase from direct-mapped to 1.6-way with Mult2.2. For a direct-mapped cache, the dashed line in the right graph compares the careful mapping miss reduction (with a 1-megabyte cache) to the reduction from a randomly-mapped cache size increase. Careful mapping also eliminates a large portion of the misses eliminated by a direct-mapped cache size doubling; it makes a 1-megabyte direct-mapped cache act about like a 1.7-megabyte direct-mapped cache for Mult2.2.

Most importantly, careful page mapping reduces mean *MPI* (across different page mappings) by avoiding poor page mappings. Also, a secondarily useful trait of careful mapping is a reduction in the *MPI* variance for different runs of a trace. Variance in cache behavior is undesirable because it increases the execution time variance of a workload. Figure 6.17 shows that the Mult2.2 mappings produced carefully have less *MPI* variations than the random mappings because all of the solid (Hierarchical) lines are much closer together than the dotted (random) lines. The different careful mapping simulations tend to follow an outline that the workload dictates, but the random mapping simulations show the bursts that are caused by random contention for cache locations. Poor random mappings into the cache, not the workload references themself, cause many of the random mapping misses.

Although careful mappings eliminate much of the variance in cache performance, they don’t remove all of it. In more rare circumstances than with random, a careful mapping implementation also decides poorly. Figure 6.20 shows simulation results of mappings produced randomly and by Hierarchical for the Mult1.2 trace. The one Hierarchical mapping (a solid line) shown in Figure 6.20 caused considerably more cache misses over the first half of Mult1.2, but its misses returned near that of the other Hierarchical simulations for the second half of the trace. This is the bad simulation that biased the Hierarchical reduction of Mult1.2.

Bad decisions near the start of the trace caused more contention over the phase of execution of the process that referenced the poorly mapped pages. Once this phase completed, the poor mapping decisions no longer affected the cache misses. This emphasizes the importance of the page mapping function choosing a ‘‘good’’ mapping. Though the page may only be mapped once, the cost of a poor mapping can occur on every reference to the page.

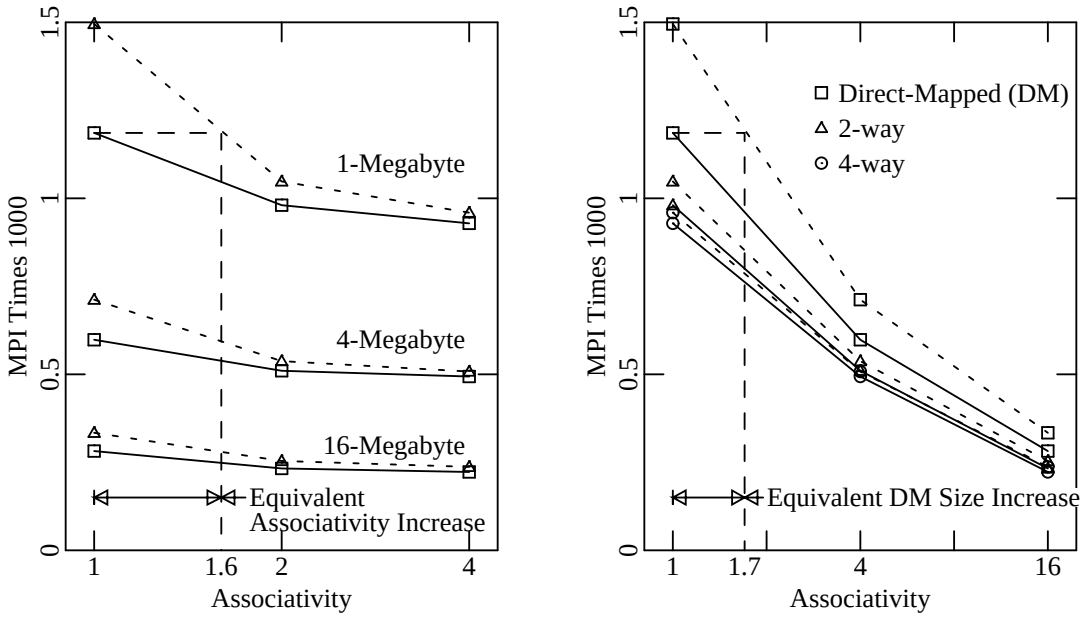


Figure 6.19. Mult2.2 Careful Mapping Improvement.

This figure shows the (mean) Mult2.2 simulation data from Appendix B. The misses per thousand instructions is plotted versus the cache associativity and size for random (dotted) page mappings and Hierarchical (solid) page mappings. The left graph shows *MPI* versus associativity for the different cache sizes. The right graph shows *MPI* versus cache size for different associativities (square = direct-mapped, triangle = 2-way, and circle = 4-way)

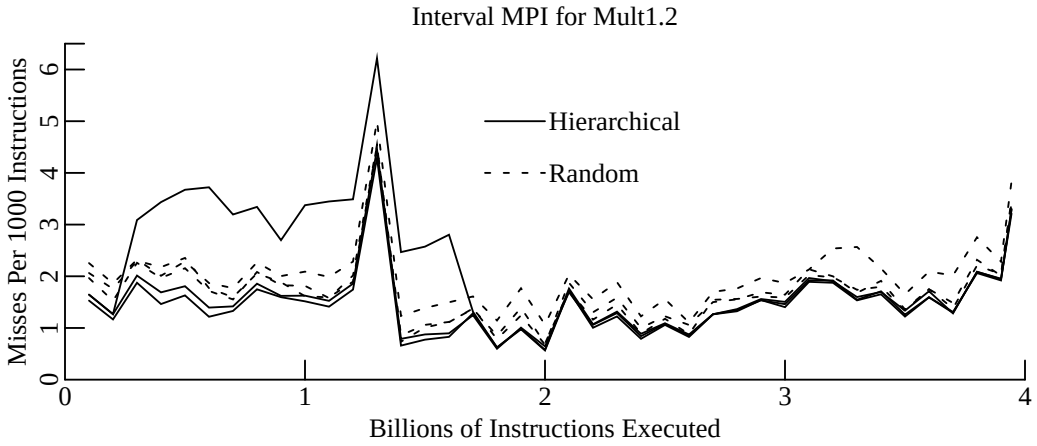


Figure 6.20. Mult1.2 Results for Different Page Mappings.

This figure shows the *MPI* effect of three different virtual to real mappings generated randomly (dotted lines) and by Hierarchical (solid lines) on the *MPI* of a real-indexed secondary cache. It shows the average *MPI* for the previous 100 million instructions of the Mult1.2 trace for 1-megabyte direct-mapped cache.

The four-sample results for all the traces verify that careful mapping reduces the *MPI* variance for a number of workloads[28]. When Hierarchical eliminates the most misses, it also tends to eliminate the

28. Appendix B indirectly gives the *MPI* variance of the four simulations because the widths of the

most variance. This isn't surprising since most of the *MPI* reduction comes from eliminating bursts of misses; these bursts increase both the mean and variance of the *MPI* across different simulation runs. Hierarchical has a lower *MPI* variance than random mapping in eighteen out of the twenty-four direct-mapped cases, nineteen out of the twenty-four 2-way set-associative cases, and twenty out of the twenty-four 4-way set-associative cases. Except for once with Tree, only Mult1.2, Tv, and Sor have Hierarchical variances that are larger than the random *MPI* variances. For Mult1.2, one bad Hierarchical page placement in one simulation causes the occasionally higher four-sample Mult1.2 variance. For Tv, the Hierarchical variance is occasionally higher for the same reason that the Tv Hierarchical reduction is not very good: Hierarchical couldn't help Tv's cache performance much because Tv's active address space is much larger than the cache. For Sor, the variance is slightly higher with some of the smaller secondary caches because Hierarchical couldn't fit some of the Sor arrays into the cache. Hierarchical eliminates a lot of variance over all the traces and cache configurations. With fourteen out of the twenty-four direct-mapped caches, the random mapping four-sample variance is more than five times larger than Hierarchical. This suggests that Hierarchical will yield more predictable execution times than random mapping.

6.4.2.2. Static Page Conflict Minimization

To understand more fully the causes of the *MPI* improvement from careful mapping, Figure 6.21 shows the differences in the number of page conflicts (within single address spaces) produced randomly and by Hierarchical for Mult2.2. Each point shows the mapping conflicts ($C - C_{\min}$) from the address space of one process in the Mult2.2 trace for a 1-megabyte direct-mapped cache. The Mult2.2 trace contains the execution of hundreds of processes (since many of them completed), so many points were available. The squares, the sampled values produced by a random mapping, follow closely the predicted outline shown in Figure 6.5, validating the accuracy of the simple conflict model.

The sampled values clearly show that Hierarchical was successful in reducing the number of conflicts in each mapping below that from random mappings. Not a single address space had an unnecessary conflict from the Hierarchical mapping. This is largely because of the number of pages available (in the free page frame pool) when the simulator maps each page, 4-megabytes worth. On the average, there were four pages available for mapping in each bin of the 1-megabyte cache. These extra pages allowed Hierarchical enough freedom to obtain perfect conflict reduction in this cache.

While Hierarchical eliminated all unnecessary conflicts in the 1-megabyte cache, some remained in the 16-megabyte cache. The average 0.25 pages available in each 16-megabyte bin is not enough for complete eradication of the unnecessary conflicts, though Hierarchical removes most of them, significantly improving cache performance.

Figure 6.22 establishes the positive relationship between the conflicts eliminated by Hierarchical and the corresponding miss reduction. It shows the Hierarchical reduction for only the Mult2.2 address spaces that were of size 0.5-megabyte to 1.0-megabyte; this isolates the effects of conflicts on the 58 address spaces within a small size range. The *MPI* of an address space is the *MPI* of the instructions that use the address space; this includes intra-address-space and inter-address-space conflict components. Figure 6.22 isolates the relationship between the static conflicts that Hierarchical eliminates

---

confidence intervals are directly proportional to the standard deviation. The standard deviation is the square root of the variance.

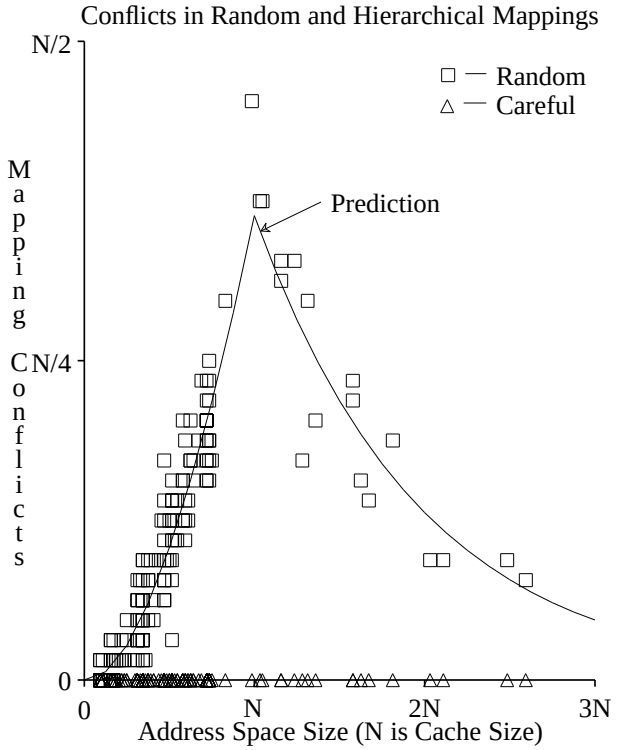


Figure 6.21. Conflicts for Mult2.2.

This figure shows the number of page conflicts (16-kilobyte pages) in a 1-megabyte direct-mapped cache ($N = 64$, 128-megabyte main memory) for various processes from the Mult2.2 trace for a random mapping (squares) and a mapping using Hierarchical (triangles).

from an address space, and the amount that Hierarchical reduces the *MPI* of the address space. It compares the *MPI* of an address space with random page mapping to the *MPI* of the exact same references to the exact same address space, only with Hierarchical used rather than Random page mapping. Although the figure shows Hierarchical reductions for *many* different address spaces with different static conflicts in each, rather than a *single* address space with different static conflicts, the results for the 58 similar address spaces should estimate the *average* relationship between static conflicts and dynamic cache performance.

Each square in Figure 6.22 shows the average miss reduction for those address spaces with a a fixed number of conflicts. Note that most of the miss reductions are larger than zero. This verifies that careful page mapping reduces cache *MPI* because it eliminates conflicts and improves cache utilization. The slope of the best-fit line is further evidence of the positive relationship between conflict elimination and miss reduction. Though careful mapping always eliminates conflicts, several address spaces with few conflicts had a slightly higher *MPI* with careful mapping. This is caused by random inter-address-space conflicts; these conflicts are more severe with careful page mapping than without it because each carefully-mapped address space more fully utilizes the cache. On the average, when careful mapping eliminates more conflicts, it eliminates more cache misses. Together, Figure 6.21 and Figure 6.22 support a key hypothesis of this chapter: techniques that eliminate static page conflicts in a real-indexed cache also improve dynamic cache performance.

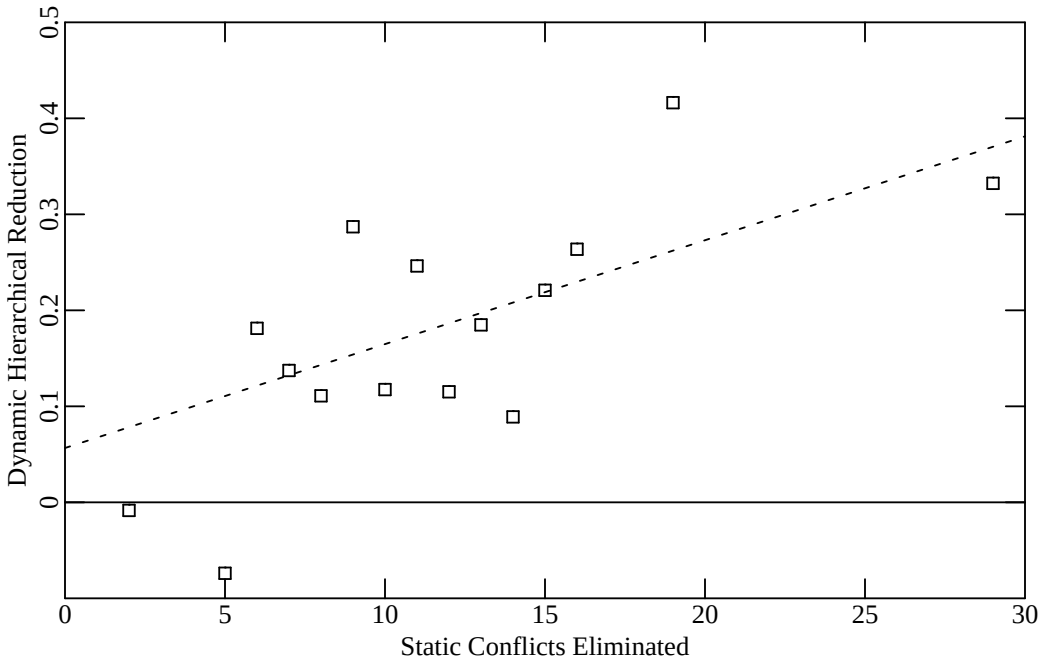


Figure 6.22. Hierarchical Reduction vs. Conflicts Eliminated.

For the 58 Mult2.2 address spaces of size 0.5-megabyte to 1.0-megabyte, this figure shows the Hierarchical reduction versus the number of static conflicts that Hierarchical eliminates (in a 1-megabyte direct-mapped cache, 16-kilobyte pages). Each square is the average Hierarchical reduction of the address spaces at the given number of unnecessary static conflicts. The dashed line is the least-squares best-fit for all 58 address spaces.

6.4.2.3. Careful Mapping Alternatives

This section compares the algorithms from Section 6.3 using the Mult2.2 trace. The size of the available page frame pool has an important affect on the comparison. The careful mapping implementations that can eliminate the most misses with the least available page frames are preferred because they will be most adaptable and have the minimum (or no) page replacement effect. For the Hierarchical results given so far in this chapter, the pool is 4-megabytes. As an experiment to learn the pool size effect, Table 6.2 compares the 4-megabyte direct-mapped Mult2.2 *MPI* of the different algorithms (four-sample mean) with a 4-megabyte pool of available pages (large) and a 256-kilobyte pool (small). (Except for Page Coloring and random mapping, most of the results in this table are accurate to within ± 0.02 or less with 90% confidence.) On the average there was one available page frame in each bin with the large pool, and 0.0625 available page frames in each bin with the 256-kilobyte pool.

Equality Page Coloring performs poorly in the Mult2.2 comparison presented in Table 6.2. Its particularly high mean *MPI* for the small pool comes from the exceptionally high *MPI* of two (out of the four) simulations. The *MPI* variance of these four simulations is over an order of magnitude larger than those for the other careful mapping algorithms. This is a symptom of the basic problem with equality page coloring: the placement of commonly used virtual pages from different address spaces in the same cache bins often leads to many unnecessary cache misses. Because of this problem, MIPS probably does not use the strict equality match Page Coloring simulated here. The Page Coloring implementation with PID-hashed matching performs much better for Mult2.2 since the same virtual

Secondary Cache *MPI*×1000 **(Mult2.2)**		
Heuristic	Large Pool	Small Pool
Random	0.71 (19%)	0.71 (8%)
Page Coloring (equal)	0.69 (15%)	0.85 (30%)
Page Coloring (hash)	0.64 (8%)	0.71 (7%)
Bin Hopping	0.60 (0%)	0.67 (1%)
Best Bin	0.60 (0%)	0.61 (-8%)
Hierarchical	0.60 (0%)	0.66 (0%)

Table 6.2. Careful Mapping Comparison.

This table compares results of the different mapping algorithms outlined in Section 6.3 for different page frame pool sizes for the Mult2.2 trace. It shows the four-sample mean *MPI* for 4-megabytes of available page frames (Large Pool) and 256-kilobytes of available page frames (Small Pool). In parentheses, it also gives the relative difference from the Hierarchical results for each pool size. The results shown are for a 4-megabyte direct-mapped cache.

addresses do not conflict, yet it still performs poorly compared to the other careful mapping schemes. The limited flexibility of the Page Coloring implementation is the cause of this poor performance: if Page Coloring does not find an available page frame in the matching bin, it abandons the search and chooses any arbitrary page frame. With the small pool, an exact match is unlikely (probability 0.0625). The other careful mapping schemes are more adaptable to situations with few available page frames.

As expected, Best Bin produces superior maps (ones that give a lower *MPI*) in these simulations, whether there is a small or large pool of available page frames. A smaller pool doesn't significantly effect Best Bin, but it does cause the four simulations with the other algorithms to have more cache misses. With the 4-megabyte pool of available page frames in Table 6.2, Bin Hopping, Best Bin, and Hierarchical perform nearly equivalently. Each of them finds it easy to eliminate cache contention when there is one available page frame in each bin on average. With 0.0625 available page frames per bin, Best Bin eliminates the most misses. This shows that Best Bin is the most flexible across both large and pool sizes, and that it improves Mult2.2 cache performance with the smallest page replacement effect. About one available page frame per bin may be a good target pool size since it gives the careful mapping algorithms sufficient flexibility, but is small enough to have little effect on the page replacement policy.

Table 6.3 compares the Mult2.2 cache performance of the different careful mappings for a variety of caches. The pool of available page frames is 4-megabytes. In **Bold**, Table 6.3 shows the caches for which Page Coloring, Bin Hopping, and Best Bin specifically produced their mappings. Because they are not size-independent, these algorithms only optimize their page mappings for the 4-megabyte direct-mapped and 16-megabyte 4-way set-associative caches. The size independence of Hierarchical implies that its mappings are optimized for the entire range of caches, so some entries in Table 6.3 are an ''unfair'' comparison. Hierarchical performed better than the other careful mapping schemes for the non-optimized caches. These results show the usefulness of size independent mappings.

6.4.2.4. Does PID-Hashed Virtual-Indexing Approximate Real-Indexing?

The simulation time needed to obtain virtual-indexed simulation *MPI* results is much less than the time needed for real-indexed results. Real-indexed cache *MPI* estimates are more difficult to obtain, both because virtual to real address translation increases simulation time, and because multiple

Secondary Cache *MPI*×1000 **(Mult2.2)**						
Config Size	A	Page Coloring (equality)	Page Coloring (PID hash)	Bin Hopping	Best Bin	Hierarchical
1M	1	1.37 (15%)	1.32 (11%)	1.33 (12%)	1.38 (16%)	1.19 (0%)
	2	1.03 (5%)	1.02 (4%)	1.00 (2%)	1.03 (5%)	0.98 (0%)
	4	0.94 (2%)	0.94 (2%)	0.94 (1%)	0.95 (2%)	0.93 (0%)
4M	**1**	**0.69 (15%)**	**0.64 (8%)**	**0.60 (0%)**	**0.60 (0%)**	**0.60 (0%)**
	2	0.53 (3%)	0.53 (3%)	0.52 (1%)	0.52 (2%)	0.51 (0%)
	4	0.50 (2%)	0.50 (2%)	0.50 (0%)	0.50 (1%)	0.49 (0%)
16M	1	0.31 (12%)	0.31 (10%)	0.30 (6%)	0.29 (4%)	0.28 (0%)
	2	0.25 (9%)	0.25 (7%)	0.24 (4%)	0.24 (2%)	0.23 (0%)
	4	**0.24 (6%)**	**0.23 (4%)**	**0.23 (2%)**	**0.22 (0%)**	**0.22 (0%)**

Table 6.3. Performance of Mapping Schemes for Mult2.2.

This table compares the cache performance of mappings produced by the careful mapping algorithms for the Mult2.2 trace. It gives the *MPI* resulting from each scheme in absolute value and relative to the Hierarchical *MPI* results (in parentheses) for each cache. Page Coloring, Bin Hopping, and Best Bin optimized their page mappings for the 4-megabyte direct-mapped cache and the 16-megabyte 4-way associative cache. The table shows these caches in **bold**.

simulations are required to remove the randomness of different page mappings. If virtually-indexed cache *MPI* results could be used as an approximation of real-indexed ones, the required simulation time could be greatly reduced. Figure 6.23 compares the *MPI* of real-indexed and virtual-indexed caches. The figure uses careful mapping for the real-indexed results and PID-hashing for the virtually-indexed results. Of course, the results are heavily dependent on the form of PID-hashing used. The simulator bitwise exclusive-ors the eight bit PID with the high-order eight bits from the index field of the virtual address to choose the cache set (see Figure 6.1 for the definition of the set-index field). Figure 6.23 shows the ratio of virtually-indexed *MPI* to real-indexed *MPI*.

Figure 6.23 shows that PID-hashed virtual indexing gives a lower *MPI* estimate than physical indexing (because all the ratios are less than 1.0). Since sequential virtual pages do not conflict in a virtually-indexed cache, the *MPI* is smaller. Simulation results for PID-hashed virtually-indexed caches consistently underestimate the carefully-mapped real-indexed *MPI*, but in over half of the cases the estimation error is 10% or less. PID-hashed virtually-indexed cache results are a strongly-optimistic estimate of real-indexed cache performance with random page mapping, however.

Another important factor in the approximation of real-indexed caches with virtual-indexed caches is the accuracy in predicting relative *MPI* changes. One important relative variation is the fraction of misses that an associativity increase eliminates. Figure 6.24 plots, for virtually-indexed and carefully mapped real-indexed caches, the miss reduction of doubling associativity from direct-mapped to 2-way. The comparison shows that virtually-indexed caches also consistently underestimate this relative reduction, though as with the absolute value, often by only a small amount. This occurs for the same reason that virtual-indexing underestimated the absolute value: there is less direct-mapped contention with virtual-indexing because sequential addresses do not conflict. Though careful mapping has some contention reduction advantages of virtual-indexing, virtual-indexing still gives lower miss frequencies.

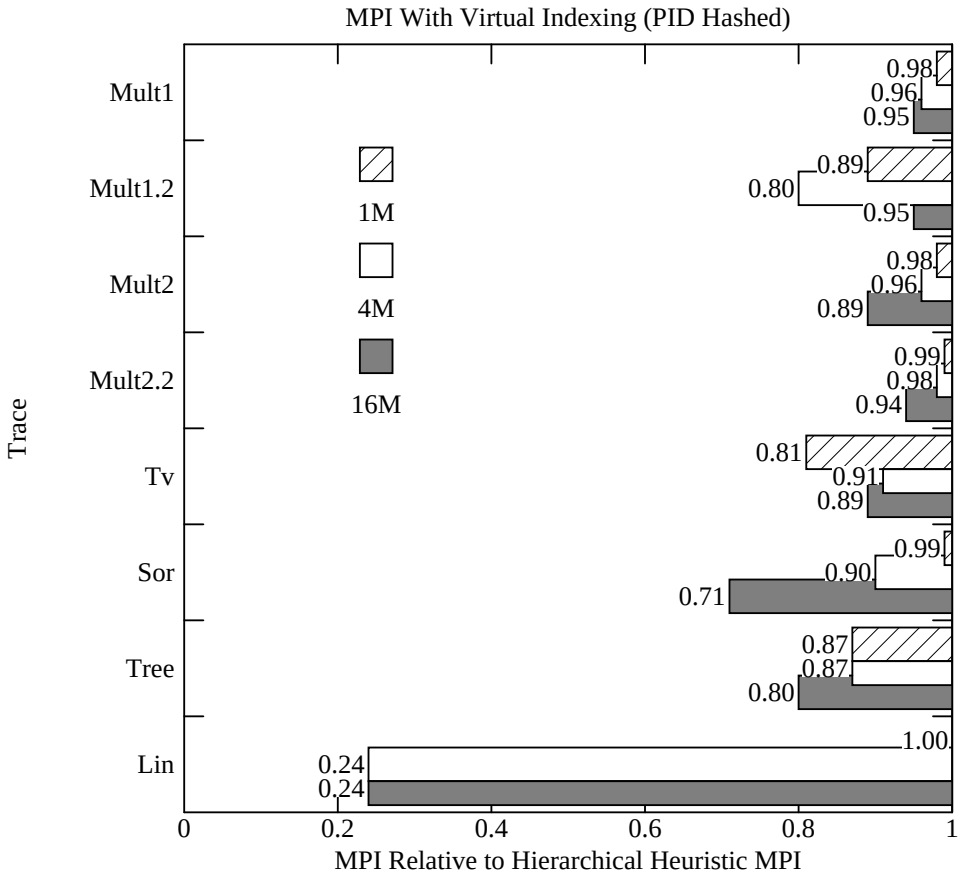


Figure 6.23. Virtual and Real Indexing Comparison.

This figure shows, for different direct-mapped secondary caches and traces, the miss frequencies produced by virtual indexing (with PID-hashing) relative to that produced by real-indexed caches (from Hierarchical mappings).

6.4.2.5. Page Size Performance Effects

The page size affects the comparison between random and careful mappings. Table 6.4 shows the Mult2.2 cache *MPI* for different page sizes. It shows that the random mapping *MPI* decreases slightly with increasing page size. A combination of factors could produce this effect. The first is that sequential addresses will not conflict with larger pages. Also, there are fewer page frames in each bin, so it is less likely that many pages will be mapped to the same bin. With careful mapping, cache performance is largely independent of the page size. Hierarchical reduced contention with all three page sizes.

A disadvantage of bigger page sizes is the larger amount of main memory used by an application because of *internal fragmentation*. Some portions of large pages are not referenced though an entire page must be mapped. Table 6.5 shows the page misses and amount of memory used by the Mult2.2 trace versus the page size. The memory usage is the number of unique referenced pages times the page size. The referenced memory nearly doubles as the page size increases from 4-kilobytes to 64-kilobytes.

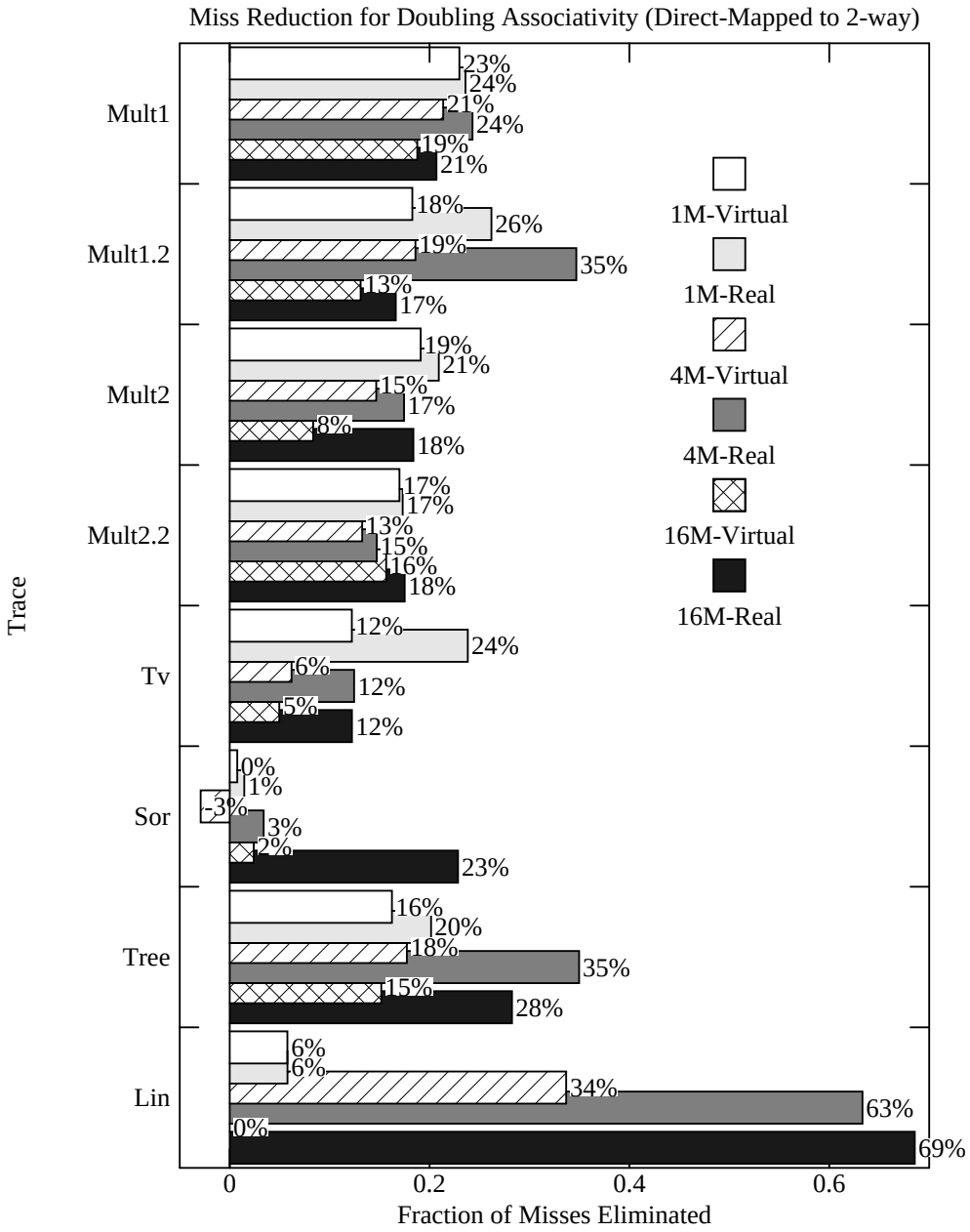


Figure 6.24. Virtual and Real Indexing Associativity Improvement.

This figure shows, for different direct-mapped cache sizes and traces, the reduction in misses by doubling the associativity from direct-mapped to 2-way for virtual-indexed caches (with PID-hashing) and real-indexed caches (with Hierarchical mappings).

6.4.2.6. The Effect of Main Memory Contention

As an experiment to understand the effect of main memory contention on cache performance and careful mapping algorithms, the execution of the Mult2.2 trace was simulated on a system with a main memory size of 32-megabytes. This experimental design created main memory contention since the Mult2.2 workload had about 40-megabytes active at any given time, more than the 32-megabytes of main memory. Random mappings could then be compared to careful mappings under these conditions.

Secondary Cache *MPI*×1000 **(Mult2.2)**							
Config		Random			Hierarchical		
Size	Assoc.	4K	16K	64K	4K	16K	64K
1M	1	1.49	1.50	1.36	1.18	1.19	1.19
	2	1.05	1.05	1.02	0.97	0.98	0.99
	4	0.96	0.96	0.94	0.92	0.93	0.93
4M	1	0.70	0.71	0.66	0.59	0.60	0.60
	2	0.54	0.54	0.53	0.51	0.51	0.51
	4	0.51	0.51	0.50	0.49	0.49	0.49
16M	1	0.33	0.33	0.32	0.28	0.28	0.28
	2	0.25	0.25	0.25	0.23	0.23	0.23
	4	0.24	0.24	0.24	0.22	0.22	0.22

Table 6.4. Cache Misses vs. Page Size.

This table shows, for various cache sizes and associativities, Random and Hierarchical mapped real-indexed cache miss frequencies for page sizes of 4-kilobytes, 16-kilobytes, and 64-kilobytes.

Memory Referenced (Mult2.2)		
Page Size	Page Misses	Memory (megabytes)
4K	22651	92.8
16K	7471	122.4
64K	2428	159.1

Table 6.5. Memory Referenced vs. Page Size.

This table shows, for the Mult2.2 trace, the number of page misses and the memory referenced (calculated by multiplying the number of page misses times the page size) for page sizes of 4-kilobytes, 16-kilobytes, and 64-kilobytes with a 128-megabyte main memory.

Using static traces of multiprogrammed workloads, trace driven simulation results can be inaccurate with memory contention. The statically ordered traces represent a process execution ordering that is unaffected by system load control policies [DENN80]. Load control policies decide the processes that execute at any given time. When main memory demand is high, overall system performance might be improved by delaying execution of some processes so that main memory demand decreases. Proper load control policies can eliminate the *thrashing* problem [DENN68] of poor system performance caused by the overcommitment of main memory.

Another potential inaccuracy of static traces is that the simulations do not model the effects of write-backs and retrievals of pages between main memory and secondary storage. The simulator ignores any secondary storage accesses required to support page placement or replacement. This is equivalent to the assumption of an infinitely fast secondary storage. A slow secondary storage device could greatly alter the process execution interleaving. The assumption should not bias the results too much since cache performance, not main memory and secondary storage performance, is the major interest in this study.

Table 6.7 shows the results of the memory contention experiment. Note that the page miss frequencies are unrealistically high because of cold start. The results show that there are more page misses with decreasing main memory size because of main memory contention. While the experiment reduced memory size (to 32-megabytes), it did not reduce the size of the available page frame pool (4-megabytes). The experiment used this large pool to gauge the effect of a large available pool on LRU

page replacement decisions. Table 6.7 shows that while Hierarchical reduces *MPI* with or without contention, there are slightly more page misses with contention. This suggests that the pool caused poorer page replacement decisions because it was a relatively large portion of the main memory. The 4-megabyte pool was probably too large given that the main memory was only 32-megabytes in this experiment. The optimal pool size depends on the balance between improved cache efficiency and the costs of any poorer page replacement decisions. With higher page fault costs, a smaller page frame pool would be preferred. Likewise, a larger pool is more beneficial with lower costs.

Memory Size Effect			
Memory Size	Mapping	Secondary *MPI*×1000	Page Faults Per 10^6 Inst.
128 M	Random	1.50	2.00
	Hierarchical	1.19	2.00
32 M	Random	1.42	2.69
	Hierarchical	1.21	2.73

Table 6.7. Cache and Page Misses for Memory Contention.

This table compares Mult2.2 1-megabyte direct-mapped simulation results for large (128-megabyte) and small (32-megabyte) main memories. This experiment shows results for Random Mappings and Hierarchical Mappings (4-megabyte available page pool). The table shows secondary cache misses per thousand instructions and page misses (faults) per million instructions.

With Random page mapping, the *MPI* decreases slightly with the smaller main memory. This is similar to the *MPI* reduction with larger pages, and may occur because there are fewer page frames in each bin, making it less likely that many pages from the same address space map to the same bin.

6.5. Discussion

It is important to compare the algorithms to see how well they meet their goals. Section 6.4 shows that implementations of careful mapping reduce the mean cache *MPI* and its variance. Those are the most essential requirements of a careful mapping scheme. It also shows that, for Mult2.2, Best Bin eliminates the most cache misses with the smallest pool of available page frames. This is a useful property. It says that Best Bin is the most adaptable and has the smallest page replacement policy effect of any careful mapping scheme. Another important goal is that careful mapping should be low overhead, both in terms of required storage space and execution time. Page Coloring and Bin Hopping have the lowest space overhead, and Page Coloring has the smallest execution time overhead. Best Bin and Hierarchical have more space overhead. Best Bin's execution time can be too large; if it requires only a few instructions per bin (on average), and there are thousands of bins, Best Bin could take thousands of instructions. Hierarchical's execution time is low: a couple hundred instructions should traverse a tree with thousands of bins. Also, Hierarchical is the only careful mapping algorithm from this dissertation that is size independent. Size independence can be a dominant concern with a hierarchy of real-indexed caches.

The careful mapping implementations should adapt to the features of virtual memory systems, including sharing among different address spaces. Static page placement optimization becomes complicated when different address spaces share the same page because shared pages can cause contention in all address spaces accessing the page frame. A careful page mapping function should adapt to sharing among different address spaces. It should preferably map the page to a page frame that minimizes the

contention in all address spaces sharing the page. As described in Section 6.3, none of the careful mapping implementations account for sharing.

Bin Hopping may not adapt well to sharing because it optimizes for one address space and may cause conflicts in another. The problem is that Bin Hopping remembers only the last placement within a single address space, and uses only this information to place the next page. Thus, Bin Hopping maps a page to improve the bin utilization of a *single* address space, which may or may not improve the bin utilization of the other address spaces accessing a shared page.

The strict equality match implementation of Page Coloring adapts well to sharing when the shared pages are at the same virtual address in all address spaces because the preferred bin for a virtual page is the same for each address space. More sophisticated versions of Page Coloring, such as the PID-hashing implementation considered in this study, may not adapt for the same reason that Bin Hopping will not: the mapped bin depends on the PID used to map the page. A Page Coloring implementation that, instead, maintains an offset for each shareable unit (a collection of pages, such as a shared code segment), could better adapt to sharing [TADF90]. Then, all address spaces map pages similarly since each address space sharing a page uses the same bin offset.

With the Best Bin and Hierarchical careful mapping implementations, the *used* state information should be maintained per shareable unit to better adapt to sharing. The *used* state of an address space is then the sum of the *used* states of each shareable unit referenced within the address space[29]. Mapping decisions for shared pages will still be optimized only with respect to the address space mapping the page. However, Best Bin and Hierarchical would still adapt to sharing because all subsequent mapping decisions (of all address spaces referencing the page) adjust to the positioning of the shared page through the updated *used* state. Note that the summation of *used* state for multiple shareable units can make the mapping process more time consuming. Consequently, the efficiency of Hierarchical will be even more essential for good performance.

6.6. Conclusions

This chapter shows the usefulness of careful virtual to real page mapping policies, describes several practical careful mapping implementations, and shows their performance improvement using trace-driven simulation. When a real-indexed cache is large, the virtual to real page mapping function and the cache mapping function interact to determine cache performance. This gives the page mapping policy the opportunity to improve cache performance by carefully choosing the virtual to real page mapping of an address space. It is worthwhile to expend a little effort to map a page carefully since it can improve the performance of all following accesses to the page. This chapter shows that virtually-indexed caches have advantages over real-indexed caches with poor virtual to real mappings, but that careful mapping can eliminate many of the random real-indexed misses.

This chapter shows that static cache bin placement optimizations produce dynamic cache performance improvements. A simple static analysis shows that as many as 30% of the pages from an address space are unnecessarily in conflict; this is the static potential available to the careful page mapping

29. If the implementations instead only maintain one array (tree) of *used* state information per address space, when the operating system adds or removes a shared page it must update the *used* state of *all* address spaces using the page so that all *used* information is correct. This update may be both difficult and time consuming.

algorithms. This analysis also correctly predicts that conflicts are less of a problem in caches of higher associativity.

This chapter describes Page Coloring and introduces several other practical careful page mapping algorithms that improve the cache bin placement of pages based solely on static information. These algorithms equalize cache bin utilization and eliminate cache conflicts. They range from the simple Page Coloring and Bin Hopping to the more sophisticated Best Bin and Hierarchical. This chapter quantitatively compares the *MPI* improvements of the algorithms with trace-driven simulations of a multiprogrammed trace (Mult2.2). This comparison shows that Bin Hopping, Best Bin, and Hierarchical perform well. The comparison also shows that Equality match Page Coloring performs poorly. The problem with equality match Page Coloring is that it maps frequently used areas of the virtual memory space to the same cache bin. A version of Page Coloring that eliminates this problem still does not compare favorably to the other careful mapping schemes. Under the conditions that this chapter studies, Best Bin produces the best page maps with the smallest pool of available page frames, so it improves cache performance with the minimum (perhaps no) page replacement policy effect. Hierarchical has good computational efficiency and cache size independent mappings; these properties are desirable, particularly for hierarchies of real-indexed caches.

This chapter correlates the static conflict optimizations of the mapping algorithms with the dynamic cache miss reductions. It also shows that virtual-indexing (with PID-hashing) optimistically estimates real-indexed cache performance, that careful mapping is effective over a range of page sizes, and that careful mapping with a large pool of available page frames (relative to the main memory size) can cause more page misses.

Careful virtual to real page mapping is a useful, low cost, technique to improve the performance of large, real-indexed CPU caches. This chapter presents simulation results from several workloads to measure the improved *MPI* mean and variance of careful page mapping (Hierarchical). The improvement depends on the workload, but, 10%-20% of direct-mapped cache misses are eliminated (on average) from our traces. Thus, at no hardware cost, software careful page mapping eliminates about half of the direct-mapped cache misses that either doubling the associativity or cache size can eliminate from the traces used in this dissertation.

Chapter 7

Summary and Future Work

7.1. Summary

This dissertation provides the analysis required to understand multi-megabyte secondary CPU caches. The contributions of this work include new cache performance analysis techniques, and design suggestions for both hardware and software memory system designers. This section summarizes the contributions of this dissertation.

Chapter 2 defines basic concepts, gives default parameters, and surveys state-of-the-art research in the area of cache memory analysis. It then shows that the *MPI* and *SCPI* cache performance metrics used in this dissertation take into account the frequency that a cache is accessed, in addition to the cache miss ratio, to determine the performance effect of a cache. *SCPI* is a useful performance metric to gauge the effect of cache misses on each instruction; it allows the effects of each cache in the hierarchy to be determined separately, then combined into a system total. Then Chapter 2 shows the statistics used in this dissertation, and also describes a simulation environment with enough processing power to gather the results needed for this dissertation.

Chapter 3 describes a collection of new traces for the analysis of multi-megabyte caches that were used throughout this dissertation. These traces overcome several deficient aspects of previous workloads since they are one hundred times longer than previous traces, and are from workloads that are ten times larger than previously traced. These workloads are the type that can be expected when high-performance processors with large main memories become available. This dissertation describes new tracing techniques required to gather these long traces. Chapter 3 describes the code modification mechanism used to gather them; Anita Borg and David Wall developed this mechanism at DEC Western Research Laboratory. Chapter 3 also describes the sophisticated trace differencing and compression needed to capture the traces; each memory reference required only about two bits of

storage.

Chapter 3 also shows two advantages of long traces. The first is that longer traces give a more complete characterization of the variations of behavior across different execution phases of the workload. The second advantage is that long traces can overcome the long cache initialization, or cold-start, transient of the multi-megabyte caches. These new traces are superior to previous traces for the analysis of multi-megabyte caches, and were essential to obtain the results contained in this dissertation.

The problem with long traces is the enormous simulation and storage resources required to use them. Chapter 4 compares two different trace-sampling techniques that use only a portion of the full trace memory references to estimate cache performance for the full trace. Since simulation time and storage space is proportional to the number of required memory references, this reference reduction translates directly into reduced simulation and storage requirements. One sampling technique, set sampling, extracts references to a portion of the address space, those addresses that index to a portion of the sets in the cache. Chapter 4 introduces a constant-bits technique that produces set samples that are useful for the widest range of cache configurations; a constant-bits sample is a set sample when the cache index bits contain the constant-bits. Set sampling is effective: a trace data reduction factor of 16 still gave less than 10% errors.

The other trace-sampling technique, time sampling, extracts many samples, where each is a time-contiguous portion of the full trace references. Cold start is a serious problem limiting the usefulness of time sampling. Chapter 4 compares five different cold-start reduction techniques. It shows that initialization reference miss ratio prediction [WOHK91] was most effective at mitigating the bias. Even with this technique, multi-megabyte caches require time samples of millions of instructions to remove cold-start effects. Additionally, time sampling requires many samples to characterize cache performance for a long trace. Only many samples can capture adequate portions of different execution phases of the traced workloads to estimate cache performance accurately. The combination of large time-samples with many time-samples makes time sampling less desirable than set sampling. Provided some restrictions can be tolerated, set sampling is preferred.

Chapter 5 focuses on the design of multi-megabyte caches. In so doing, it gives a detailed motivation for two-level hierarchical cache configurations with secondary multi-megabyte caches. A feature of multi-megabyte caches is the significant reduction in memory traffic required when the cache is inserted between the processor and the main memory. Smaller and faster primary caches are an important component of a hierarchical memory system containing multi-megabyte caches since large caches are too slow to service processor references directly. A hierarchical configuration with both primary and multi-megabyte secondary caches services most references at processor speeds, and retains the capacity advantages of the large secondary cache.

The *SCPI* results from Chapter 5 show that multi-megabyte caches are extremely useful when the processor-main-memory speed gap reaches a factor of 100 or more. They also show that the workload has a large effect on *SCPI*, so the expected workload greatly affects the cache size choice. An examination of associativity design alternatives shows that 2-way set-associativity extracts most of the advantage of higher associativities. While associativity is useful for smaller secondary caches, it is less necessary with multi-megabyte caches because direct-mapped already performs well. LRU replacement can improve replacement decisions, but it also can have an implementation cost for higher associativities, so random replacement may be preferred. Chapter 5 examines several inexpensive implementations of associativity. It shows that the partial comparison technique performs well for smaller

secondary caches. The MRU implementation also performs well for the largest secondary caches.

Chapter 5 also examines different secondary cache block sizes. Its findings are different than previous studies: larger block sizes are better for multi-megabyte caches. It prefers larger blocks because it considers larger caches and different memory system parameters than the previous work. The block size with equal latency and transfer components is a good design point, though block size choice varies with cache size. The *SCPI*-minimal block size is often within a factor of two of this design point, and the *SCPI* at this design point is often near the minimum *SCPI* for all block sizes.

Chapter 5 finally looks at multi-level inclusion design options. It finds that replacement decisions can affect the frequency of primary cache invalidations needed to maintain inclusion, and that software cache consistency may be more appropriate than hardware cache consistency (with inclusion) in uniprocessors because of the increased complexity and performance degradation potential of inclusion.

To complete this dissertation, Chapter 6 examines the interaction of virtual memory and set-associative CPU caches and develops software techniques to improve cache performance. Cache performance can only be maximized with a combination of both software and hardware techniques. For multi-megabyte real-indexed set-associative caches, the associativity disparity of virtual memory and caches implies that the placement of pages in the main memory determines the indexing of pages in the cache. A simple model shows that with a naive mapping of pages to page frames 30% of the pages from an address space may be unnecessarily in conflict (on average). It also predicts that the most page conflicts can be eliminated at the point where the address space size equals the cache size.

Chapter 6 explores the design issues of careful page mapping policies that will eliminate these conflicts. It introduces and examines two simple policies and two more complex ones, and outlines the key properties of each. Appendix A shows that one of them, Hierarchical, satisfies the size-independent property. Chapter 6 examines careful page mapping using trace-driven simulation. The results show that careful page mapping can reduce the frequency of direct-mapped cache misses by 10%-20% for user references. Thus, at no hardware cost, careful page mapping makes a direct-mapped cache appear 50% larger. The results also show that careful page mapping reduces the variance in cache performance for different executions of a workload, a desirable trait. A comparison of the different policies for one trace shows that Best Bin chooses the best page maps, but it can be costly. Hierarchical is an efficient variant that retains many of the desirable properties of Best Bin, and produces cache size independent maps. Bin Hopping works well, but Page Coloring without any hashing does not. Chapter 6 also shows that carefully-mapped cache performance is largely independent of page size, that virtual indexing with PID-hashing gives optimistic results as compared real-indexing, and that the maintenance of a large available page frame pool can cause more page faults.

7.2. Future Work

This section suggests areas for future work to complement this research. This dissertation contributes to memory system performance analysis, but rapid technology change always ensures that new research avenues open up, so there is always more work to be done.

One important area for further research would be more and better traces. Chapter 3 showed the dramatic changes in cache performance for different workloads. It would be useful to have many more workloads to more precisely characterize the differences for different workloads. Related to this would be a study to answer the question: is it better to have fewer long traces or to use trace sampling to trace more workloads with the same number of references? Chapter 4 shows that sampling can be effective.

However, the answer to this question also depends on the accuracy and usage flexibility of the samples, and on what the intended use of the samples is. In particular, regarding set sampling, what constant bits should be used? With time sampling, how long of a sample is long enough? While INITMR in Chapter 4 successfully eliminates the cold-start bias when there are more known misses than initialization references, it would be useful to more precisely characterize the relationship between cold start and trace length, and to find more sophisticated techniques to eliminate the cold-start bias in time samples. It also would be useful to develop better techniques that use only the samples themselves to establish confidence levels in the acquired results, and find a more accurate relationship between the number of samples (or the trace data fraction) and the accuracy of the result.

An examination of the methodology used to gather the traces would be fruitful. The workloads were chosen particularly because they were large, hopefully to reflect characteristics similar to the large workloads that will likely be common as high-performance processors with large main memories become available. It is unclear how well these workloads reflect the real workloads of multi-megabyte caches. A comparison of different workload gathering methodologies would be very interesting.

Related to the gathering of traces, a serious deficiency of the traces is the lack of operating system references. The *MPI*'s of most traces probably would increase with operating system references, so the conclusions with operating system references could differ from the conclusions without them.

There are several ways that the results of Chapter 5 could be extended. It would be useful to decide the design range that is appropriate for two-level cache configurations, and when more cache levels are needed. Furthermore, what is the effect of primary caches on the design and performance of multi-megabyte secondary caches? What is the design effect of the differences between real-indexed and virtual-indexed cache performance? How accurately does *SCPI* predict the true performance of hierarchical caches? What is the relationship of workload behavior to the cache size choice? For a given workload, what cache size is large enough? There is also always the need for exploring a larger design space with a wider variation in parameters because parameters can vary in different implementations.

There are many ways that the research in Chapter 6 could be extended. A precise characterization of the relationship between static page conflicts and the resulting *MPI* of the address space would be useful. There are many opportunities to develop more sophisticated careful page mapping functions. It might be useful to discriminate between different types of pages, such as code and data pages. For instance, one could minimize conflicts between pages of the same type in addition to minimizing the all conflicts within an address space. More sophisticated bin choice heuristics could be used by Best Bin and Hierarchical. Another area where research should be directed is the relationship between page pool size and page miss frequency. This could help find the optimal page pool size by trading off the reduced cache miss frequency with the increased page fault frequency of a larger pool.

Multi-megabyte caches promise to be a more and more important component of high-performance memory system design. This thesis is about the performance analysis and design of multi-megabyte secondary CPU cache memories. It is the first performance evaluation of multi-megabyte caches. Though there are many more possibilities for further research, this dissertation is an important starting point to understand multi-megabyte caches.

Appendix A

This appendix consists of a proof that Hierarchical produces cache size independent mappings (as defined by Definition A.3).

First, this appendix formalizes some of the concepts that are developed in the text of the chapter. A page mapping function places a virtual page in a real page frame. More formally:

Definition A.1. Page Mapping Function.

Given a current mapping of virtual pages to real page frames and the page frames available for mapping, a *page mapping function* $MAP(vp, CC) = pf$ chooses an available page frame, *pf*, to map the virtual page *vp* into the main memory and the real-indexed (by bit-selection) set-associative CPU cache *CC*.

To understand much about the cache effects of accesses to memory locations within an address space, it is necessary only to know the bins to which the pages map. The page mapping function can easily learn the bin a page frame is in by examining the bottom bits of the page frame number, the bin bits:

Definition A.2. Page Frame and Cache Bin Correspondence.

Figure 6.1 shows that a page frame belongs to a cache bin defined by the value of its *bin* bits. That is, the bin that a page frame *pf* will map to in a cache *CC* is

$$bin(pf, CC) = pf \bmod 2^{depth(CC)},$$

where $depth(CC)$ is the number of *bin* bits for the given cache (given the page size). Two page frames, pf_1, pf_2, are equivalent with respect to a cache *CC* when their *bin* bits are equivalent,

$$bin(pf_1, CC) = bin(pf_2, CC).$$

A formal statement of the size independence property is:

Definition A.3. Cache Size Independent Mapping Function.

A page mapping function $MAP(vp, CC)$ produces size independent mappings (i.e. is size independent) when $MAP(vp, CC_2)$ chooses a page frame that is equivalent with respect to CC_1 to $MAP(vp, CC_1)$ for $depth(CC_1) \le depth(CC_2)$ ($depth(CC)$ is the number of *bin* bits in the cache *CC*). That is,

$$bin(pf_2, CC_1) = bin(pf_1, CC_1)$$

($pf_2 = MAP(vp, CC_2)$ and $pf_1 = MAP(vp, CC_1)$) for some arbitrary choices by each mapping function.

Effectively, size independence requires that the decisions made by the mapping function optimized for a larger cache must be equivalent to the decisions made by the function optimized for a smaller cache.

A formal definition of a Bit-Reversed Bin Tree is:

Definition A.4. Bit-Reversed Bin Tree.

A *Bit-Reversed Bin Tree* (BRBT) is a fully balanced binary bin tree of depth *d* with nodes at each level of the tree labeled in a bit-reversed fashion as shown in Figure 6.14. The value of a leaf node is the number of page frames, either mapped or available, in the labeled bin. The value of an interior node is the sum of the values of its children. A node in the tree can be uniquely identified by the pair $<x, l>$, where *x* is the label of the node and *l* is the level of the node in the tree ($0 \le l \le d$, $0 \le x < 2^l$). Page frames, identified by their page frame number *pf*, map to node $<x, l>$ when the *bin* bits are equivalent to the node label, that is, $pf \bmod 2^l = x$. The root node is <0, 0> and all page frames map to it.

This node labeling identifies the parent of node $<x, l>$ by removing the top bit of the *l* bit quantity *x*: $<x \bmod 2^{l-1}, l-1>$ ($l > 0$) is the parent. It also finds the left and right children ($l < d$) by adding a most significant bit: $<x, l+1>$ and $<x+2^l, l+1>$ are the left and right child, respectively.

The node identification in a BRBT shows the refinement of the numbering with increasing tree level. At each level, *l*, each label can be expressed as an *l* bit number. Furthermore, the page frames that map to a node of a BRBT are partitioned into those that map to its left child and those that map to its right child. Every page frame that maps to the parent node must also map to a child. Furthermore, any page frame that maps to a child (a page frame can only map to one child, not both) also must map to the parent node. Given that the children (direct descendants) partition the page frames mapping to a node, all page frames mapping to a sub-tree node also must map to the sub-tree root node. This is the key refinement property that allows a BRBT to maintain size independence.

Theorem A.1 says that the value of all nodes in a BRBT tree is the number of page frames in the labeled bin, an extension of the statement in Definition A.4 that the leaf nodes have this property. Its proof heavily depends on the partitioning properties of the labeling of a BRBT.

Theorem A.1.

The value of a node in a BRBT of depth *d* (as defined in Definition A.4) is the number of pages that map to that node.

Proof -

Let $val(x, l)$ and $card(x, l)$ be the value of $<x, l>$ and number of page frames that map to $<x, l>$, respectively. The proof consists of showing that $val(x, l) = card(x, l)$ for all $<x, l>$ and is by induction.

Basis -

By Definition A.4 $val(x, d) = card(x, d)$ for all leaf nodes $<x, d>$.

Induction -

Given that the condition holds for all nodes at level $l+1$, in particular $val(x, l+1) = card(x, l+1)$ and $val(x+2^l, l+1) = card(x+2^l, l+1)$ (the left and right children) it is necessary to show that $val(x, l) = card(x, l)$ $(0 \le l < d,\ 0 \le x < 2^l)$. The partitioning property of the two children of a node implies that

$$card(x, l) = card(x, l+1) + card(x+2^l, l+1).$$

$Card(x, l)$ is simply the sum of these two values since the direct children are a disjoint partition of all the page frames that map to the node itself. Substituting, this implies that

$$card(x, l) = val(x, l+1) + val(x+2^l, l+1),$$

which also means that

$$card(x, l) = val(x, l)$$

since the value of a node is the sum of the values of its children (Definition A.4). Since x is arbitrary, the condition holds for all nodes at level l. The theorem is complete since by the basis clause and a finite number of inductions $val(x, l) = card(x, l)$ can be shown true for any node $<x, l>$. □

Theorem A.1 is particularly useful since it shows that the value of a node is dependent only on the number of pages mapping to a node, which is independent of the tree depth. Thus, the value of a BRBT node $<x, l>$ is the same in two BRBT's that are of different depths.

Theorem A.2 formally states the proof that Hierarchical produces size independent mappings. It uses Theorem A.1 to show that Hierarchical (using a BRBT) will make the same upper level decisions independent of the tree depth. A taller tree then only refines decisions made with a shorter tree. This implies the larger tree decision is equivalent (with respect to the smaller cache) to the decision with the smaller tree.

Theorem A.2.

Hierarchical (as defined by the code in Figure 6.9 and the example in Figure 6.13, further defined in Figure 6.15) that uses a BRBT (Definition A.4) produces size independent mappings (Definition A.3).

Proof -

The proof is in two parts. The first part is by induction and shows that both $MAP(vp, CC_1)$ $(depth(CC_1) = d)$ and $MAP(vp, CC_2)$ $(depth(CC_2) = D,\ d \le D)$ traverse their trees downward along the same path up to level d, provided they make the same arbitrary choices.

Basis -

Both algorithms start traversal at node <0, 0>, the root of the corresponding used and free trees. They assume that there is at least one available page.

Induction -

Given that the traversal path is the same up to node <*x*, *l* > at level *l* (*l* < *d*) in the tree, it must be shown that the same traversal direction is taken into level *l* + 1. This hinges on the property that tree values must be the same, independent of the depth of the tree. The same pages map to the used (free) trees for each Hierarchical Heuristic and thus the node values of the different-sized used (free) trees are equivalent (Theorem A.1).

The code in Figure 6.9 makes the direction decision based on the <*used*, *free* > node values of the children of <*x*, *l* >. `Right_free` (`left_free`) and `right_used` (`left_used`) are the values of the right (left) child of node <*x*, *l* > in the free and used bin trees. Thus, each version of Hierarchical will call the `Choice()` function with precisely the same parameters when at node <*x*, *l* >.

Since there is at least one available page in the sub-tree rooted at <*x*, *l* >, either `right_free` or `left_free` must be non-zero. The zero checking of the `left_used` and `right_used` variables in the first two `if` clauses in Figure 6.9 ensures that `Choice()` will always pick a sub-tree with an available page.

There are two further cases within the `Choice()` function that must be considered:

(1) Either `right_free` ≠ `left_free` or `right_used` ≠ `left_used`. Here, the `Choice()` function will deterministically decide based on the parameters of the function (as captured in the six `if` clauses). The decision will be the same in each case.

(2) Otherwise. `Choice()` will arbitrarily decide (as captured in the `else` clause) for both heuristics. The arbitrary decisions can be assumed to be the same, ensuring that `Choice` makes the same direction decision in each case.

Thus, the Hierarchical functions will make the same decision at node <*x*, *l* >.

Through an application of the basis clause and *d* applications of the inductive clause, the Hierarchical heuristics $MAP(vp, CC_1)$ and $MAP(vp, CC_2)$ can both traverse to the same node <*x*, *d* > in the tree, provided they make the same arbitrary direction decisions.

The proof now continues by showing that $MAP(vp, CC_2)$ will refine its decisions to level *D* in the trees. During its downward traversal it will remain within the sub-tree rooted at <*x*, *d* >. Thus, it will complete its traversal at a leaf node in this sub-tree rooted at <*x*, *d* >. At that point it will choose an available page frame that maps to the corresponding leaf node. Since all page frames mapping to a leaf node in a sub-tree also must map to the root of the sub-tree, the chosen page frame also will map to <*x*, *d* >. Thus both $MAP(vp, CC_1)$ and $MAP(vp, CC_2)$ will choose a page frame that maps to <*x*, *d* > and satisfy the condition that $bin(p, CC_1) = p \bmod 2^d = x$. The required result that

$$bin(MAP(vp, CC_1), CC_1) = bin(MAP(vp, CC_2), CC_1)$$

is reached and the theorem is complete. □

Appendix B

This appendix shows the statistical significance of the four-sample *MPI* results used in Chapter 6. The simulator introduces randomness into a static virtual address trace by changing the virtual to real page mapping. The goal is to determine the (true) mean *MPI* over all Hierarchical and Random page maps. For each trace and cache configuration this true mean is estimated with the mean of a sample of size four. The tables which follow show the estimates of the mean MPI with Random ($\overline{\overline{MPI}}_{random}$) and Hierarchical ($\overline{\overline{MPI}}_{Hierarchical}$). They also show the four-sample medians. They also show the Hierarchical reduction ($100\% \times (\overline{\overline{MPI}}_{random} - \overline{\overline{MPI}}_{Hierarchical}) / \overline{\overline{MPI}}_{random}$).

The sample means are most useful, however, if it is likely that they are close to the true mean MPI. 90% confidence intervals measure this likelihood. This appendix calculates 90% confidence intervals using the techniques described in Section 2.5. The width of these student-t confidence intervals is proportional to the four-sample standard deviation. For any particular sample, the 90% confidence interval may or may not contain the true mean of the distribution. It is called a 90% confidence interval because it contains the true mean for 90% of all possible samples. But are the MPI simulation results normally-distributed? (One might think so because each is the sum of the per-set MPI for thousands of sets over a very long trace.) This is an important question because the student-t confidence intervals assume normally-distributed samples.

To test whether these four-sample confidence intervals are meaningful, with one trace (Mult2.2) and one cache configuration (4-megabyte direct-mapped), a sample of size of size thirty was gathered for (a) random mapping and (b) Hierarchical mapping. For each of (a) and (b), the true mean was estimated by the mean of all thirty simulations (this can be done with reasonable confidence because the thirty-sample 90% confidence interval is (a) ±0.34% and (b) ±1.6% about the estimated mean). All possible unique selections of four samples from these thirty were examined. The four-sample 90% confidence intervals successfully contained the thirty-sample mean (a) 92% and (b) 88% of the time. While this test was only for one trace, it supports the accuracy of these four-sample confidence intervals.

Thus, in addition to the means, the tables which follow give the 90% confidence intervals. The results show that the confidence intervals are small. For example, 71% of the random mapping intervals and 86% the Hierarchical intervals extend less than ±5% from the estimated mean. The random mapping intervals are not quite as tight as Hierarchical because poor random page placement can more greatly vary the simulation *MPI*.

Secondary Cache *MPI*×1000 **Confidence Intervals (Mult1)**						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.8199 ± 0.1879	1.7851	1.5827 ± 0.0258	1.5891	13.0%
	2	1.2648 ± 0.0236	1.2561	1.2093 ± 0.0089	1.2110	4.4%
	4	1.1173 ± 0.0302	1.1061	1.0746 ± 0.0101	1.0772	3.8%
4M	1	0.8083 ± 0.0401	0.7995	0.7299 ± 0.0221	0.7367	9.7%
	2	0.5757 ± 0.0087	0.5747	0.5526 ± 0.0052	0.5534	4.0%
	4	0.5289 ± 0.0033	0.5282	0.5162 ± 0.0007	0.5163	2.4%
16M	1	0.3807 ± 0.0178	0.3782	0.3442 ± 0.0063	0.3433	9.6%
	2	0.2893 ± 0.0024	0.2895	0.2730 ± 0.0020	0.2725	5.6%
	4	0.2716 ± 0.0021	0.2718	0.2619 ± 0.0008	0.2620	3.5%

Secondary Cache *MPI*×1000 **Confidence Intervals (Mult1.2)**						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.7477 ± 0.1878	1.6843	1.6305 ± 0.3856	1.4797	6.7%
	2	1.2540 ± 0.0106	1.2521	1.2035 ± 0.0108	1.2002	4.0%
	4	1.1343 ± 0.0091	1.1362	1.1102 ± 0.0061	1.1103	2.1%
4M	1	0.8880 ± 0.1837	0.8471	0.8596 ± 0.3898	0.6952	3.2%
	2	0.5873 ± 0.0082	0.5863	0.5615 ± 0.0023	0.5617	4.4%
	4	0.5405 ± 0.0026	0.5403	0.5302 ± 0.0014	0.5303	1.9%
16M	1	0.4753 ± 0.1925	0.4043	0.3343 ± 0.0173	0.3362	29.7%
	2	0.2966 ± 0.0033	0.2954	0.2788 ± 0.0051	0.2797	6.0%
	4	0.2797 ± 0.0015	0.2792	0.2703 ± 0.0012	0.2705	3.4%

Secondary Cache *MPI*×1000 **Confidence Intervals (Mult2)**						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.5687 ± 0.1535	1.5419	1.2742 ± 0.0224	1.2707	18.8%
	2	1.0944 ± 0.0217	1.0969	1.0076 ± 0.0036	1.0077	7.9%
	4	0.9749 ± 0.0047	0.9761	0.9417 ± 0.0022	0.9412	3.4%
4M	1	0.7360 ± 0.1067	0.6967	0.6314 ± 0.0230	0.6284	14.2%
	2	0.5462 ± 0.0056	0.5476	0.5212 ± 0.0022	0.5215	4.6%
	4	0.5152 ± 0.0021	0.5153	0.5018 ± 0.0005	0.5018	2.6%
16M	1	0.3352 ± 0.0109	0.3329	0.2931 ± 0.0047	0.2941	12.6%
	2	0.2599 ± 0.0015	0.2597	0.2392 ± 0.0010	0.2390	8.0%
	4	0.2427 ± 0.0022	0.2423	0.2294 ± 0.0003	0.2293	5.5%

Secondary Cache *MPI*×1000 Confidence Intervals (Mult2.2)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.4953 ± 0.1264	1.4873	1.1858 ± 0.0205	1.1884	20.7%
	2	1.0480 ± 0.0186	1.0409	0.9805 ± 0.0049	0.9824	6.4%
	4	0.9593 ± 0.0038	0.9589	0.9289 ± 0.0014	0.9291	3.2%
4M	1	0.7121 ± 0.0428	0.7104	0.5982 ± 0.0062	0.5976	16.0%
	2	0.5375 ± 0.0091	0.5355	0.5101 ± 0.0027	0.5106	5.1%
	4	0.5079 ± 0.0033	0.5088	0.4938 ± 0.0009	0.4940	2.8%
16M	1	0.3339 ± 0.0235	0.3254	0.2820 ± 0.0069	0.2824	15.6%
	2	0.2539 ± 0.0034	0.2542	0.2326 ± 0.0021	0.2333	8.4%
	4	0.2371 ± 0.0041	0.2369	0.2226 ± 0.0006	0.2225	6.1%

Secondary Cache *MPI*×1000 Confidence Intervals (Tv)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	3.3632 ± 0.1861	3.3093	3.2713 ± 0.1084	3.2610	2.7%
	2	2.4658 ± 0.0214	2.4633	2.4919 ± 0.0536	2.4900	-1.1%
	4	2.3055 ± 0.0094	2.3017	2.2997 ± 0.0052	2.3006	0.2%
4M	1	2.1151 ± 0.0549	2.1263	2.0682 ± 0.0268	2.0684	2.2%
	2	1.8294 ± 0.0079	1.8271	1.8108 ± 0.0067	1.8103	1.0%
	4	1.7800 ± 0.0037	1.7802	1.7772 ± 0.0045	1.7764	0.2%
16M	1	1.1899 ± 0.0062	1.1888	1.1589 ± 0.0293	1.1513	2.6%
	2	1.0337 ± 0.0043	1.0337	1.0171 ± 0.0051	1.0170	1.6%
	4	0.9910 ± 0.0027	0.9910	0.9779 ± 0.0031	0.9768	1.3%

Secondary Cache *MPI*×1000 Confidence Intervals (Sor)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	14.9794 ± 0.0211	14.9757	14.8567 ± 0.0284	14.8627	0.8%
	2	14.7033 ± 0.0299	14.6937	14.6402 ± 0.0053	14.6401	0.4%
	4	14.5816 ± 0.0152	14.5774	14.5507 ± 0.0040	14.5496	0.2%
4M	1	9.4718 ± 0.0983	9.4729	8.3389 ± 0.1115	8.3545	12.0%
	2	8.6826 ± 0.0581	8.6821	8.0557 ± 0.1145	8.0312	7.2%
	4	8.3524 ± 0.0291	8.3509	8.0611 ± 0.0323	8.0529	3.5%
16M	1	4.2208 ± 0.3183	4.1976	2.7843 ± 0.1638	2.7574	34.0%
	2	2.8435 ± 0.1069	2.8471	2.1481 ± 0.0824	2.1383	24.5%
	4	2.4179 ± 0.1447	2.3788	2.0743 ± 0.0197	2.0754	14.2%

Secondary Cache *MPI*×1000 Confidence Intervals (Tree)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	3.9061 ± 2.0853	3.3053	2.4720 ± 0.0573	2.4855	36.7%
	2	2.0930 ± 0.0580	2.1085	1.9738 ± 0.0516	1.9561	5.7%
	4	1.8580 ± 0.0494	1.8476	1.8573 ± 0.0584	1.8392	0.0%
4M	1	1.2112 ± 0.2466	1.1478	0.9124 ± 0.0286	0.9174	24.7%
	2	0.7179 ± 0.0292	0.7124	0.5934 ± 0.0284	0.5848	17.3%
	4	0.5292 ± 0.0218	0.5260	0.4932 ± 0.0119	0.4897	6.8%
16M	1	0.5416 ± 0.2284	0.4682	0.3771 ± 0.0292	0.3805	30.4%
	2	0.2914 ± 0.0194	0.2859	0.2706 ± 0.0047	0.2707	7.1%
	4	0.2579 ± 0.0008	0.2578	0.2545 ± 0.0002	0.2544	1.3%

Secondary Cache *MPI*×1000 Confidence Intervals (Lin)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.1481 ± 0.0107	1.1511	1.1609 ± 0.0021	1.1616	-1.1%
	2	1.0806 ± 0.0037	1.0800	1.0938 ± 0.0010	1.0935	-1.2%
	4	1.0611 ± 0.0030	1.0613	1.0766 ± 0.0005	1.0766	-1.5%
4M	1	0.5277 ± 0.0263	0.5264	0.3564 ± 0.0283	0.3488	32.5%
	2	0.3103 ± 0.0392	0.3054	0.1307 ± 0.0234	0.1230	57.9%
	4	0.1557 ± 0.0275	0.1567	0.0494 ± 0.0108	0.0452	68.2%
16M	1	0.1532 ± 0.0230	0.1529	0.0689 ± 0.0063	0.0692	55.0%
	2	0.0416 ± 0.0101	0.0438	0.0217 ± 0.0038	0.0204	47.8%
	4	0.0179 ± 0.0006	0.0178	0.0167 ± 0.0000	0.0167	6.7%

Bibliography

[AGSH86] A. AGARWAL, R. L. SITES and M. HOROWITZ, ‘‘ATUM: A New Technique for Capturing Address Traces Using Microcode,’’ *Proceedings of the 13th International Symposium on Computer Architecture*, 1986, pp. 119-127.

[AGAR87] A. AGARWAL, ‘‘Analysis of Cache Performance for Operating Systems and Multiprogramming,’’ Ph.D. Thesis, Technical Report CSL-TR-87-332, Stanford University, Stanford, CA, May 1987.

[AGHH88] A. AGARWAL, J. HENNESSY and M. HOROWITZ, ‘‘Cache Performance of Operating System and Multiprogramming Workloads,’’ *ACM Transactions on Computer Systems*, vol. 6, no. 4, November 1988, pp. 393-431.

[AGHH89] A. AGARWAL, M. HOROWITZ and J. HENNESSY, ‘‘An Analytical Cache Model,’’ *ACM Transactions on Computer Systems*, vol. 7, no. 2, May 1989, pp. 184-215.

[AGAH90] A. AGARWAL and M. HUFFMAN, ‘‘Blocking: Exploiting Spatial Locality for Trace Compaction,’’ *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 48-57.

[AHHU85] A. V. AHO, J. E. HOPCROFT and J. D. ULLMAN, *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachusetts, 1985.

[BABJ81] O. BABAOGLU and W. JOY, ‘‘Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits,’’ *Proceedings of the 8th Symposium on Operating System Principles*, 1981, pp. 78-86.

[BAEW88] J. BAER and W. WANG, ‘‘On the Inclusion Properties for Multi-Level Cache Hierarchies,’’ *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 73-80.

[BART89] J. BARTLETT, ‘‘SCHEME->C A Portable Scheme-to-C Compiler,’’ Research Report 89/1, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, January 1989.

[BOKL89] A. BORG, R. E. KESSLER, G. LAZANA and D. W. WALL, ‘‘Long Address Traces from RISC Machines: Generation and Analysis,’’ Research Report 89/14, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1989.

[BOKW90] A. BORG, R. E. KESSLER and D. W. WALL, ‘‘Generation and Analysis of Very Long Address Traces,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 270-279.

[BUKB90] H. O. BUGGE, E. H. KRISTIANSEN and B. O. BAKKA, ‘‘Trace-Driven Simulations for a Two-Level Cache Design in Open Bus Systems,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 114-121.

[CAKP91] D. CALLAHAN, K. KENNEDY and A. PORTERFIELD, ‘‘Software Prefetching,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40-52.

[CHCS87] J. H. CHANG, H. CHAO and K. SO, ''Cache Design of A Sub-Micron CMOS System/370,'' *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987, pp. 208-213.

[CLAR83] D. W. CLARK, ''Cache Performance in the VAX-11/780,'' *ACM Transactions on Computer Systems*, vol. 1, no. 1, February 1983, pp. 24-37.

[CLAE85] D. W. CLARK and J. S. EMER, ''Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement,'' *ACM Transactions on Computer Systems*, vol. 3, no. 1, February 1985, pp. 31-62.

[CLBK88] D. W. CLARK, P. J. BANNON and J. B. KELLER, ''Measuring VAX 8800 Performance with a Histogram Hardware Monitor,'' *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 176-185.

[DEGR75] M. H. DEGROOT, *Probability and Statistics*, Addison-Wesley, Reading, MA, 1975.

[DEMA88] R. DE LEONE and O. L. MANGASARIAN, ''Serial and Parallel Solution of Large Scale Linear Programs by Augmented Lagrangian Successive Overrelaxation,'' in *Optimization, Parallel Processing and Applications*, A. KURZHANSKI, K. NEUMANN and D. PALLASCHKE (eds.), 1988, pp. 103-124.

[DENN68] P. J. DENNING, ''The Working Set Model for Program Behavior,'' *Communications of the ACM*, vol. 11, no. 5, May 1968, pp. 323-333.

[DENN80] P. J. DENNING, ''Working Sets Past and Present,'' *IEEE Transactions on Software Engineering*, vol. 6, no. 1, January 1980, pp. 64-84.

[DION88] J. DION, ''Fast Printed Circuit Board Routing,'' Research Report 88/1, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, 1988.

[EASF78] M. C. EASTON and R. FAGIN, ''Cold-Start vs. Warm-Start Miss Ratios,'' *Communications of the ACM*, vol. 21, no. 10, October 1978, pp. 866-872.

[EGKK90] S. J. EGGERS, D. R. KEPPEL, E. J. KOLDINGER and H. M. LEVY, ''Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor,'' *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 37-46.

[ELLG82] R. ELLIS and D. GULICK, *Calculus With Analytic Geometry*, Harcourt Brace Jovanovich, New York, NY, Second Edition 1982.

[FARP89] M. K. FARRENS and A. R. PLESZKUN, ''Improving Performance of Small On-Chip Instruction Caches,'' *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 234-241.

[FERR76] D. FERRARI, ''The Improvement of Program Behavior,'' *IEEE Computer*, November 1976, pp. 39-47.

[FOTH61] J. FOTHERINGHAM, ''Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store,'' *Communications of the ACM*, vol. 4, no. 10, October 1961, pp. 435-436.

[GOOC84] J. R. GOODMAN and M. CHIANG, ''The Use of Static Column RAM as a Memory Hierarchy,'' *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984, pp. 167-174.

[GOOD87] J. R. GOODMAN, ''Coherency For Multiprocessor Virtual Address Caches,'' *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[HAIK84] I. J. HAIKALA, ''Cache Hit Ratios With Geometric Task Switch Intervals,'' *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984, pp. 364-371.

[HeiS90] P. Heidelberger and H. S. Stone, ‘‘Parallel Trace-Driven Cache Simulation by Time Partitioning,’’ IBM Research Report RC 15500 (#68960), February 1990.

[HenP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[HigB90] L. Higbie, ‘‘Quick and Easy Cache Performance Analysis,’’ *Computer Architecture News*, vol. 18, no. 2, June 1990, pp. 33-44.

[Hill87] M. D. Hill, ‘‘Aspects of Cache Memory and Instruction Buffer Performance,’’ Ph.D. Thesis, Computer Science Division Technical Report UCB/CSD 87/381, University of California, Berkeley, CA, November 1987.

[Hill88] M. D. Hill, ‘‘A Case for Direct-Mapped Caches,’’ *IEEE Computer*, vol. 21, no. 12, December 1988, pp. 25-40.

[HilS89] M. D. Hill and A. J. Smith, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions on Computers*, vol. 38, no. 12, December 1989, pp. 1612-1630.

[HwuC89] W. W. Hwu and P. P. Chang, ‘‘Acheiving High Instruction Cache Performance with an Optimizing Compiler,’’ *Proceedings of the 16th International Symposium on Computer Architecture*, 1989, pp. 242-251.

[JoDB87] N. P. Jouppi, J. Dion, D. Boggs and M. J. K. Nielsen, ‘‘MultiTitan: Four Architecture Papers,’’ Research Report 87/8, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, 1987.

[Joup87] N. P. Jouppi, ‘‘Timing Analysis and Performance Improvement of MOS VLSI Designs,’’ *IEEE Transactions on Computer-Aided Design*, July 1987, pp. 650-665.

[Joup89] N. P. Jouppi, ‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU,’’ *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 281-289.

[JoTD89] N. P. Jouppi, J. Y. F. Tang and J. Dion, ‘‘A 20 MIPS Sustained 32 bit CMOS Microprocessor with 64 bit Data Busses,’’ *Proceedings of the 36th International Solid State Circuits Conference*, February 1989.

[Joup90] N. P. Jouppi, ‘‘Improving Direct-Mapped Cache Peformance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364-373.

[KapW73] K. R. Kaplan and R. O. Winder, ‘‘Cache-Based Computer Systems,’’ *IEEE Computer*, vol. 6, no. 3, March 1973, pp. 30-36.

[Kell90] E. Kelly, Personal Communications, 1990.

[KeJL89] R. E. Kessler, R. Jooss, A. Lebeck and M. D. Hill, ‘‘Inexpensive Implementations of Set-Associativity,’’ *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 131-139.

[KilB62] T. Kilburn, ‘‘One-Level Storage System,’’ *I.R.E. Transactions on Electronic Computers*, vol. 11, no. 2, April 1962, pp. 223-235.

[KiEL82] T. Kilburn, D. B. G. Edwards, M. J. Lanigan and F. H. Summer, ‘‘One-Level Storage System,’’ in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell and A. Newell (eds.), McGraw-Hill, 1982, pp. 135-148.

[Krof81] D. Kroft, ‘‘Lockup-Free Instruction Fetch/Prefetch Cache Organization,’’ *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981, pp. 81-87.

[LaPI88] S. Laha, J. H. Patel and R. K. Iyer, ‘‘Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems,’’ *IEEE Transactions on Computers*, vol. 37, no. 11, November 1988, pp. 1325-1336.

[LAHA88] S. LAHA, ‘‘Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems,’’ Ph. D. Thesis, University of Illinois, Urbana-Champaign, Illinois, 1988.

[LARW91] M. S. LAM, E. E. ROTHBERG and M. E. WOLF, ‘‘The Cache Performance and Optimizations of Blocked Algorithms,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 63-74.

[LARU90] J. R. LARUS, ‘‘Abstract Execution: A Technique for Efficiently Tracing Programs,’’ Computer Sciences Department Technical Report, University of Wisconsin, Madison, WI, 1990.

[LIPT68] J. S. LIPTAY, ‘‘Structural Aspects of the System/360 Model 85 II: The Cache,’’ *IBM Systems Journal*, vol. 7, no. 1, 1968, pp. 15-21.

[LILM88] M. J. LITZKOW, M. LIVNY and M. W. MUTKA, ‘‘Condor - A Hunter of Idle Workstations,’’ *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988, pp. 104-111.

[MATI89] R. E. MATICK, ‘‘Functional cache chip for improved system performance,’’ *IBM Journal of Research and Development*, vol. 33, no. 1, January 1989, pp. 15-31.

[MAGS70] R. L. MATTSON, J. GECSEI, D. R. SLUTZ and I. L. TRAIGER, ‘‘Evaluation Techniques for Storage Hierarchies,’’ *IBM Systems Journal*, vol. 9, no. 2, 1970, pp. 78-117.

[MCFA89] S. MCFARLING, ‘‘Program Optimization for Instruction Caches,’’ *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 183-191.

[MILF77] I. MILLER and J. E. FREUND, *Probability and Statistics for Engineers*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition 1977.

[MOGB91] J. C. MOGUL and A. BORG, ‘‘The Effect of Context Switches on Cache Performance,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 75-84.

[MUBB91] T. N. MUDGE, R. B. BROWN, W. P. BIRMINGHAM, J. A. DYKSTRA, A. I. KAYSSI, R. J. LOMAX, O. A. OLUKOTUN, K. A. SAKALLAH and R. A. MILANO, ‘‘The Design of a Microsupercomputer,’’ *IEEE Computer*, vol. 24, no. 1, January 1991, pp. 57-64.

[NIEL86] M. J. K. NIELSEN, ‘‘Titan System Manual,’’ Research Report 86/1, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1986.

[OLMB91] O. A. OLUKOTUN, T. N. MUDGE and R. B. BROWN, ‘‘Implementing a Cache for a High-Performance GaAs Microprocessor,’’ *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 138-147.

[OUHM85] J. K. OUSTERHOUT, G. T. HAMACHI, R. N. MAYO, W. S. SCOTT and G. S. TAYLOR, ‘‘The Magic VLSI Layout System,’’ *IEEE Design and Test of Computers*, February 1985, pp. 19-30.

[PRZY88] S. A. PRZYBYLSKI, ‘‘Performance-Directed Memory Hierarchy Design,’’ Ph.D. Thesis, Technical Report CSL-TR-88-366, Stanford University, Stanford, CA, September 1988.

[PRHH88] S. PRZYBYLSKI, M. HOROWITZ and J. HENNESSY, ‘‘Performance Tradeoffs in Cache Design,’’ *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 290-298.

[PRHH89] S. PRZYBYLSKI, M. HOROWITZ and J. HENNESSY, ‘‘Characteristics of Performance-Optimal Multi-Level Cache Hierarchies,’’ *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 114-121.

[PRZY90] S. PRZYBYLSKI, ‘‘The Performance Impact of Block Sizes and Fetch Strategies,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 160-169.

[PUZA85] T. R. PUZAK, ‘‘Analysis of Cache Replacement Algorithms,’’ Ph.D. Thesis, University of Massachusetts, Amherst, MA, February 1985.

[SAMP89] A. D. SAMPLES, ‘‘Mache: No-Loss Trace Compaction,’’ *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1989, pp. 89-97.

[SHOL88] R. T. SHORT and H. M. LEVY, ‘‘A Simulation Study of Two-Level Caches,’’ *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 81-88.

[SIST88] J. P. SINGH, H. S. STONE and D. F. THIEBAUT, ‘‘An Analytical Model for Fully Associative Cache Memories,’’ IBM Research Report RC 14232 (#63678), November 1988.

[SITA88] R. L. SITES and A. AGARWAL, ‘‘Multiprocessor Cache Analysis Using ATUM,’’ *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 186-195.

[SMIT77] A. J. SMITH, ‘‘Two Methods for the Efficient Analysis of Memory Address Trace Data,’’ *IEEE Transactions on Software Engineering*, vol. 3, no. 1, January 1977, pp. 94-101.

[SMIT82] A. J. SMITH, ‘‘Cache Memories,’’ *Computing Surveys*, vol. 14, no. 3, September 1982, pp. 473-530.

[SMIT85] A. J. SMITH, ‘‘Cache Evaluation and the Impact of Workload Choice,’’ *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985, pp. 64-73.

[SMIG85] J. E. SMITH and J. R. GOODMAN, ‘‘Instruction Cache Replacement Policies and Organizations,’’ *IEEE Transactions on Computers*, vol. 34, no. 3, March 1985, pp. 234-241.

[SMIT86] A. J. SMITH, ‘‘Bibliography and Readings on CPU Cache Memories and Related Topics,’’ *Computer Architecture News*, January 1986, pp. 22-42.

[SMIT87] A. J. SMITH, ‘‘Line (Block) Size Choices for CPU Cache Memories,’’ *IEEE Transactions on Computers*, vol. 36, no. 9, September 1987, pp. 1063-1075.

[SORE88] K. SO and R. N. RECHTSCHAFFEN, ‘‘Cache Operations by MRU Change,’’ *IEEE Transactions on Computers*, vol. 37, no. 6, June 1988, pp. 700-709.

[SOHF91] G. SOHI and M. FRANKLIN, ‘‘High-Bandwidth Data Memory Systems for Superscalar Processors,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 53-62.

[STAM84] J. W. STAMOS, ‘‘Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory,’’ *ACM Transactions on Computer Systems*, vol. 2, no. 2, May 1984, pp. 155-180.

[STAH89] D. STARK and M. HOROWITZ, *Techniques for Calculating Currents and Voltages in VLSI Power Supply Networks*, Computer Systems Laboratory, Stanford University, Stanford, CA, 1989. To be Published in IEEE Transactions on Computer-Aided Design.

[STON90] H. S. STONE, *High-Performance Computer Architecture*, Addison-Wesley, Reading, MA, Second Edition 1990.

[STRE83] W. D. STRECKER, ‘‘Transient Behavior of Cache Memories,’’ *ACM Transactions on Computer Systems*, vol. 1, no. 4, November 1983, pp. 281-293.

[STUF89] C. B. STUNKEL and W. K. FUCHS, ‘‘TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation,’’ *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1989, pp. 70-78.

[STJF91] C. B. STUNKEL, B. JANSSENS and W. K. FUCHS, ''Address Tracing for Parallel Machines,'' *IEEE Computer*, vol. 24, no. 1, January 1991, pp. 31-38.

[TADF90] G. TAYLOR, P. DAVIES and M. FARMWALD, ''The TLB Slice -- A Low-Cost High-Speed Address Translation Mechanism,'' *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 355-363.

[THIS87] D. THIEBAUT and H. S. STONE, ''Footprints in the Cache,'' *ACM Transactions on Computer Systems*, vol. 5, no. 4, November 1987, pp. 305-329.

[THIE88] D. THIEBAUT, ''From the fractal dimension of the intermiss gaps to the cache miss ratio,'' *IBM Journal of Research and Development*, vol. 32, no. 6, November 1988, pp. 796-803.

[THIE89] D. THIEBAUT, ''On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio,'' *IEEE Transactions on Computers*, vol. 38, no. 7, July 1989, pp. 1012-1026.

[THOS89] J. G. THOMPSON and A. J. SMITH, ''Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories,'' *ACM Transactions on Computer Systems*, vol. 7, no. 2, February 1989, pp. 78-116.

[VOMH83] J. VOLDMAN, B. MANDELBROT, L. W. HOEVEL, J. KNIGHT and P. ROSENFELD, ''Fractal Nature of Software-Cache Interaction,'' *IBM Journal of Research and Development*, vol. 27, no. 2, March 1983, pp. 164-170.

[WALP87] D. W. WALL and M. L. POWELL, ''The Mahler Experience: Using an Intermediate Language as the Machine Description,'' *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating System*, 1987, pp. 100-104.

[WABL89] W. WANG, J. BAER and H. M. LEVY, ''Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy,'' *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 140-148.

[WANG89] W. WANG, ''Multi-Level Cache Hierarchies,'' Ph.D. Thesis, University of Washington, Seattle, WA, September 1989.

[WANB90] W. WANG and J. BAER, ''Efficient Trace-Driven Simulation Methods for Cache Performance Analysis,'' *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 27-36.

[WELC84] T. A. WELCH, ''A Technique for High-Performance Data Compression,'' *IEEE Computer*, June 1984, pp. 8-19.

[WOOD90] D. A. WOOD, ''The Design and Evaluation of In-Cache Address Translation,'' Ph.D. Thesis, Computer Science Division Technical Report UCB/CSD 90/565, University of California, Berkeley, CA, March 1990.

[WOHK91] D. A. WOOD, M. D. HILL and R. E. KESSLER, ''A Model for Estimating Trace-Sample Miss Ratios,'' *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 79-89.

[ZIVL76] J. ZIV and A. LEMPEL, ''A Universal Algorithm for Sequential Data Compression,'' *IEEE Transactions on Information Theory*, vol. 23, 1976, pp. 75-81.

[ZIVL78] J. ZIV and A. LEMPEL, ''Compression of Individual Sequences via Variable-Rate Coding,'' *IEEE Transactions on Information Theory*, vol. 24, 1978, pp. 530-536.