Optimizing Hardware Cache to Read-Once Memory Accesses

by

Joshua D. Ain

A Thesis
Submitted in partial fulfillment of
the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

May 22, 2003

Abstract

Hardware memory caches improve memory access speed by taking advantage of the temporal and spatial locality common to most memory references. By assuming that access patterns exhibit these localities, hardware designers can optimize caches design, creating systems in which more than 95% of memory references in an average program need not access main memory.

Exceptions occur in which caches display alternative access patterns which are not efficiently processed by the cache. Read-once memory accesses are one such exception. This exception occurs during multimedia playback and distributed scientific computation internode communication. It results in unnecessary cache pollution and increased cache conflict misses.

We introduce a theoretical modification to the cache called noTime which enables a read-once mode for the on chip cache. This mode disables those cache functions designed to take advantage of temporal locality, while preserving the spatial locality advantages offered by the cache.

We examine the effect of simulated read-once memory accesses on the performance of a standard computer, and the performance of noTime in the same environment. We present results supporting the viability of noTime as a means to improve performance in most situations involving read-once memory accesses.

Acknowledgements

My work on this thesis would not have been possible without the help of one person. Thanks first and foremost to my advisor, Jim Teresco. His encouragement and guidance throughout the year have been invaluable.

Thanks to the all the CS faculty for their feedback, and especially to Duane Bailey, and Tom Murtagh for their advice on this project. Thanks to Mary Bailey for her help whenever I needed anything.

Thanks to Shimon Rura for his support and encouragement in the lab all year, and Brent Yorgey for his feedback. Also thanks to Rachelle Hassan and Adam Ain for coming to support me at every opportunity. Thanks to all those that made the lab a great place to work, especially all the WSO guys.

Finally, thank you to my family for their continued love and support: Dad, Mom, Carolyn, Adam and Becca.

Contents

List of Figures

Chapter 1

Introduction

The hardware cache optimizes memory accesses following standard access patterns, which include both temporal and spatial locality. In instances where memory accesses do not follow normal access patterns, room for improvement in cache performance exists. Instruction references and data references interact in a way not predicted by temporal and spatial localities of reference. To improve performance in this case, we use a split cache. We examine another special case of access patterns, that of read-once memory accesses.

Our work with read-once memory accesses is motivated by the cache effects of message passing on machines running distributed scientific computations. Read-once memory accesses also occur elsewhere, notably in multimedia applications and programs processing configuration files.

We begin by providing some background information to establish a framework for cache analysis, and supply an outline for how systems performing distributed scientific computation function and interact with the cache. We investigate an improvement to hardware design and then compare the performance of the improved system to the performance of the traditional hardware. Our results are based on a selection of SPEC2000 Integer benchmarks with simulated read-once accesses.

Chapter 2

Cache Background

2.1 The Cache

2.1.1 Cache Overview

Since 1987, processor speed has increased by 55% every year, whereas memory speed has increased by 7% per year [11, 17]. These trends are shown in Figure 2.1. When the processor requires data from memory, all processing must halt until the data is loaded into the processor. To bridge this gap and keep processors from sitting idle waiting for memory access, modern day computers utilize one or more small banks of fast memory called a cache. In this section we examine the role and design of the cache. We then survey common means employed through or alongside the cache to improve performance.

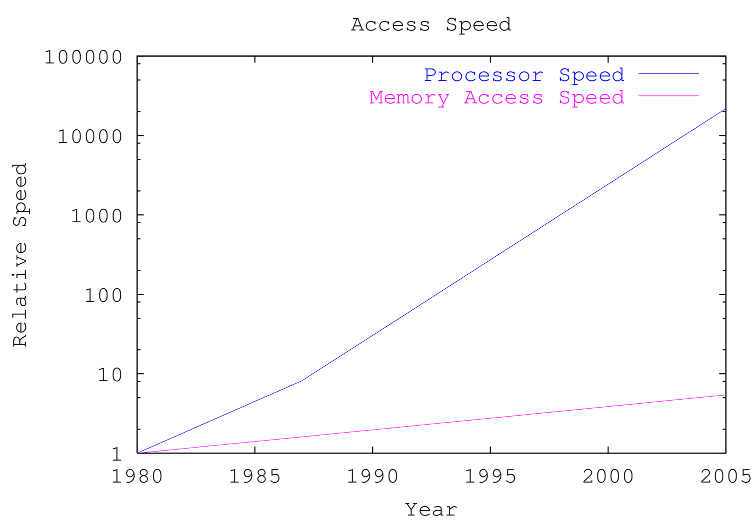


Figure 2.1: Relative memory throughput and processor speed since 1980 [17].

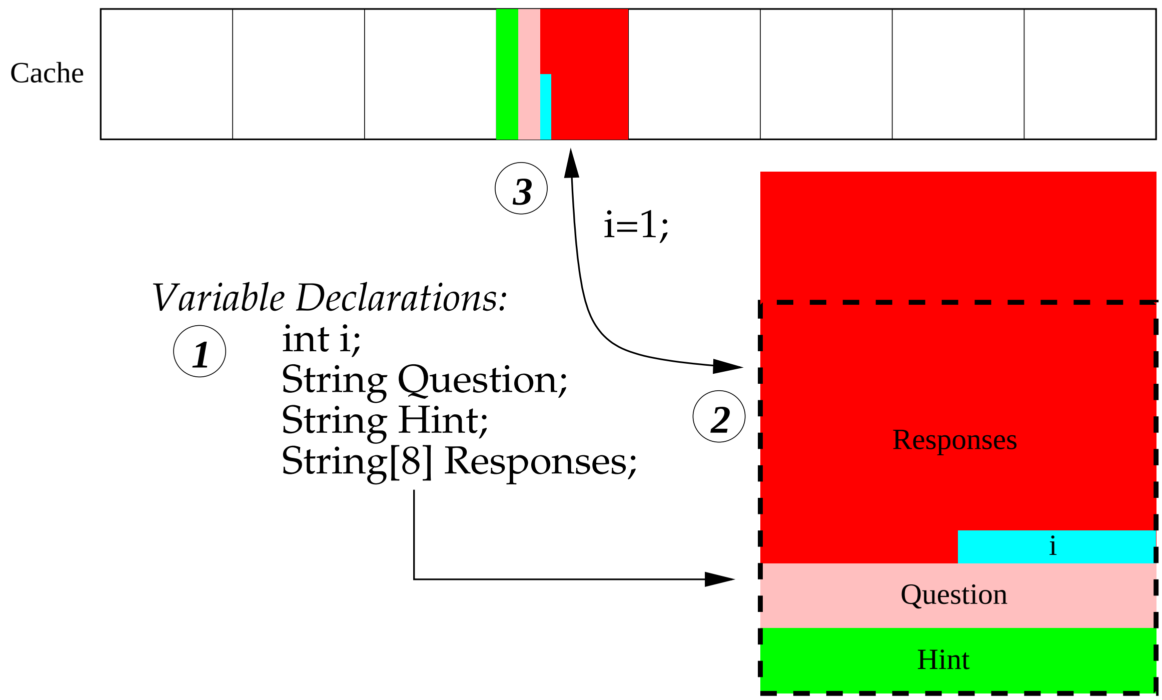


Figure 2.2: Compilers will allocate contiguous memory (2) for program variables (1). This memory is separated into cache blocks. When a variable in the block is first referenced, the entire memory block is loaded into the cache (3).

2.1.2 Locality of Reference

When a program accesses one address in memory, subsequent memory accesses are likely to be close to the address already accessed. This property is called spatial locality. Arrays of data are naturally arranged in this manner, and modern compilers will arrange instance variables and instructions similarly, to maximize the ability of caches to utilize this locality.

When we load any data from memory, we load the entire block surrounding this data (Figure 2.2). We perform this loading by pipelining sequential memory requests through the data bus. These pipelined requests will begin with the block offset of the piece of data requested, and fill the rest of the block afterwards. Since the majority of the access time to memory is spent scanning for the correct address on the disk, fetching the entire block adds only a reduced marginal cost to access time.

Programs typically perform the same actions repeatedly. Most of a program's lifetime is spent in loops, referencing the same variables and executing the same code. This behavior generates memory references with a high degree of temporal locality. We will call the set of instructions and data repeatedly referenced in one of these loops the cache working set. An instruction or piece of data referenced in the cache working set will likely be referenced again in the near future. If the entire cache working set can be simultaneously stored in the cache, the processor can operate without loading data from memory until the completion of the loop, when its cache working set changes (Figure 2.3).

The memory access patterns generated by most programs exhibit both temporal and spatial locality. Caches found in general purpose computers are designed to take advantage

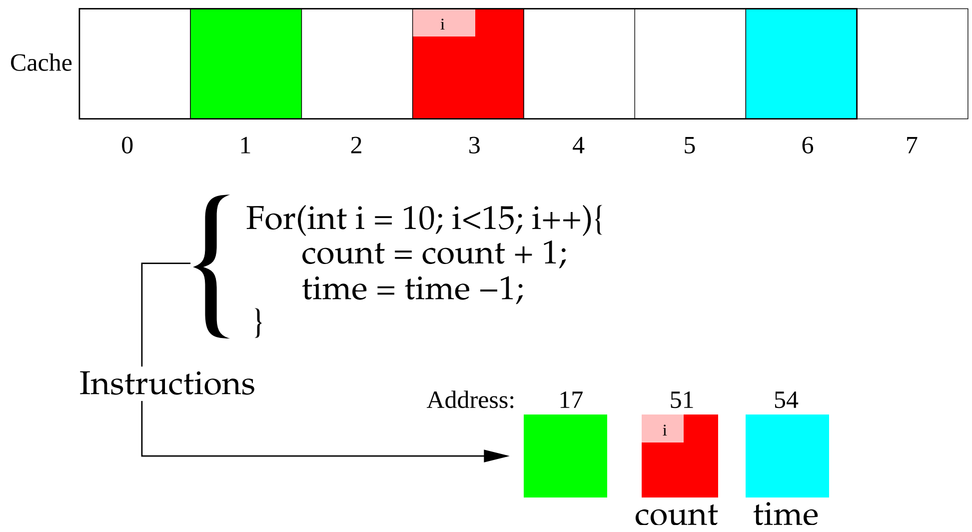


Figure 2.3: Each of these memory blocks is mapped to a cache line through a modular mapping function. After each is loaded into the cache, the loop can execute without referencing main memory because of the temporal locality of the access patterns generated by the loop.

of these localities. To take advantage of the temporal locality of the accesses, we divide our cache into a number of cache lines. In a simple cache, each cache line would hold one cache block. Thus, a 16K cache of this type with a 1k block-size would consist of 16 cache lines. Each block in memory has a block number which can be calculated by examining the x bits of its address larger than the block-size, where x is equal to the log_2 of the cache size (Figure 2.4). Thus, consecutive blocks in memory will be mapped to different lines, and we will not cause conflicts in the cache when we iterate through data.

2.2 Cache Analysis

On cache equipped systems, when a processor receives a memory request, instead of loading the data from memory, it queries the much faster cache. If the word requested lies on the appropriate cache line, a cache hit occurs and the processor loads this word quickly. Otherwise a cache miss occurs and the cache loads the block containing the requested word from memory at a much lower rate, evicting some existing line from the cache. The block remains available in the cache until replaced [20].

This basic cache framework allows the cache to take advantage of both the temporal and spatial locality typical of memory references. The average execution time E of a memory access depends on the hit time to the cache (H), the miss rate of the cache (M_r) and the miss penalty of accessing memory (M_p) and can be represented by Equation 2.1 [2, 17].

$$E = H + M_r * M_p \tag{2.1}$$

Thus, to improve cache speed, we must either reduce the hit time, decrease the miss rate or reduce the miss penalty. A variety of methods improve these three times; no single

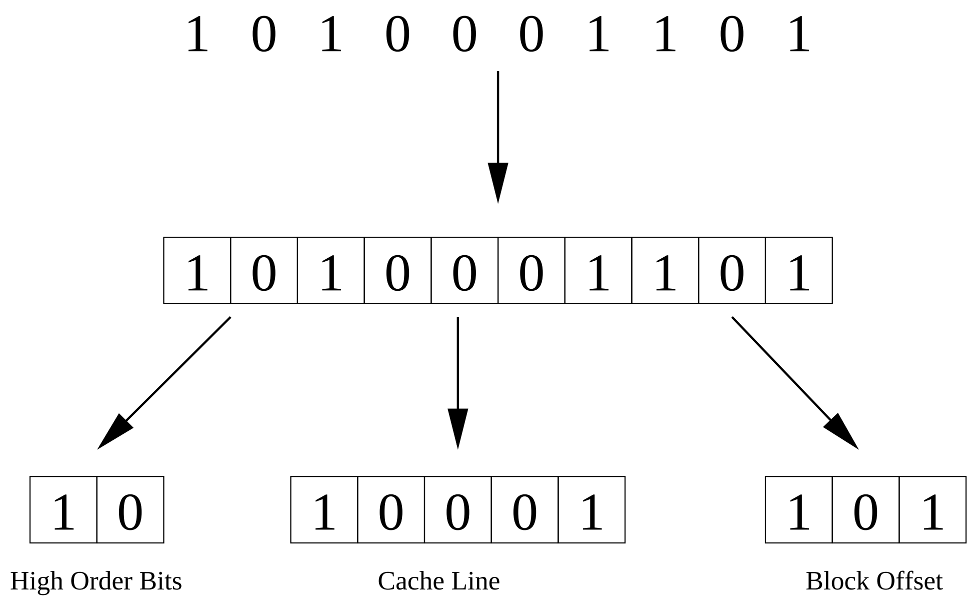


Figure 2.4: A simplified model of the separation of a memory address into cache line, block offset and high order bits. The number of bits used to specify the cache line is equal to log_2 of the cache size.

improvement helps more than one time. A selection of improvements are described below, along with the tradeoffs for their execution effects.

2.2.1 Reducing Miss Rate

Cache misses fall into one of three categories: [17]

- *Compulsory* - The block has not been accessed yet in this execution, and thus cannot have been resident in the cache.
- *Capacity* - The cache is not large enough to contain all blocks necessary for execution; this miss is generated because the block referenced has been discarded due to a full cache.
- *Conflict* - Another block is mapped to the same location, forcing the non-full cache to discard this block rather than discarding another more suitable block.

Associativity

To decrease the miss rate of the cache, we can increase the associativity of the cache. The associativity measures the number of different cache lines in which a given memory address may reside. For example, the cache discussed above was a 1-way set associative cache, also known as a direct mapped cache. Associativity increases by powers of two. In a fully associative cache, any memory block can be mapped to any cache line. In practice, an 8-way set associative cache is used in place of a fully associative cache.

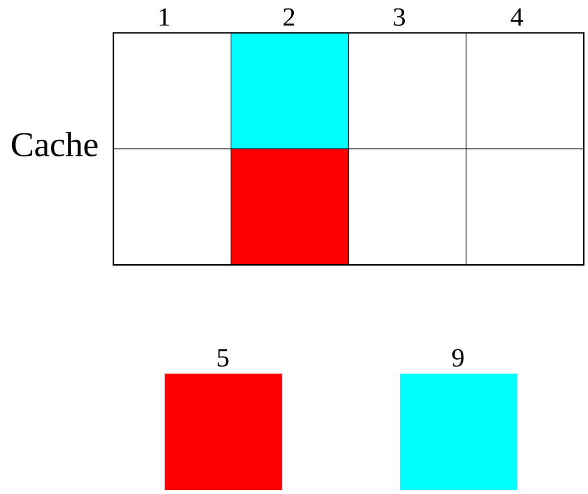


Figure 2.5: In a two way set associative cache, each memory block may be mapped to one of two cache lines. This more flexible mapping allows the cache to avoid some conflict misses at a cost of increased cache hit time.

When the processor requests a memory address from a 2-way set associative cache, the memory block maps to two cache lines. The processor checks each of these cache lines for the memory block, then loads the block from memory if it is not available in either line.

Associative caches decrease the cache miss rate by reducing the number of conflict misses, but increase the cache hit time. Many first level caches are direct mapped, but second level caches (Section 2.2.2) and other caches have higher associativities if the performance tradeoff is suitable [14].

Replacement Policy

In the simplest mapping, a direct mapped cache, each memory block can only be mapped to one cache line, making replacement policy choice irrelevant. All caches which are not direct mapped require a more complex replacement policy scheme to determine which line to replace. These replacement policies are referenced whenever a block can map to multiple cache lines. Three common replacement policies are listed below.

- *Least Recently Used (LRU)* - the least recently accessed line is replaced.
- *First In First Out (FIFO)* - The longest lived line is replaced.
- *Random* - A randomly chosen line from those eligible is replaced.

LRU is the most effective policy; however, it increases the execution time of the cache. Random is fastest but less effective. FIFO is less effective than either of the others. Random and LRU are most commonly used.

Split Cache

Programs can be divided into a stream of instructions performed on a stream of data. The cache working set of the data is often much larger than the cache working set of

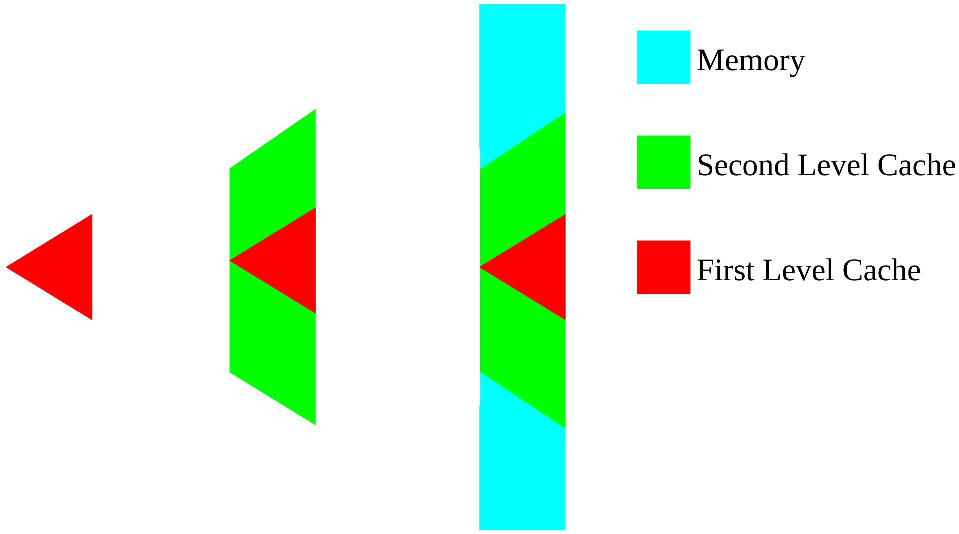


Figure 2.6: Each level of the memory hierarchy contains all levels above it. In this diagram, the second level cache encompasses all data stored in the first level cache, and main memory encompasses all data stored in the second level cache.

the instructions. This trend has the unfortunate side effect of evicting instructions from the cache through conflict misses in favor of high-throughput data when the instructions evicted may still possess temporal locality.

To account for this important and common case, a split cache, also known as a Harvard cache, is often used. A split cache provides a separate instruction cache and data cache. A split cache can also take advantage of the fact that instructions are never written to memory. Thus, a split cache need not check whether an instruction cache access is a read from or a write to memory [17].

2.2.2 Reducing Miss Penalty

Second Level Cache

A second level cache can be added to reduce the penalty incurred for a first level cache miss. The data contained in the second level cache is often a superset of all data contained in the first level cache (Figure 2.6). A typical arrangement might use a 16k first level cache and a 256k second level cache. Because of the trade-offs associated with the less accessed second level cache, it is frequently 2-way set associative. Access to the second level cache takes longer than access to the first level cache, but less time than access to memory. When a memory request misses on the first level cache, it has a chance to hit in the second level cache before being forced to access memory.

We can refine our calculation of average memory access time when using a second level cache by expanding the term M_p from Equation 2.1. The average execution time of a memory access now depends on the hit time to the cache (H_1), the miss rate of the first level cache (M_{r1}) the miss penalty of accessing the second level cache (M_{p1}), the hit time of the second level cache (H_2), the miss rate of the second level cache (M_{r2}), the miss penalty of accessing memory (M_{p2}) [2, 17]:

$$E = H_1 + M_{r1} * (M_{p1} + H_2 + M_{r2} * M_{p2}) \tag{2.2}$$

Or, if we incorporate H_2 into M_{p1} we can simplify this equation to:

$$E = H_1 + M_{r1} * (M_{p1} + M_{r2} * M_{p2}) \tag{2.3}$$

2.2.3 Reducing Hit Time

Write Policy

There are two write policies for caches: write-through and write-back. The hit time of the cache can be decreased by writing values back to memory from the cache only when the block will be removed from the cache. A cache block which has been changed but not yet written back to memory is called a dirty block. This system, called a write-back cache, improves performance but adds complexity. Caches that write to memory immediately are called write-through caches [17]. The two protocols can be treated equivalently for our cache evaluations. All references to values stored in memory can thus be taken to mean values stored in memory or in a dirty block of a cache.

Simplicity

Small caches and simple caches both allow developers to decrease the hit time of a cache. Increased complexity adds to development time and makes cache designs less extendible. For these reasons, first level caches are both small and direct mapped.

2.2.4 Multiprocessor Shared Memory Caching

In multiprocessor shared memory systems, some additional caching complications arise. Consider a two processor system. On this system, processor A and processor B each has its own distinct first level cache. Assume each of them is caching memory block C. If processor A writes to block C, the values of C stored processor A's cache and processor B's cache are no longer consistent with each other, as shown in Figure 2.7. In the figure, cache A and cache B both sequentially load and write to a specific block in memory. When B changes the value in memory after A has already cached the previous value, A's cache block becomes inconsistent with the value in memory. When A makes a change, it ignores B's previous change – leading to the incorrect evaluation, $5 + 1 - 1 + 1 = 7$.

To prevent such a situation, a cache coherency protocol must be used to determine appropriate invalidations. Three cache coherency protocols are described below.

An inefficient protocol involves partitioning the memory space such that none of the memory is shared between processors. This method avoids the coherence issue, but eliminates any benefits of shared memory.

A snooping protocol broadcasts an invalidation over the CPU bus after writing a block to memory, invalidating all instances of the memory block in question in other caches. A variation of this protocol updates the cache blocks with the correct value. This method requires a bus wide enough to transmit the data efficiently and is thus used infrequently [2]. This approach must monopolize the bus for each invalidation sent, limiting the scalability.

Another method, called a directory-based coherence protocol, entails connecting each cache to a hardware directory that keeps track of which caches have copies of the blocks loaded into any cache. The directory records a state for each block, representing whether the block is in an uncached, shared, or exclusive state. A cache may only write to a block that is in the exclusive state. In order to move a block from a shared state to an exclusive state, the directory invalidates all other copies of the cache block. All cache blocks in a shared state are guaranteed to be coherent with the values in memory, although blocks in an exclusive state are not guaranteed to be. When another cache needs to read an exclusively owned block, the directory instructs the cache to store its dirty value to memory if necessary, and changes the state of the block in the directory to shared when the memory store is completed. The directory coherence method scales better than the snooping protocol, but may not run as efficiently on a smaller number of processors due to the overhead of the directory [2, 17]. Versions of the directory-based coherency protocol have been extended to use virtual memory support, and perform competitively to a snooping protocol on multiprocessor programs whose cache working sets can be fit into the cache [18].

2.3 Illusion of Uniform Access Memory

In early computing, Uniform Access Memory (UAM) was truly uniform access in that all memory accesses took a standard amount of time. With the addition of one or more levels of cache, however, access to different portions of memory can take vastly different amounts of time, depending on the cached or uncached state of the memory referenced.

UAM continues to be a valuable model even when access is non-uniform because it abstracts away the details of cache operation, allowing programmers to think about their programs as using a uniform access memory. However, by programming for a UAM model, programmers cannot design access patterns to optimize the average access time. Programming without a UAM model would require such a micromanagement of resources that, while it might improve the speed of program execution, it would do so at such a cost to development time as to be prohibitive.

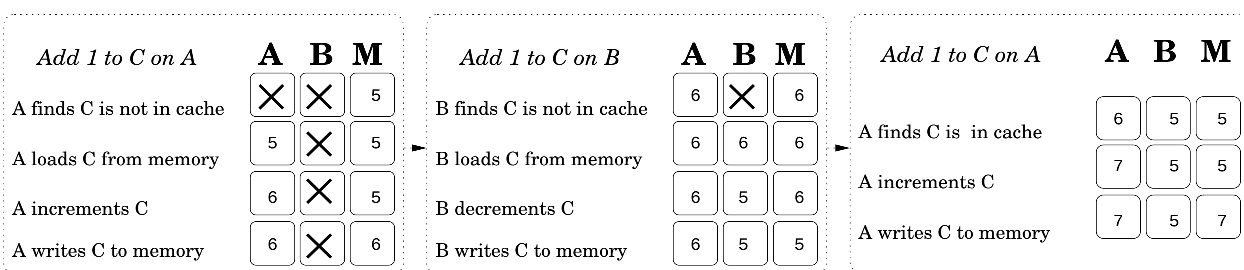


Figure 2.7: An example showing an incorrect addition on a cache incoherent multiprocessor shared memory machine. A, B and M represent the values for block C in cache A, cache B and memory, respectively. The X represents an invalid cache block.

Chapter 3

Motivation Through Parallel Scientific Computation

3.1 Parallel Scientific Computation Execution

Scientific computation has become an important tool to solve large problems in disciplines such as mathematics, physics, and engineering. Many of the problems that scientists and engineers are interested in solving are too large for even the fastest single processor to compute in a reasonable amount of time, necessitating Parallel Scientific Computation (PSC). Parallelism adds many complications to a computation, including the need to distribute the computation across cooperating processors and to coordinate the computation among them.

An important class of PSCs involve the solutions to systems of partial differential equations (PDEs). Popular methods, such as finite element, finite volume, and discontinuous Galerkin methods, utilize a mesh that represents a discretization of the domain. Packages such as the Algorithm Oriented Mesh Database (AOMD) [21] provide a standard set of mesh data structures and query and update methods on those meshes. Parallelism of a mesh-based computation is often accomplished by decomposing the domain into a number of domains equal to the number of available processors. Zoltan [9], from Sandia National Laboratories, includes a suite of tools that may be used to achieve this partitioning.

To understand how a typical PSC progresses, we will examine the three-dimensional compressible flow in a cylinder containing a cylindrical vent perpendicular to the cylinder as described by Flaherty [10]. This problem is motivated by flow studies in perforated muzzle brakes for large calibre guns. The computational domain is a portion of the inside of the cannon barrel and the interior of the hole. We initiate the problem by rupturing a hypothetical diaphragm between the two cylinders. The shock tube initially contains air flowing at Mach 1.23 and the smaller cylinder (the vent) is quiet [10].

A discretization of the domain, the mesh, is partitioned using, for example, one of the Zoltan partitioning tools to distribute mesh elements among the processors. As the computation progresses, each element is assigned new solution values at each step determined entirely by its current state and that of its neighbors. Communication is needed to ex-

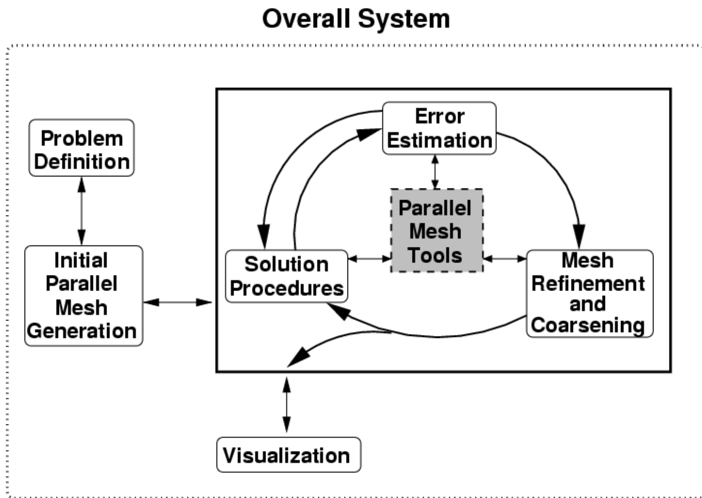


Figure 3.1: The execution states of a typical parallel adaptive computation. After initial mesh partitioning, the computation loops between the solutions procedures and the error estimation until the estimation exceeds a certain bound. The mesh is refined and coarsened, then returns to the loop.

change solution values for elements at partition borders. Each step is accepted or rejected based on whether an estimate of the elemental error exceeds a prescribed tolerance for solution accuracy. Following a rejected time step, the mesh is refined adaptively by adding vertices to parts of the domain where the error was high, forming smaller elements that can more accurately capture the behavior of the system. Areas of very low error may be coarsened through removal of vertices to avoid performing more computation than necessary. After refinement and coarsening, the meshes must be rebalanced with a dynamic load balancing tool.

In the perforated muzzle brake example, the mesh is refined in areas of the domain where the solution exhibits complex behavior. The process of computation is outlined in Figure 3.1. In each time step, if error estimates are within acceptable boundaries, the execution loops back to solution calculation. If the error exceeds these bounds, we refine and coarsen the mesh. The visualization is available at any point. A graphical representation of the mesh and its progression is shown in Figure 3.2.

Many parallel adaptive computations follow a similar pattern of execution. The only interprocessor communication necessary occurs at partition boundaries to exchange neighbor solution information, and during rebalancing, which entails the computation of a new decomposition, and migration of the mesh and corresponding solution to achieve the new decomposition.

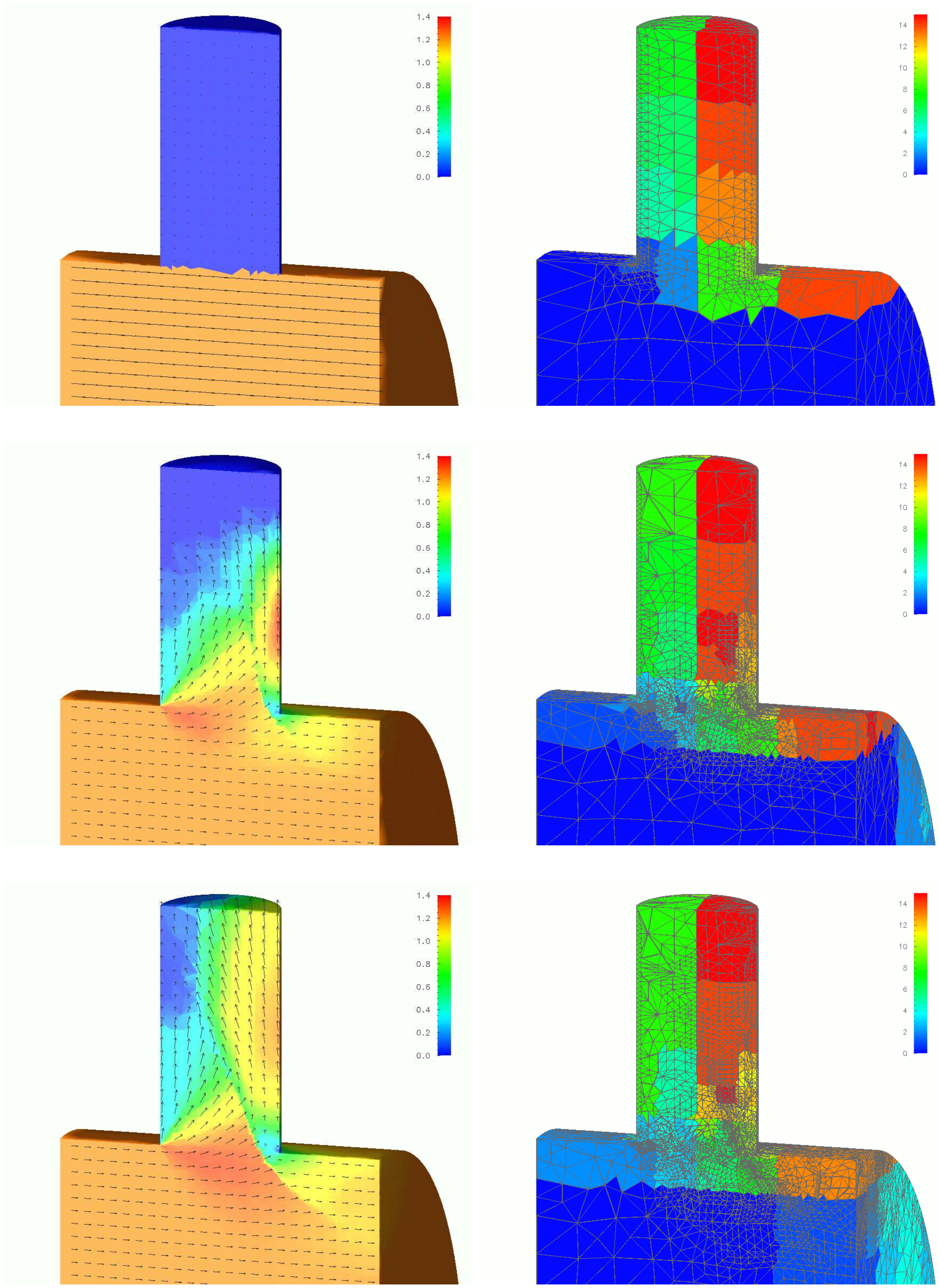


Figure 3.2: Projections of the Mach number and velocity vectors (left) and mesh and partitioning (right) onto the surfaces of a perforated cylinder at times 0 (top), 0.3 (center), and 0.6 (bottom) [10].

3.2 Message Passing

3.2.1 Overview

On a distributed memory computer, interprocessor communication requires explicit message passing. In the past, the burden for determining how to handle this message passing lay on the programmer. A variety of systems for message passing existed, often specific to a single type of system: Chameleon [13], Intel's NX/2 [19] and Chimp [7, 8]. These disparate application programming interfaces (APIs) made parallel program development difficult, and required that message passing calls be rewritten to move a parallel program from one type of system to another. Furthermore, the proprietary nature of the communication made message passing between two dissimilar systems difficult [1].

The Message Passing Interface (MPI) was developed to provide a standard API for message passing systems to implement. By using the MPI standard, applications are portable among any of the systems that provide an implementation of the standard [12].

3.2.2 Caching with Message Passing

Both read-once memory accesses and shared memory cache access issues occur in message passing programs. We will examine the effect of read-once memory access in more depth.

The memory access patterns of a message-passing program are likely to exhibit temporal and spatial locality similar to a non message-passing program, except when a message send or receive is issued. In these cases the memory accesses no longer have any temporal locality. The data from the message buffer, if large enough, will flush the cache of information which did have a potential for temporal locality. All information that could have been reused from the cache will have to be reloaded from memory, decreasing effective access time upon completion of the send or receive. Studies of cache conscious reorganization on a single processor node have shown 30% improvement in the data cache [6]. A cache designed specifically for message passing programs and other programs with read-once memory modes could improve the performance of these programs in the read-once sections on a similar scale.

For instance, assume a processor is executing a message passing program which can fit its entire cache working set into its data cache during normal computation. If the processor receives a large message, each block of the message must pass through the cache to be copied to its final location. This process can wipe the cache of the data it previously contained, leading to unnecessary conflict misses. The communication in the perforated muzzle break example would have this sort of effect on each processor during each communication phase.

Chapter 4

Problem Domain

4.1 Read-Once Memory Access

In our examination of the cache we stressed how the cache optimized performance by making assumptions about the spatial and temporal locality of memory accesses. We examined one example in which the access patterns did not conform to these assumptions: that of instructions and data (Section 2.2.1). A split cache is often used to account for this exception, and to achieve improved performance.

Message passing parallel programs are another situation in which the memory accesses do not necessarily conform to the assumptions made by the cache. These differences occur when the parallel programs pause normal execution to perform interprocess communication. Whenever messages are sent or received, the processor iterates over the message datastream. These memory accesses lack any temporal locality. We will refer to this type of memory access pattern as a read-once memory access.

When processing music, video, or other types of multimedia, the data is input and examined in a largely linear fashion, providing another example of read-once memory access. Configuration files are similarly processed only once.

Recall that caches take advantage of spatial locality by loading entire blocks of information, and of temporal locality by storing blocks of memory in separate cache lines after the initial reference (Section 2.1.2). When information examined has no temporal locality, using these lines provides no advantage. Storing the information in the normal manner removes the blocks previously stored in those lines. These previously stored blocks may still be part of the cache working set of the program; by removing them from the cache we are forcing a subsequent conflict miss.

By disabling the cache during read-once memory accesses, we can prevent these unnecessary cache misses. However, the read-once memory accesses do possess spatial locality. Disabling the cache would not allow us to take advantage of this locality.

Instead we propose disabling only the aspect of the cache which takes advantage of temporal locality: the lines. If, once we have finished referencing a block, we will not need that block again, we can use the same cache line for the next block regardless of where the block would normally map, insuring that we are overwriting information that we do not need. This method does not guarantee that the information preserved within the cache is useful, but it does guarantee that the information removed from the cache is not useful.

4.2 Cache Improvement

We propose a theoretical hardware improvement to the cache which temporarily disables the cache features designed to take advantage of temporal locality. This theoretical new hardware, which we call noTime, is designed to optimize memory accesses with no temporal locality.

NoTime supports a cache mode in which all data cache blocks loaded are assigned to a single location. This added functionality is built on top of normal caching hardware and can be activated by an environment bit. This bit is either set manually by the user, or automatically through the compiler. Alternatively, noTime could activate based on a specified bit in memory addresses.

NoTime modifies the mapping function of a normal cache such that when activated, the mapping function always returns line 0 as the correct mapping. When noTime is inactive, normal mapping results are returned.

A separate one-line buffer could be used instead of noTime with similar or superior results. However, this buffer would share most characteristics of a cache line, resulting in a duplication of this functionality. Using an already existing line in the cache, as noTime does, reduces the added complexity and provides nearly as much benefit as creating a separate one-line buffer would. Furthermore, all results we discuss with noTime apply equally to a one-line buffer, whereas results from a one-line buffer would not apply to noTime.

NoTime breaks the illusion of UAM as a programming model, bringing both greater power and greater complexity to the end programmer. As such, use of noTime as a manual system must be weighed against side-effects of breaking the illusion of UAM from the programmer's perspective. The greater power brought by noTime for most programs is outweighed by the added complexity of micro-management necessary without the UAM model. With compiler support, by augmenting a message-passing library, or in certain clearly evident situations however, the illusion of UAM could be largely preserved for the end user.

Manually activating and deactivating noTime to optimize performance on array iteration is also not as simple as it seems. While each iteration through an array will include a number of read-once accesses, it will also include a reference to an array index, which does possess temporal locality. For this reason, most implementations of noTime would likely require compiler support.

4.3 Formal Framework

In the following sections, we examine the effect of read-once memory accesses on normal computation. We focus specifically on a simulation of the class of access resembling data

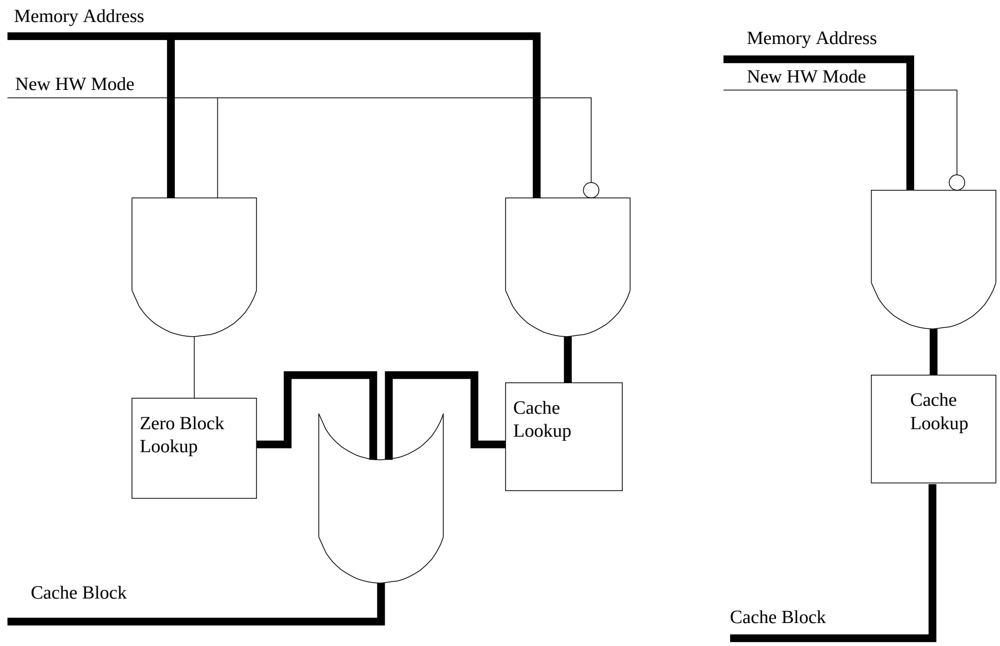


Figure 4.1: Proposed new cache design. On the left, we see the full implementation of the new cache, in which the normal mapping and the noTime mapping are merged depending on whether the new hardware bit is active or not. This version more explicitly demonstrates the changes made, but may be simplified to the version on the right for increased efficiency. In this model, the operations performed by the gates connected to the zero block lookup in the original diagram are now performed by the failure of the And gate when the new hardware bit is off in the second diagram.

accesses encountered in a message passing program, but the results we generate should generalize to other types of read-once accesses.

In our examination of message passing programs, we make a number of simplifying assumptions. We assume for the purposes of discussion that accesses of this type can be identified perfectly either by the user or the compiler. This is a significant assumption which allows us to calculate the best case performance statistics. By assuming perfect knowledge, we can determine the possible room for improvement over current performance. Future research topics include developing heuristics to distinguish read-once memory access which will allow performance with imperfect knowledge to converge on the values found using perfect knowledge.

Our simulator also uses a simplified processor model which performs no prefetching of blocks. Results using this simplified model should extend gracefully to cache models which do perform prefetching. A model which performs three block prefetching would simply map fetched blocks to one of three cache lines sequentially. This approach would decrease first level cache hit rates, but would have a lesser effect on the second level cache as discussed in Chapter 6. Performance with prefetching would follow the trend shown by our results without prefetching.

We model our read-once memory accesses by assuming a uniform frequency and uniform read-once memory access string length. This assumption may not always be true, but we were unable to access or generate accurate memory traces of message passing program execution (As detailed in Section 5.1), and have found that this model provides a good approximation of program execution. We also performed an evaluation of the address traces we acquired which grounds our finding from the simulated read-once accesses with results from real read-once accesses.

These assumptions are summarized in below:

- Read-once memory accesses arrive at a uniform frequency and last a uniform length
- Read-once memory accesses can be identified by programmer or compiler

In all of our simulations, we examine only the data cache, as the data cache is the only place where these read-once memory accesses occurred in our examples.

Chapter 5

Cache Simulation

In order to study the effects of memory access patterns on cache efficiency, we must model the cache. A variety of cache modeling software packages exist, but none provide the versatility required for the experiments described in Chapter 6. Thus, we opted to design our own cache simulator.

5.1 Address Traces

A cache simulator requires information about the hardware of the modeled system, and address trace information from the target program. The hardware information is readily available, but acquiring address traces is a more difficult issue.

An address trace is a sequential listing of memory accesses. With this memory listing, we can simulate the behavior of multiple cache levels and determine the hit rate on each. We may also analyze access traces in search of further patterns we use to optimize performance.

Acquiring address traces consists of intercepting memory requests at a low level of the computer architecture, recording the access, and then allowing the request to proceed. We will examine three trace recording methods.

5.1.1 WARTS

The Wisconsin Architecture Research Toolkit (WARTS) contains a variety of tools built for cache analysis and other low level operating systems research [15]. Of note to us is the Executable Editing Library (EEL) which allows the user to modify executables after compilation by modifying the control flow graph or inserting foreign code snippets. Using EEL, executables could be modified to record all memory accesses during normal execution [16]. Unfortunately, EEL and WARTS are no longer supported and could not be compiled on modern systems.

5.1.2 ATUM

Address Tracing Using Microcode (ATUM) performs the tracing at the lowest level possible without custom hardware. ATUM is possible on any machine with writable microcode. Typically, only a small number of micro-routines generate all memory addresses. These micro-routines can be modified to record addresses and some type information (such as read, write or instruction fetch) before performing normal operations. This type of recording can be done without a large slowdown of execution, but some slowdown does occur. As a result, ATUM does not report precisely accurate results, but when developed, its results were much more accurate than alternatives [3]. We were unable to use ATUM, as it was incompatible with our available processors.

5.1.3 BYU

The Performance Evaluation Laboratory at Brigham Young University (BYU) have developed a hardware-based approach to gathering address traces. In their approach, they attach Tektronix TLA720 logic analyzers to the pins of the CPU. These logic analyzers record the electrical signals representing memory access requests output by the chip. When the analyzer's memory buffer is full, it sends a pause signal to the computer via its parallel port. The computer goes into a tight waiting loop while the analyzer uploads its data to a storage machine. Once the data is uploaded, the remote machine extracts the required bits, and packs them in the proper format. The logic analyzer then prompts the computer to continue processing [5].

5.1.4 Conclusion

Acquiring address traces is a research field in its own right. Since we are interested more in simulating cache execution using address traces than in the methodology of trace acquisition, we chose to use traces from the BYU trace archive [4]. The BYU archive is a large collection of traces generated using the BYU logic analysis methods. We chose memory traces from executions of the "crafty", "bzip", "eon" and "mcf" SPEC2000 integer benchmarks [22]. These span a variety of access patterns.

5.2 Benchmark Details

The SPEC2000 benchmark is a caching benchmark set composed of a variety of execution types. These benchmarks display a range of access patterns, as shown in Table 5.1 and Figure 5.1.

- Crafty

  Crafty is a search program that uses 64-bit word to investigate the movement progression of a chess board from an initial configuration. Crafty uses a large number of logical operations [24].

Benchmark Name	1-way Associative	2-way Associative	8-way Associative
Crafty	90.9%	91.3%	91.5%
Bzip	89.7%	96.8%	99.8%
Mcf	86.0%	92.4%	93.3
Eon	98.9%	99.3%	99.5%

Table 5.1: Table comparing benchmarks with first level cache hit rates. The cache results occur when the benchmark is executed with an 16k first level cache and show variance with change in associativity.

- Bzip

  The bzip benchmark uses the bzip program to compress and decompress different sample files varying from a large tiff file, a binary program, and a source file, all jobs that bzip might commonly encounter [23]. We used the bzip tiff benchmark.

- Mcf

  The mcf benchmark simulates the set of scheduling problems occurring in the planning process of public transportation companies. This benchmark uses integer arithmetic almost exclusively [26].

- Eon

  The eon benchmark is a probabilistic ray-tracer. The computational demands of the program involve large numbers of vector intersection, similar to the pattern of many conventional ray-tracers. Eon has less memory coherency than other ray-tracers because of the random nature of the rays generated. The rendering algorithms used are Kajiya, Cook, and Rushmeier [25]. We used the Cook algorithm trace for our tests.

5.3 Simulation

To analyze the address traces we acquired, we designed and built the Caching for Read-once Accesses Simulation System (CRASS). CRASS is a cache simulator written in C++ that takes a BYU trace file as input and simulates cache execution for a specified number of data reads or writes. Non-data memory accesses are ignored, as our focus is only on data accesses.

CRASS simulates the population of both first and second level cache, each of which can have a variable size, associativity and replacement policy. Currently supported replacement policies include random, first in first out (FIFO) and least recently used (LRU). The modular design of CRASS facilitates the addition of further replacement policies.

CRASS also supports the insertion of simulated read-once memory accesses. The interval of occurrence y and length x of these interruptions are parameterized. After the specified interval y, CRASS halts execution of the trace file and instead begins incrementally requesting x cache blocks from the address where the previous interruption stopped.

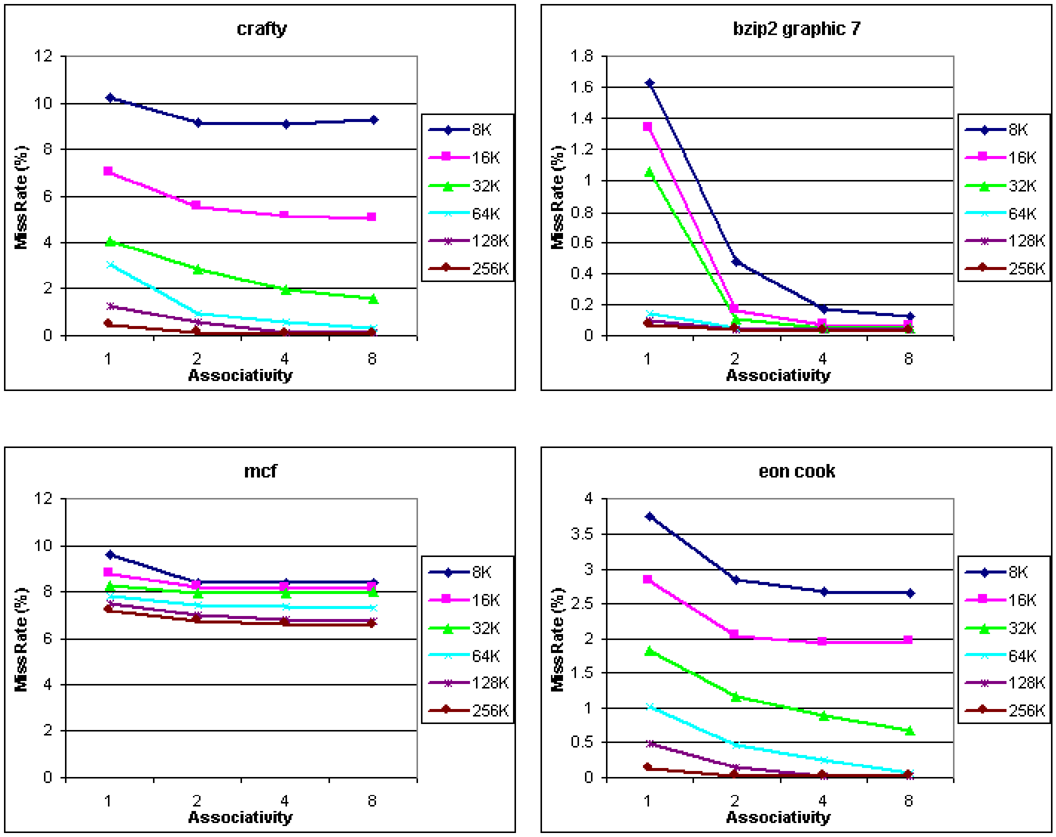


Figure 5.1: Cache associativity of benchmarks as compared against first level cache miss rate on each benchmark [4]. Each line represents performance on a cache of indicated size.

Simulating these interruptions allows us to study how read-once memory accesses affect system performance.

CRASS can also be used to simulate an implementation of noTime. When this cache mode is activated, all memory accesses treat the cache as possessing only its first line. When the mode is deactivated, all other lines still contain their previous residents.

For further information, see the CRASS documentation in Appendix A.

Chapter 6

Interruption Effects

In this chapter, we examine the results of the SPEC2000 crafty benchmark when run with strings of simulated read-once accesses inserted. We will call these strings of accesses interruptions. We ran the benchmarks against a variety of tests of different interruption intervals, interruption lengths, and cache associativities. We chose to perform our initial examination on the crafty benchmark, as it had a wider range in its cache hit percentage on variable associativities than the other benchmarks. We examine the other benchmarks as their results differ from that of crafty. The full results from the other benchmarks can be found in Appendix B.

We used as standards of comparison both cache hit rates and cycles per access. Cycles per access (CPA) are computed through Equation 2.3 as discussed in Section 2.2.2:

$$CPA = H_1 + M_{r1} * (M_{p1} + M_{r2} * M_{p2}) \tag{6.1}$$

We used a cache hit penalty H_1 of 1, a second level cache hit penalty M_{p2} of 30, and a second level cache miss penalty M_{p2} of 120 [17].

The first level cache hit rate and the CPA of each benchmark are shown in Figure 6.1.

6.1 Emptying the Cache

The following tests investigated the effect that the interruption interval and interruption length had on efficiency of computation, both with and without noTime. These tests investigated the effect of changing one parameter while holding the other constant. These tests used a direct mapped 16k first level cache, a 256k 2-way set associative fifo second level cache, and a 1k block size. We will discuss the results of these tests, and our conclusions below. We used a fifo replacement policy for all tests.

6.1.1 Variable Interval

In the variable interval test, we held the interruption length constant while varying the interval from 0 to 1000 memory accesses. When comparing this interval to the first level cache hit rate, the length of the interruption, if longer than that of the first level cache, was unimportant (Figure 6.2).

The interruption length did play a role in the second level cache hit rate and CPA when the interruption length was sufficient that the first level cache was flushed but the second level cache was not. The first level cache is flushed when a number of consecutive interruption blocks are read equal to its size (Section 2.1.2).

The CPA results for the same variable interval tests (Figure 6.3) show a rightward shift in the graph when a larger length is used in tests without noTime, representing the globally higher CPA with this larger length. This shift stops increasing in our tests when the size of the length exceeds that of the second level cache as shown in Figure 6.4. On a real machine the shift would continue until the interruptions exceeded the maximum size of each step down the memory hierarchy. Other benchmarks yielded similar results.

6.1.2 Variable Length

Further examination of interruptions with variable length and a constant interval confirms this analysis. Figure 6.5 shows the first level cache hit rate, which has a sharp downward slope until the cache has been flushed for the given interruption, at which point the line flattens. Again the CPA shown in Figure 6.6 tells a more complete story, displaying a sharp slope initially until the first level cache is flushed, then a shallower slope, flattening after the second level cache is flushed. Other benchmarks yielded similar results.

6.1.3 Interpretation

These tests demonstrated that performance in an environment with interrupts is affected up until the point where the size of the interrupts exceeds the size of the cache, thus bringing on a worst case scenario equivalent to a cold start. In each of these examples, the performance of the simulation using the new hardware converged on that of the benchmark simulation running with no interruptions.

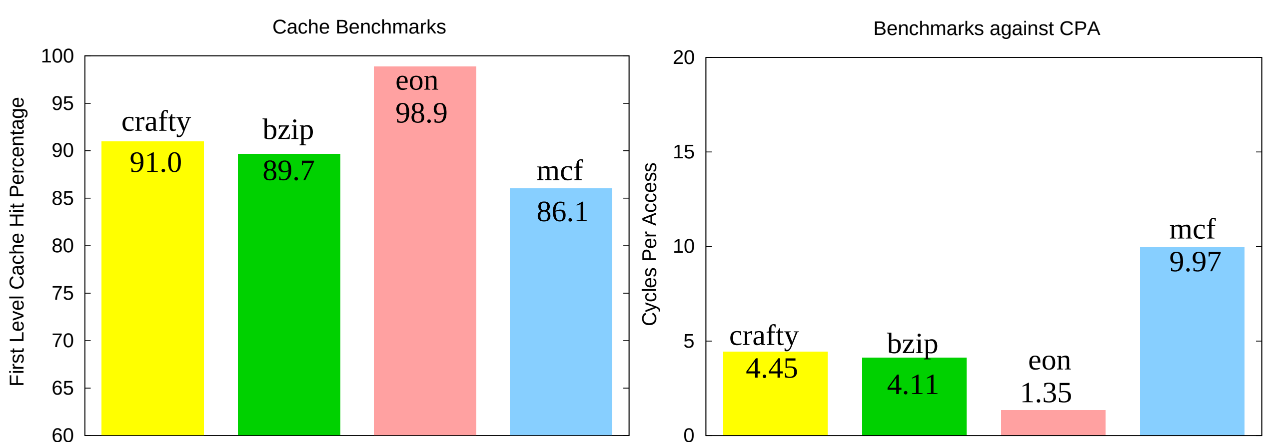


Figure 6.1: The first level hit rate and CPA of each of the benchmarks when simulated with no interruptions.

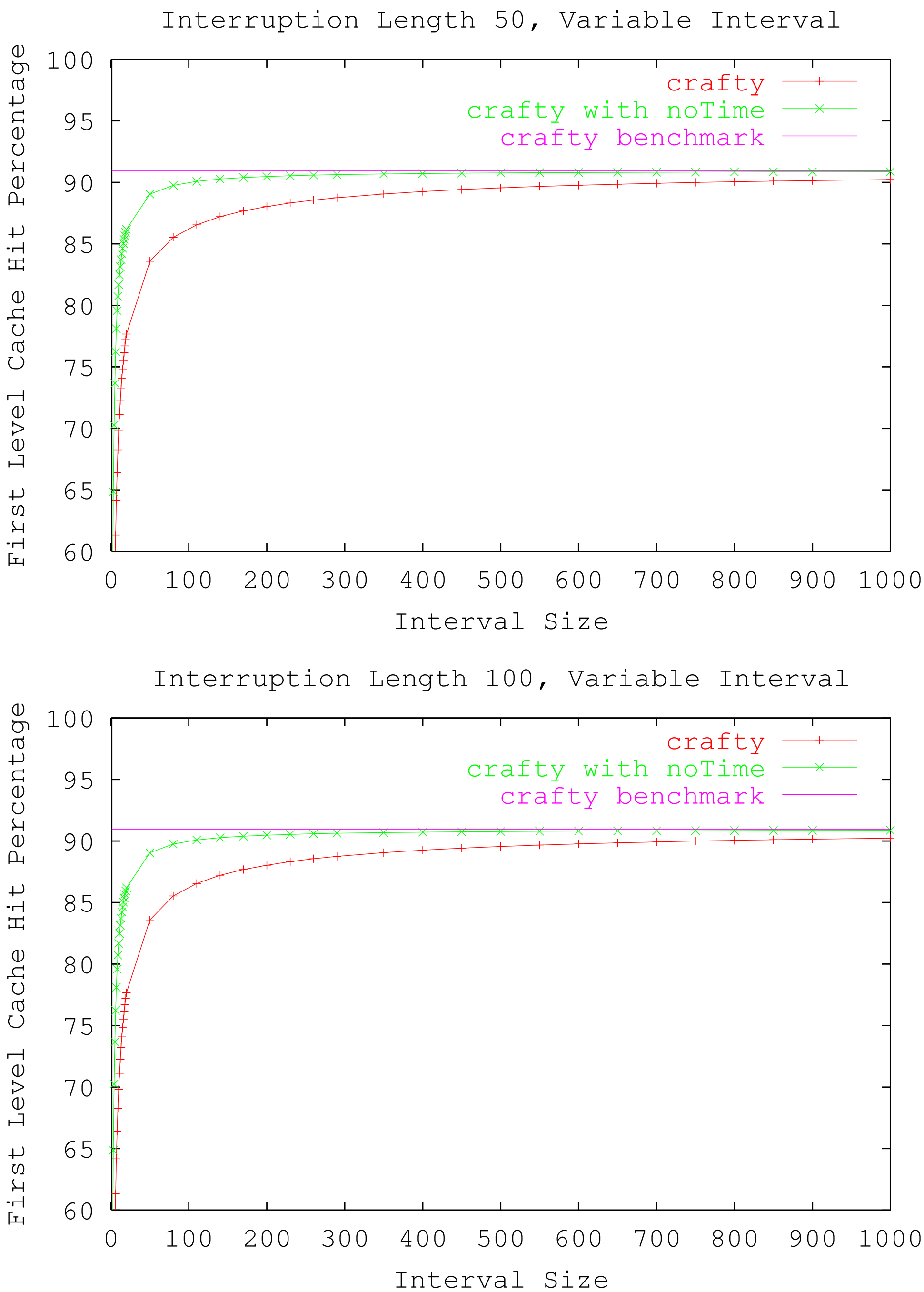
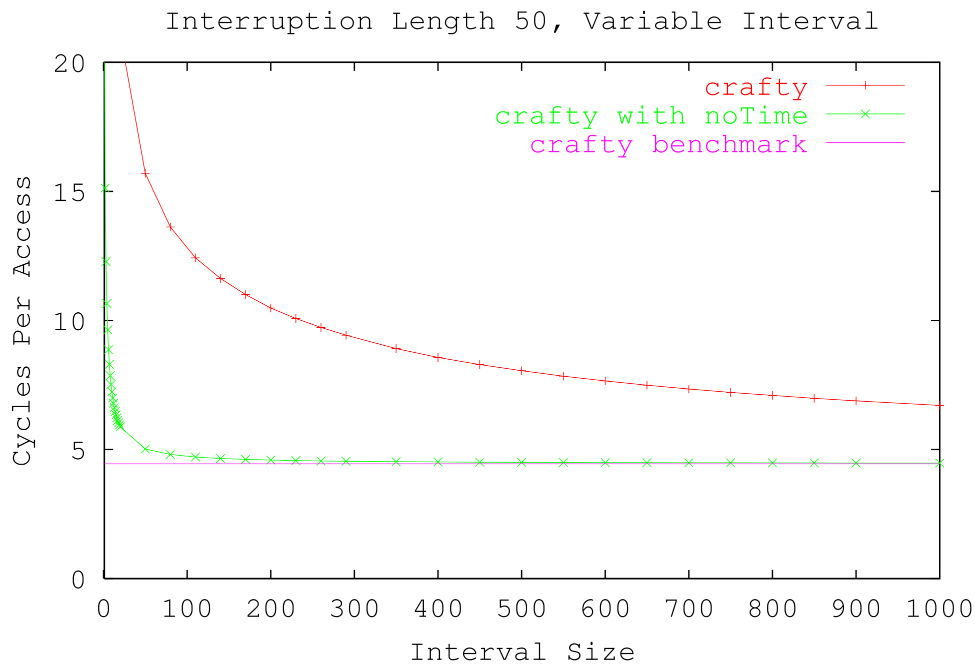


Figure 6.2: First level cache hit rate as interruption interval is varied and interruption length held constant at 50 and 100 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. The noTime line converges to the benchmark as the interruption interval increases. Note the lack of change in performance between interruption length 50 and 100.



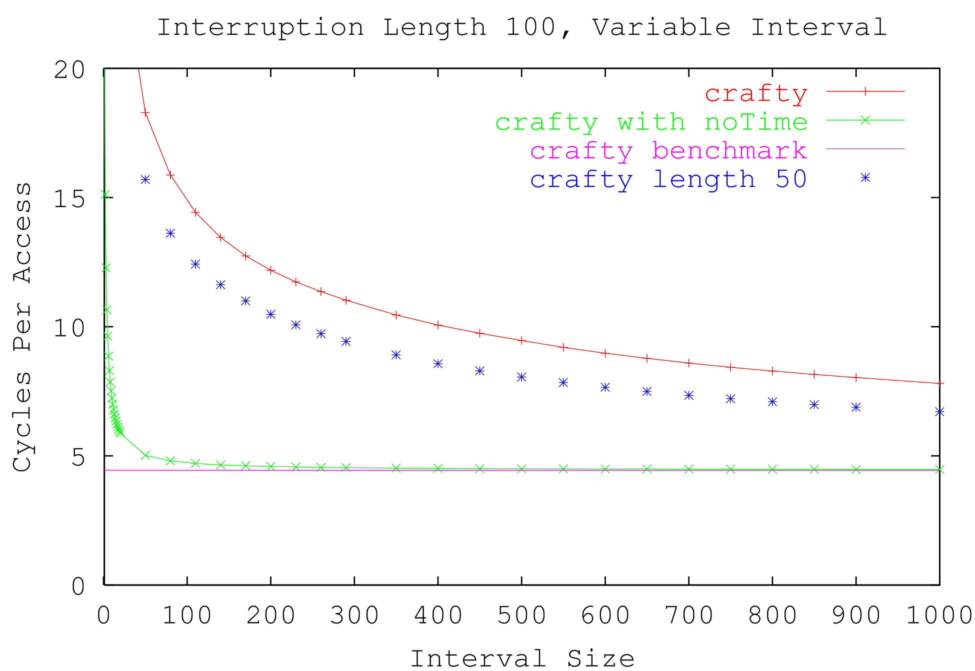


Figure 6.3: CPA as interruption interval is varied and interruption length held constant at 50 and 100 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. The noTime line converges on the benchmark again. Note the difference between the results of the tests with interruption length 50 and 100 without the use of noTime in this graph.

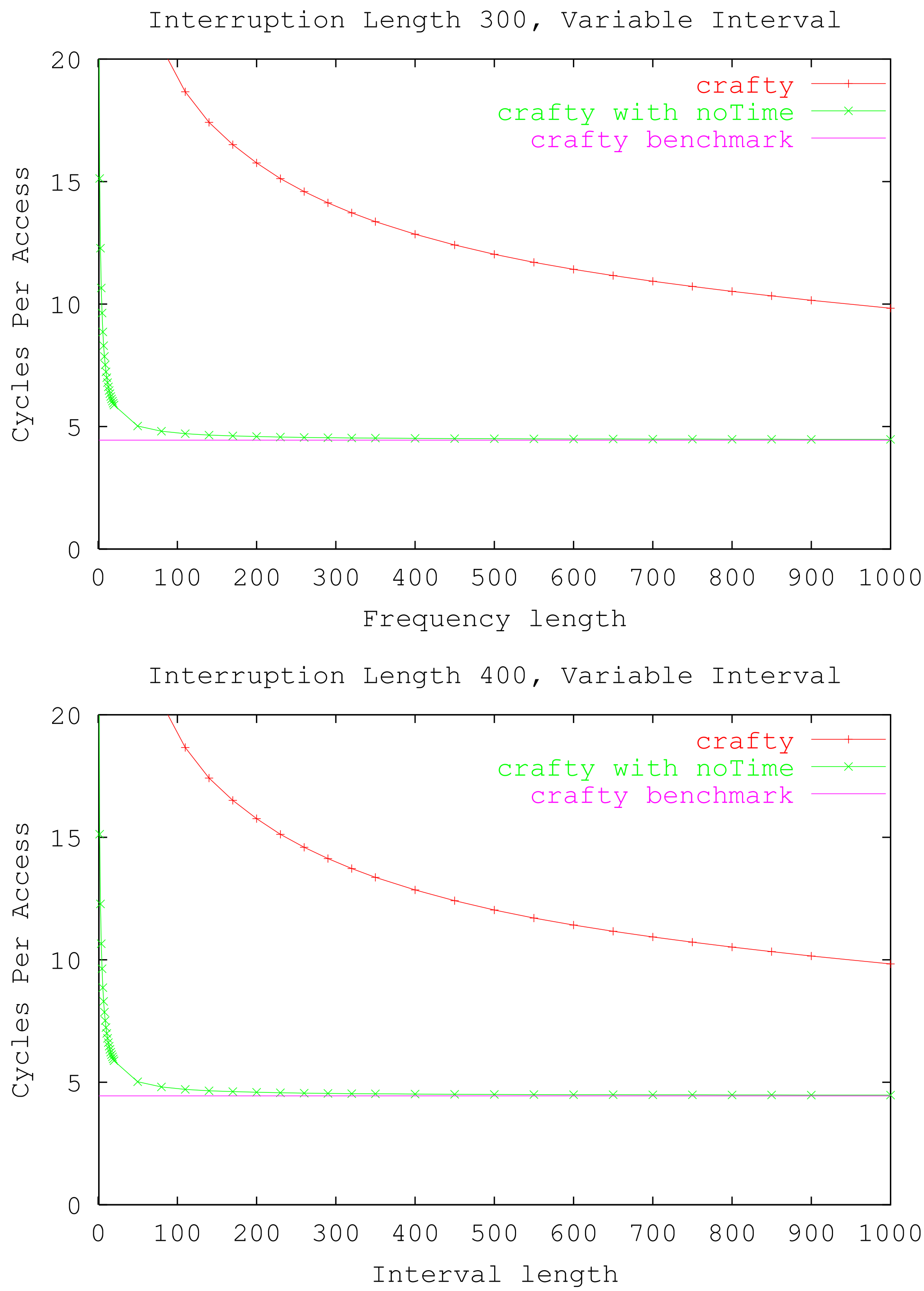


Figure 6.4: CPA in variable interruption interval test with increasing constant interruption lengths, 300 on the top and 400 on the bottom with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. Since both of these lengths are greater than the size of the second level cache, the increased length does not hurt performance.

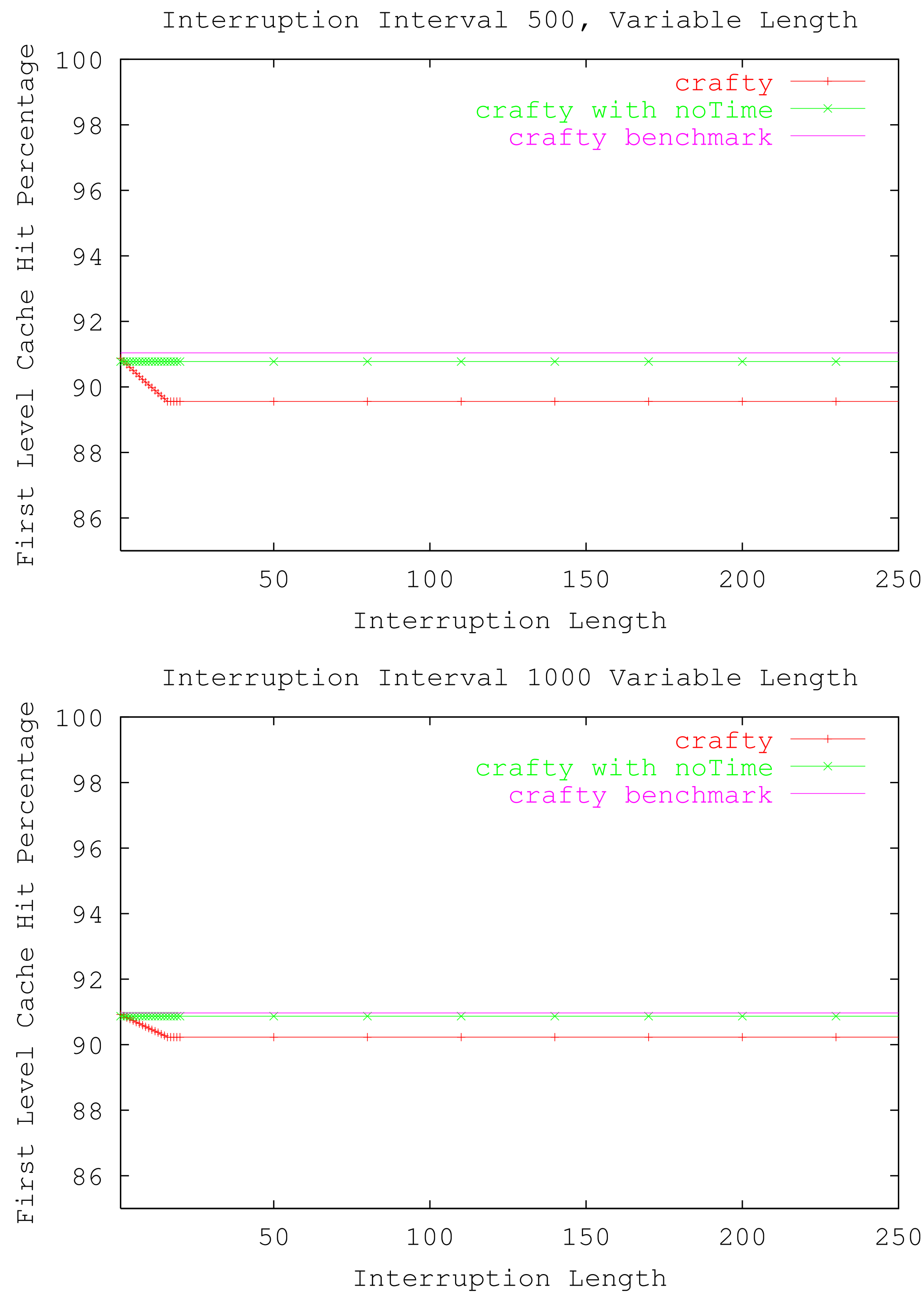


Figure 6.5: First level cache hit rate as interruption length is varied and interruption interval held constant at 500 and 1000 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. The noTime line is a constant distance below the benchmark regardless of the magnitude of the length at high lengths. Note the initial slope which ends when the first level cache has been overwritten.

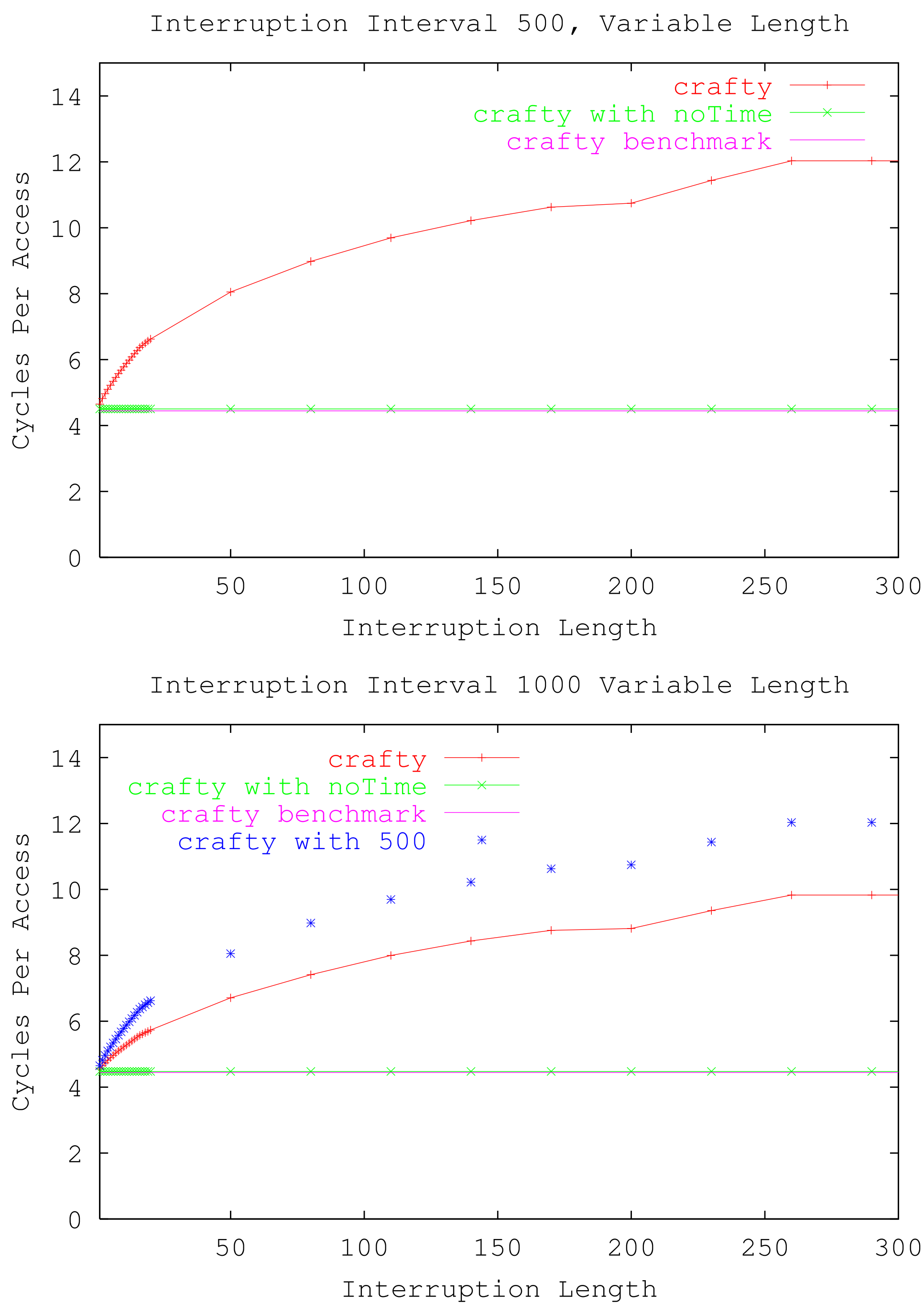


Figure 6.6: CPA as interruption length is varied and interruption interval held constant at 500 and 1000 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. The noTime line is overlapping the benchmark in this graph. Note the initial slope, which decreases when the first level cache has been overwritten, and flattens after the second level cache is empty. For selected datapoints, see Table 6.2

	20	50	80	110	200	350	500	800	3000
crafty 50	21.1	15.7	13.6	12.4	10.5	8.9	8.0	7.0	5.5
noTime 50	5.9	5.0	4.8	4.7	4.6	4.53	4.50	4.48	4.46
crafty 100	24.5	18.3	15.9	14.4	12.2	10.5	9.5	8.3	6.1
noTime 100	5.9	5.0	4.8	4.7	4.6	4.53	4.50	4.48	4.46

Table 6.1: Selected datapoints from Figure 6.3 which graphs CPA as interruption interval is varied and interruption length held constant at 50 and 100 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size.

	0	20	50	110	200	260	350
crafty 500	4.4	6.6	8.0	9.7	10.7	12.0	12.0
noTime 500	4.4	4.5	4.5	4.5	4.5	4.5	4.5
crafty 1000	4.4	5.7	6.7	8.0	8.8	9.8	9.8
noTime 1000	4.4	4.47	4.47	4.47	4.47	4.47	4.47

Table 6.2: Selected datapoints from Figure 6.6 which graphs CPA as interruption length is varied and interruption interval held constant at 500 and 1000 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size.

The performance of the first level cache on the variable interval crafty test with noTime was noTime's worst performance on this test against any of the traces, as shown in Figure 6.7. Even when performing poorly, the noTime simulation converged more quickly than the tests without noTime. On all tests, the noTime hardware greatly decreased the amount of cache pollution due to read-once memory accesses.

6.2 Additional Tests

The interruption interval and length tests allowed us to isolate the effects of these interruption variables on performance. In this section we explore the effects of other variables.

6.2.1 Variable Associativity

A variety of tests performed on variable first level and second level associativity showed that associativity had no great effect on the performance of on the benchmarks. Our associativity tests used a loose approximation of associativity, by charging no additional cost for increased cache associativity. Even with this relaxed model, the results marked the greater associativities as only nominally better than direct mapped caches on any of the benchmarks. The improved performance provided by the increased associativity in a real system would be offset by the larger hit cost of the increased associativity. Each test used a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k blocksize unless otherwise specified. The full results of this test are available in Appendix B

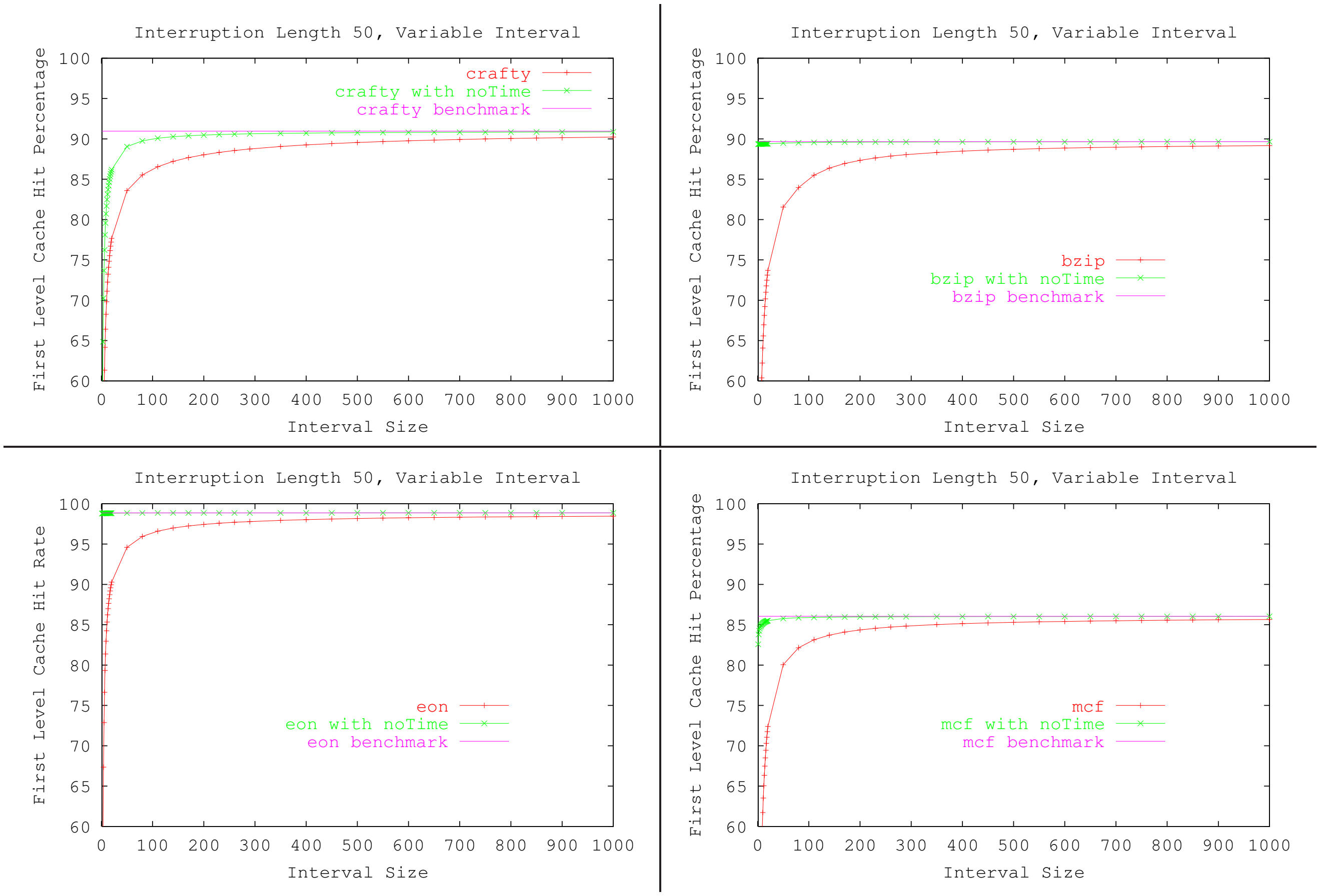
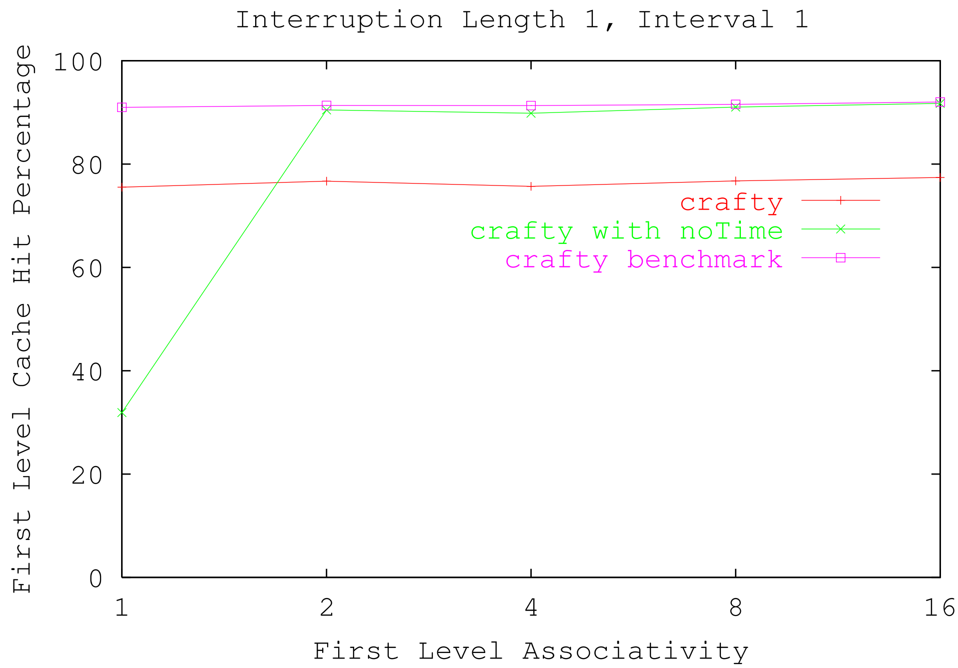


Figure 6.7: The first level cache hit rate on the benchmarks when simulated as interruption interval is varied and interruption length held constant at 50 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size. Note the range of improvement in the noTime performance. The graphs show crafty (upper left), bzip (upper right), eon (lower left) and mcf (lower right).

Direct mapped associativity led to a significant worst case, as shown in Figure 6.8. Here, frequently accessed cache blocks were mapped to the first block in the cache – the block used by noTime. 59.5% of all data mappings in this memory trace were made to the first cache line, as opposed to 3.9% in the mcf benchmark. This unlucky mapping weight also explains the relatively poor performance of crafty with noTime as compared to the performance of the other benchmarks. Thus, we greatly decreased the first level cache hit rate, but even so did not seriously affect performance. This sort of behavior was observed on only one of the four data sets.

6.2.2 Second Level Cache

Examination of the second level cache is desired at times, but because of its dependence on the first level cache, the results are not comparable to alternative test runs in which the second level cache did not receive the same memory requests. In some instances,



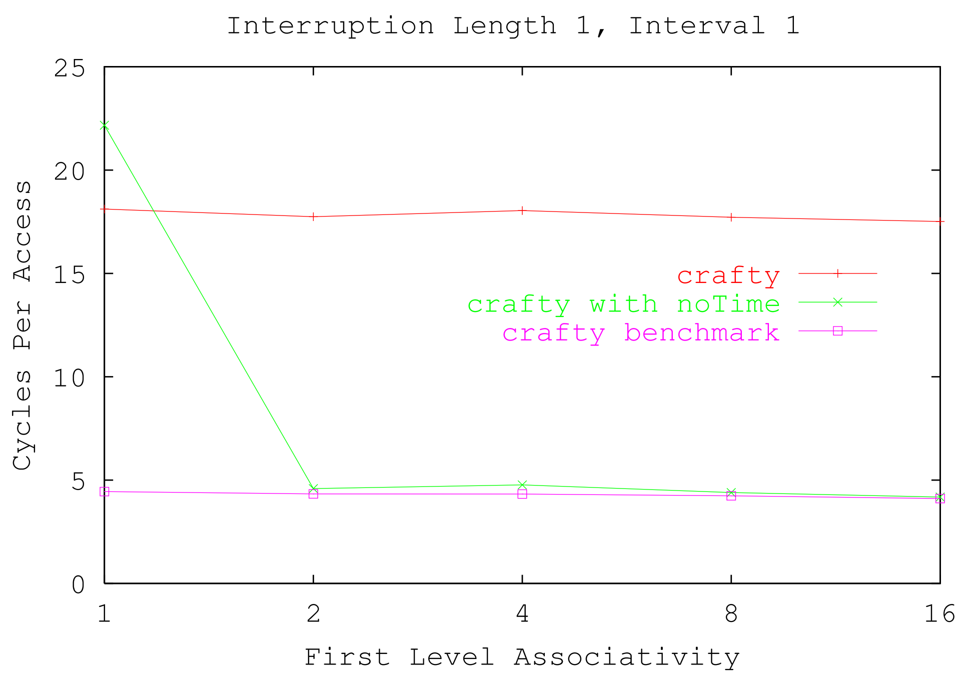


Figure 6.8: First level cache hit rate and CPA as first level associativity is varied on the crafty benchmark with the interruption length and interval held constant at 1. Note the performance hit in this case using noTime with a direct mapped cache due to mapping conflicts.

the second level cache may perform better than the first level cache at low interruption intervals. Suitable alternatives to analyzing the behavior of the second level cache include examination of the CPA and simulation without a first level cache – in essence, making the first level cache the size of the second level cache. We chose to use CPA to integrate all hit percentages.

6.3 Grounding of Results

We examined the effects of noTime on read-once memory accesses contained within the benchmarks. We used a conservative approximation to mark read-once accesses, then ran the simulation with these read-once accesses processed by noTime. Our results showed the expected small but significant improvement in CPA. Our conservative approximation consisted of recording which memory addresses from the trace were not referenced again within a certain span. Using three times the cache size as this limit gave us a trace result which consistently outperformed normal execution.

The read-once access density in these benchmarks varied from .1% to 1% of the total data accesses, while the improvement to performance ranged from .2% to 16%. In tests performed with memory traces from the target domain, we expect even greater improvement.

Chapter 7

Discussion and Future Work

7.1 Discussion and Interpretation

These results clearly show that room for improvement exists in intercepting read-once memory accesses. By avoiding the use of methods designed to take advantage of temporal locality in our cache lookup, we can prevent the read-once memory accesses from degrading later performance.

We also observe that the effect of interruptions on the cache is dependent on the cache size. The modular nature of the default mapping guarantees that a cache of size x will be flushed after x consecutive interruption blocks.

Thus the effects on performance are tied similarly to low intervals and high lengths, although the cache performance effects flatten at high lengths. Using noTime solves the performance degradation problem observed by both low intervals and high lengths, providing performance similar to that of program execution without interruptions.

We also grounded our results in real cache traces. Although these traces were not read-once heavy, they demonstrate that noTime will be beneficial at some level for programs with any level of read-once access present.

The noTime hardware requires perfect knowledge of the memory trace in advance to activate at the optimal intervals, but through simulation with noTime, we can establish an upper bound to the performance improvement possible through noTime. For many applications, the read-once memory accesses can be explicitly tagged. In these applications, performance with noTime without perfect information can converge on the ideal performance. In all other programs, any incomplete application of noTime with conservative partial activation will provide some improvement.

7.2 Future Work

We have shown that read-once memory accesses cause an unnecessary performance slowdown and that our theoretical hardware improvement, noTime, can be used to improve performance in these systems. This paper is only the first step in investigating noTime and other means of specializing the cache to take advantage of read-once memory access patterns. Some future work is described below.

7.2.1 Intelligent Heuristics

Use of intelligent heuristics to evaluate memory accesses actively and predict read-once memory access occurrence is one method of closing the gap between our current level of knowledge and the perfect knowledge used in our tests. These heuristics could be applied at compile time to make predictions about memory access status or memory requests, or at runtime to make predictions during execution.

At compile time, the heuristics could examine the predicted patterns of the entire program to get a larger picture of the scope of the execution, and use this knowledge to make predictions. This larger picture could also allow a number of compiler optimizations to consolidate read-once accesses for greater efficiency with noTime. Developing this sort of heuristic would be difficult but has large potential returns.

Alternatively, heuristics could be applied at run-time to analyze memory patterns as the programs executes. This sort of analysis could detect repeated cache working set access patterns and then activate noTime accordingly. These two methods could even be combined into an executable editor system which would mark up the executable after initial execution with read-once labels. Using such a system, a user could execute dry runs of a program to establish correct labeling after compilation.

7.2.2 Read-Once Labeling

In order to apply these compile time heuristics and manual overrides, we must have some means of labeling memory accesses as read-once for the machine to interpret at run-time.

At the compiler level, we can attempt to store some of this data to allow us to execute with increased knowledge. With a labeling implementation, the user can also explicitly label memory access before compile time. We discuss two means of implementing labels below.

High Order Bit Labeling

We propose examination of the values in the uppermost bits of memory as a viable alternative to perfect knowledge. These bits are not used to index line or block offset in the cache, and thus, with compiler support, could be used to label read-once or non-read-once memory types (Figure 7.1).

This approach is limited in that each memory address may only either be read-once or read-write. This is determined at time of allocation. However, this labeling method could be implemented without great difficulty. The modifications necessary would require that memory be partitioned into a read-once half and a read-write half. All memory allocated in the read-once half would be treated as read-once memory accesses whenever referenced. Extensions on this policy could require two or more active bits in the unused portion to activate noTime, thus creating a smaller read-once partition, but this approach retains the limitation that a read-once address can only be accessed in a read-once manner.

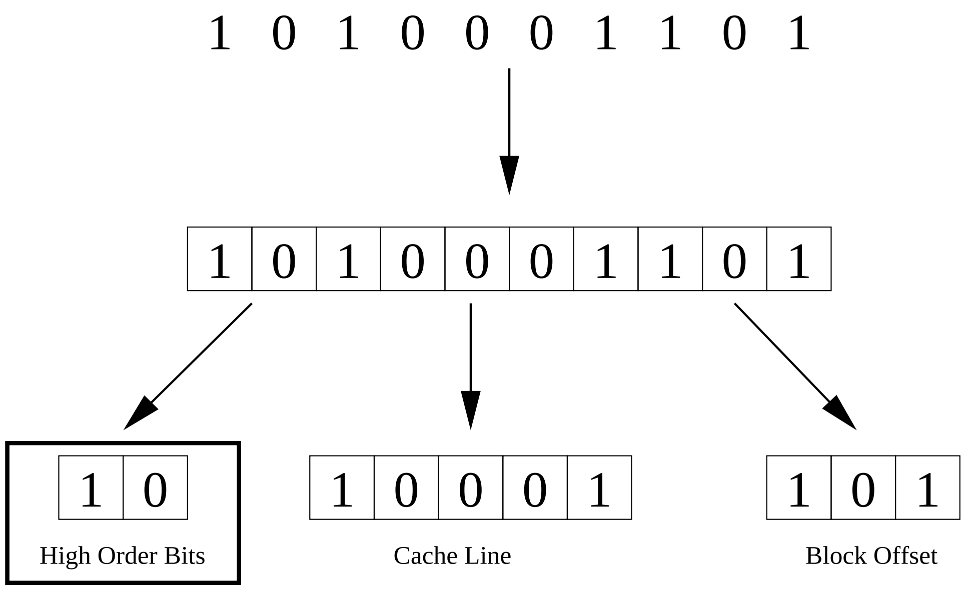


Figure 7.1: A simplified model of the separation of a memory address into cache line, block offset and unused order bits. One of the high order bits could be used as an activation bit for noTime, although this would partition all memory into read-once memory and standard memory.

Explicit Labeling

We could implement explicit labeling by adding an address mode bit to every data access to signal read-once status. This approach would increase system overhead, as well as requiring more work from the compiler, but would allow us to store the level of knowledge required for noTime.

We could combine this system with the high order bit labeling by using a series of high order bits to activate noTime, by allowing the addresses referenced when noTime was active to be also mapped to another address. This method would permit both read-once and non-read-once access to a subset of the address space, but adds complexity to the system.

7.2.3 Further Examination of Trace Files

We have investigated only a small amount of the information contained within the trace files. Described below is further work to be investigated in relation to these traces.

7.2.4 Generation of Traces

As discussed in Section 5.1.4, we used previously recorded trace files from the SPEC2000 integer benchmarks, and inserted simulated read-once memory accesses. Use of cache traces from programs performing read-once memory accesses would provide more accurate results, and could further support conclusions reached about noTime. Examination of such trace files could confirm the accuracy of read-once memory access simulation using

	1	2	4	8
Crafty	.18	.31	.28	.87
Bzip	.82	.29	.04	.28
Mcf	.54	.45	.58	.55
Eon	.81	.19	.80	.96

Table 7.1: Table comparing average value for each bit in the mapping of a 16k direct mapped cache with a 1k blocks over all benchmarks. These bits are bits 11 through 14 of the full address.

regular interrupts according to our method, and enable research into intelligent heuristics for use with read-once memory access.

7.2.5 NoTime Block Choice

We arbitrarily chose to use the first block in the cache as the block used by noTime. This block may be accesses much more heavily than other blocks in some instances (notably the crafty benchmark). Investigation into the ideal noTime cache block choice could provide interesting results. Work could also be done comparing the results using noTime with a variety of cache block choices against the results using a separate one-cache-line buffer.

Early work examining the block allocation patterns in the benchmarks have shown no conclusive trends in block mapping. Investigation into the generalizations we can draw from the behavior of the individual benchmarks could provide interesting results. Our preliminary results in a bitwise analysis of the cache blocks referenced are displayed in Table 7.1

7.2.6 Shared Memory

The field of shared memory cache coherency protocols is another area where nonstandard access patterns may occur. The access patterns of these protocols may share some of the characteristics of read-once memory accesses. Generating address traces of programs with these access patterns could be an interesting avenue of investigation.

Any message passing program making use of these shared memory cache coherency protocols may be suffering due to other types of nonstandard cache access patterns. Writes to cache blocks meant as messages should always be released from exclusive access immediately. We have not seen whether this behavior occurs. This optimization and other work in applying the methology used to develop noTime to cache coherency protocol may lead to substantial improvement.

7.2.7 CRASS

CRASS provides a good model with which to simulate read-once memory accesses at the cache level. We have not yet explored all possibilities it provides.

Extension of CRASS

With a modest amount of effort, CRASS could be extended to exclude cold starts and include code segments. By excluding cold starts, we would simply not begin recording cache misses until after a certain segment of the program had been executed. To include simulation of code segments, we would add another frequency variable, giving CRASS both a long frequency L and a short frequency l. Using these values, CRASS would perform normal computation for L accesses, then perform several stages of interrupts separated by l normal accesses, then return to L normal accesses.

Further Application of CRASS

Our cache simulator, CRASS, is a versatile simulation program. It can be adapted with little difficulty to other types of problems, and modularly expanded to solve problems which are currently beyond its grasp. In investigating read-once memory accesses, we found a similarity in our simulations to the cache behavior surrounding context switches. With minimal adaptation, CRASS could model the effect these context switches have on cache performance.

Read-Once Labeling with CRASS

We developed an approximation tool for use with CRASS which gives a conservative approximation of read-once memory access labeling on a trace file. Using CRASS, we could develop a two pass cache simulator which uses the first pass to determine which data accesses are read-once. This method would be an improvement on our conservative approximations, but would take substantial development. We did not implement these features because the sparsity of read-once accesses on the address traces we worked with did not require the level of precision this method could offer. With more address traces however, this approach could establish the upper bound on performance and provide insight into the development of intelligent heuristics for read-once prediction.

Bibliography

[1] MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall/Winter 1994. 18

[2] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Stanford University, 1987. 9, 13, 14

[3] M. H. Anant Agarwal, Richard Sites. ATUM: A new technique for capturing address traces using microcode. *IEEE*, 1986. 24

[4] Brigham Young University Performance Evaluation Team Trace Distribution Center, http://tds.cs.byu.edu/tds/index.jsp, BYU. URL, 2003. 24, 26

[5] Brigham Young University Performance Evaluation Team Trace Distribution Center, http://traces.byu.edu/new/documentation/address/technique.html, BYU. URL, 2003. 24

[6] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998. 18

[7] CHIMP concepts. University of Edinburgh, June 1991. 18

[8] CHIMP version 1.0 interface. University of Edinburgh, May 1992. 18

[9] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002. 15

[10] J. E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Load balancing and communication optimization for parallel adaptive finite element computation. In *Proc. XVII Int. Conf. Chilean Comp. Sci. Soc.*, pages 246–255, Los Alamitos, CA, 1997. IEEE. 15, 17

[11] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC benchmark suite. Technical Report CS-TR-1991-1049, 1991. 7

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22:798–828, 1996. 18

[13] W. Gropp and B. Smith. *Users Manual for the Chameleon Parallel Programming Tools*. Argonne National Laboratory, anl-93/23 edition, June 1993. 18

[14] M. D. Hill. A case for direct mapped caches. In *IEEE Computer*. IEEE Computer Science Press, 1988. 11

[15] J. Larus. WARTS, http://www.cs.wisc.edu/ larus/warts.html. URL, 1996. 23

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995. 23

[17] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, San Francisco, California, 1996. 4, 7, 9, 10, 12, 13, 14, 27

[18] K. Petersen and K. Li. An evaluation of multiprocessor cache coherence based on virtual memory support. In *Proc. of the 8th Int'l Parallel Processing Symp. (IPPS'94)*, pages 158–164, 1994. 14

[19] P. Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube*, pages 384–390. ACM Press, 1988. 18

[20] S. A. Pryzybylski. *Cache Design: A Performance-Directed Approach*. Morgan Kaufman, San Mateo, 1990. 9

[21] J.-F. Remacle, O. Klaas, J. E. Flahery, and M. S. Shephard. Parallel algorithm oriented mesh database. *Eng. Comput.*, 18(3):274–284, 2002. 15

[22] CINT2000 (Integer Component of SPEC CPU2000), http://www.spec.org/osg/cpu2000/cint2000/index.html, SPEC CINT2000 Benchmarks. URL, 2003. 24

[23] SPEC 2000 Bzip Benchmark, http://www.specbench.org/osg/cpu2000/cint2000/256.bzip2/docs/256.bzip2.html. URL, 2003. 25

[24] SPEC 2000 Crafty Benchmark, http://www.specbench.org/osg/cpu2000/cint2000/186.crafty/docs/186.crafty.html. URL, 2003. 24

[25] SPEC 2000 Eon Benchmark, http://www.specbench.org/osg/cpu2000/cint2000/252.eon/docs/252.eon.html. URL, 2003. 25

[26] SPEC 2000 MCF Benchmark, http://www.specbench.org/osg/cpu2000/cint2000/181.mcf/docs/181.mcf.html. URL, 2003. 25

Appendix A

Caching for Read-once Accesses Simulation System

A.1 CRASS Overview

CRASS is a system designed to simulate data cache operations on a variety of basic cache configuration. In addition, CRASS allows the programming of read-once interrupts of specified length and frequency to occur while marching through the trace file.

CRASS takes input in the form of a binary file file as specified by BYU in Appendix D. The simulations has only been tested on Solaris machine using g++ 3.1. One must be careful of endian problems when inputting trace files on a Sun computer, as the publically available trace files are of the opposite endian of the machine.

The command line flags are described below. All command line options except -x can also be specified in a configuration file by placing the flag letter and its argument (not always necessary) on a line. Options loaded from the configuration file can also be overruled by adding configuration options after -x. In a similar manner, multiple configuration files can be read, but a configuration file cannot be read from within a configuration file.

In addition, the Perl program packer.pl takes a base filename x as an argument, and converts x.josh to x.tr. .josh files have a single reference address on each line.

A.2 Command Line Arguments

The command line arguments are described below.

- *h* – this (help) message.
- *x file* – Use command line options as read from file.
- *v* – Verbose output.
- *b size* – Cache block size (default 1k or 1024 bytes).

- *c/C size* – Cache size for the first and second level cache (default 16k and 256k). Size should be a power of two.
- *r/R policy* – Replacement policy for the first and second level cache (only relevant if associativity is set above 1). f for fifo (default) and l for lru.
- *w* – Use new hardware policy where interruptions are always mapped to block 0.
- *a/A num* – First and second level cache associativity (default 1). The associativity should be a power of two less than or equal to the cache size.
- *l length* – Interruption length in blocks.
- *d* – toggles whether the tr file is a foreign endian (default true).
- *o policy* – Reset policy for interrupt address. (default c: contiguous advancement, also a: reset advancement and r: random).
- *e num* – Seed for random number generator. Used for repeatability.
- *f file* – Binary trace file to use.
- *M size* – Maximum number of data reference to process before exiting.
- *F interval* – Interval between interruptions.

Appendix B

Further Results

This section contains a graphical representation of some of the results we referenced in Section 6. These results are in support of the conclusions we presented earlier. Variable interval tests are presented first, followed by variable length tests and first and second level associativity tests, organized by benchmark. All tests use a 16k direct mapped first level cache, a 256k 2-way set associative fifo cache and a 1k block size unless otherwise stated.

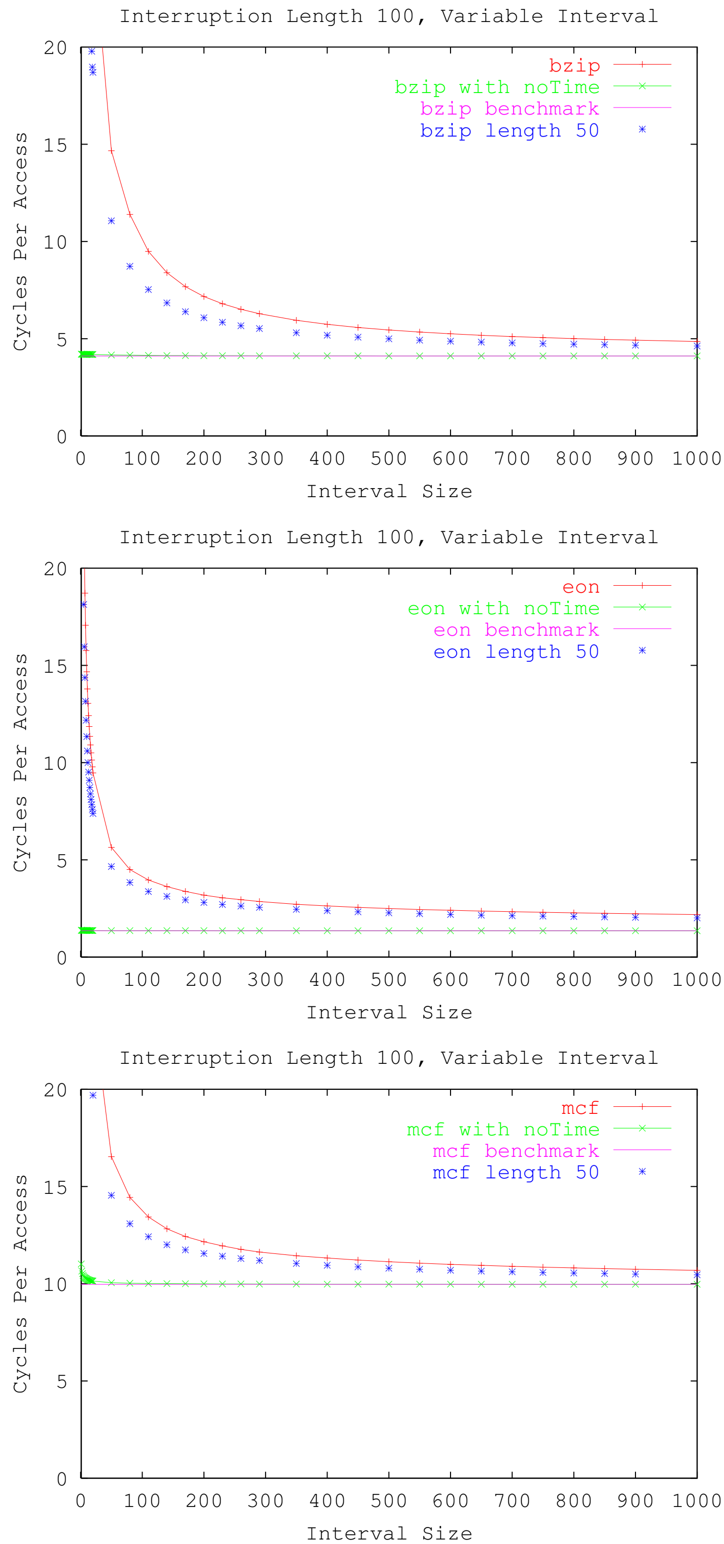


Figure B.1: Variable interval test on bzip, eon, and mcf with interruption length 100 and comparison to interruption length 50 with a 16k direct mapped first level cache, 256k 2-way set associative second level cache and a 1k block size.

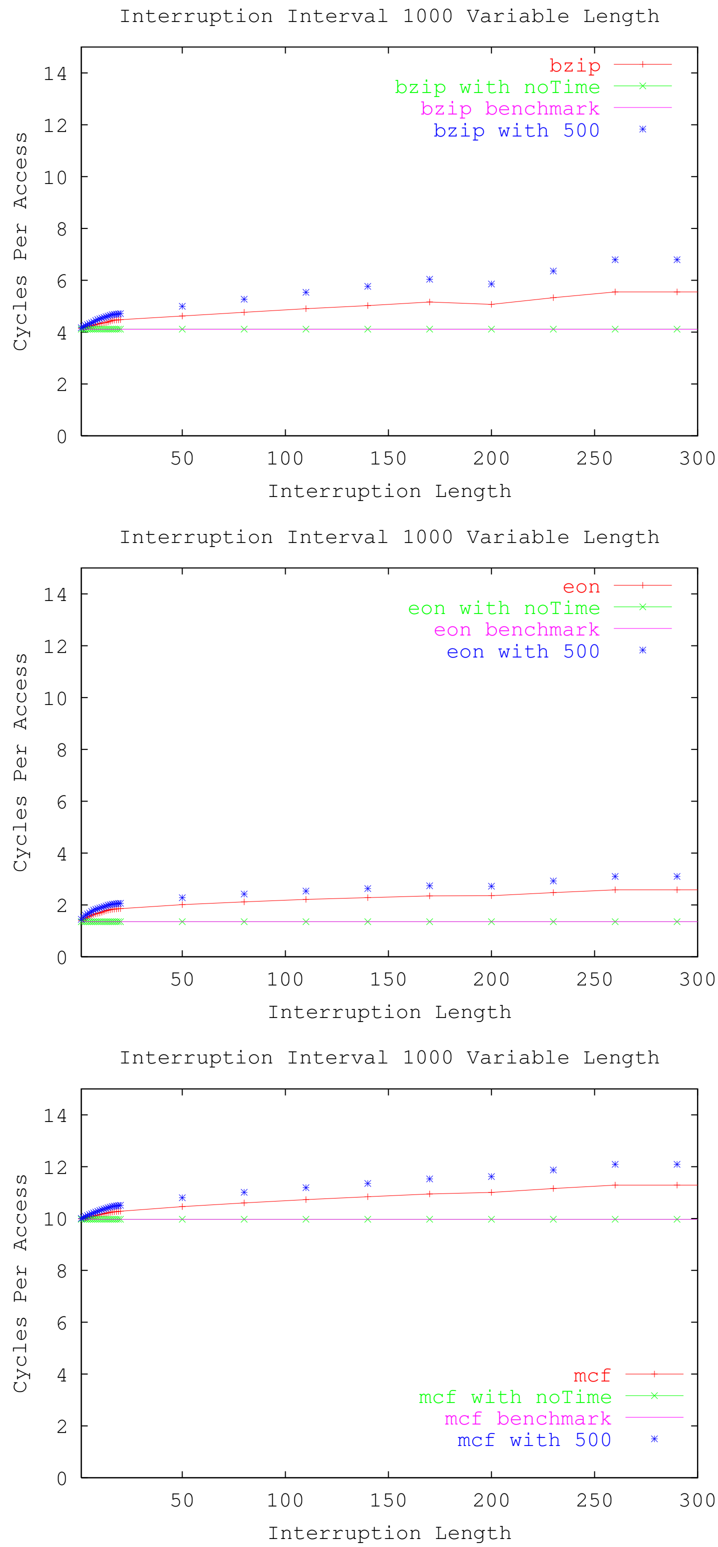


Figure B.2: Variable length test on bzip, eon, and mcf with interruption interval 1000 and comparison to interruption interval 500 with a 16k direct mapped first level cache, a 256k 2-way set associative second level cache and a 1k block size.

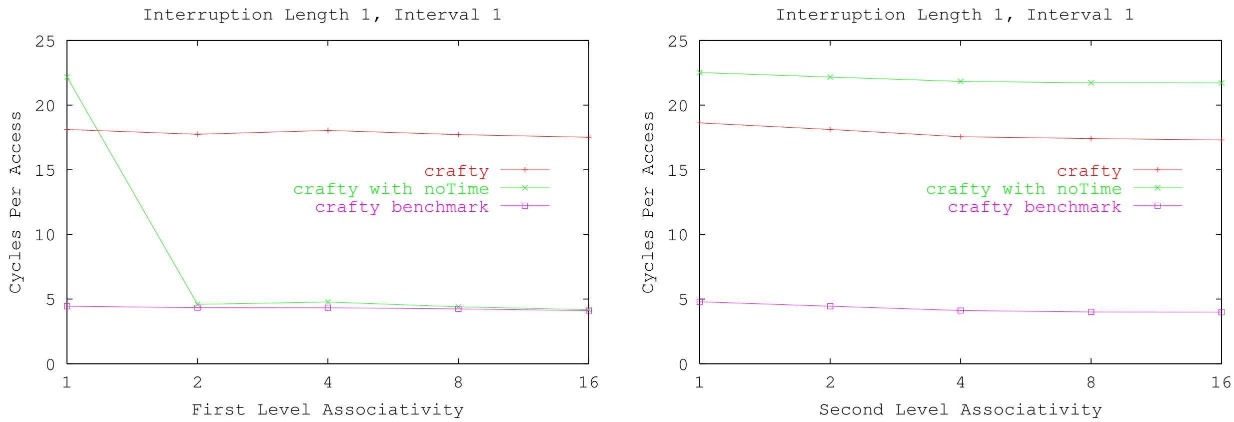


Figure B.3: Crafty variable first and second level associativity tests with interruption length 1, and interval 1, with no cost for increased associativity.

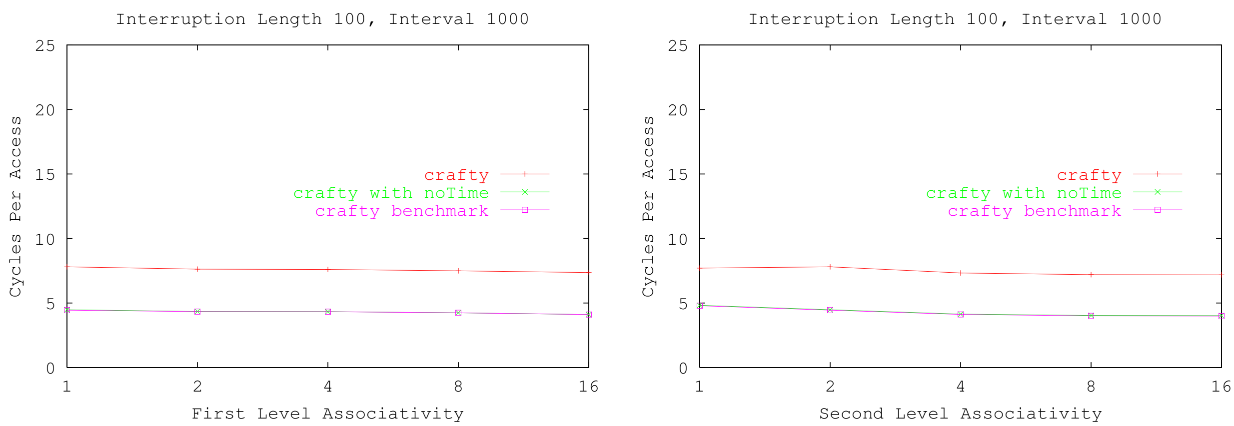


Figure B.4: Crafty variable first and second level associativity tests with interruption length 100, and interval 1000, with no cost for increased associativity.

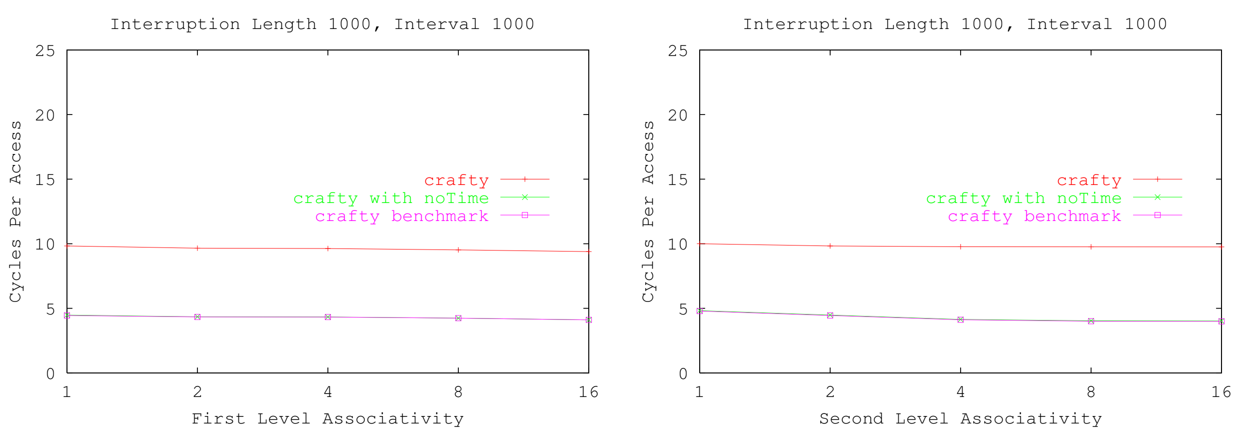


Figure B.5: Crafty variable first and second level associativity tests with interruption length 1000, and interval 1000, with no cost for increased associativity.

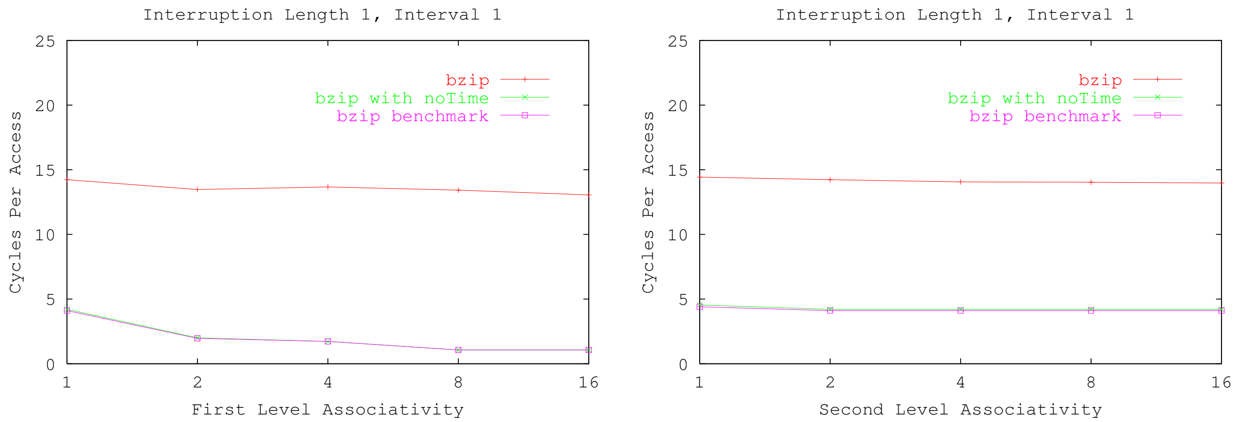


Figure B.6: Bzip variable first and second level associativity tests with interruption length 1, and interval 1, with no cost for increased associativity.

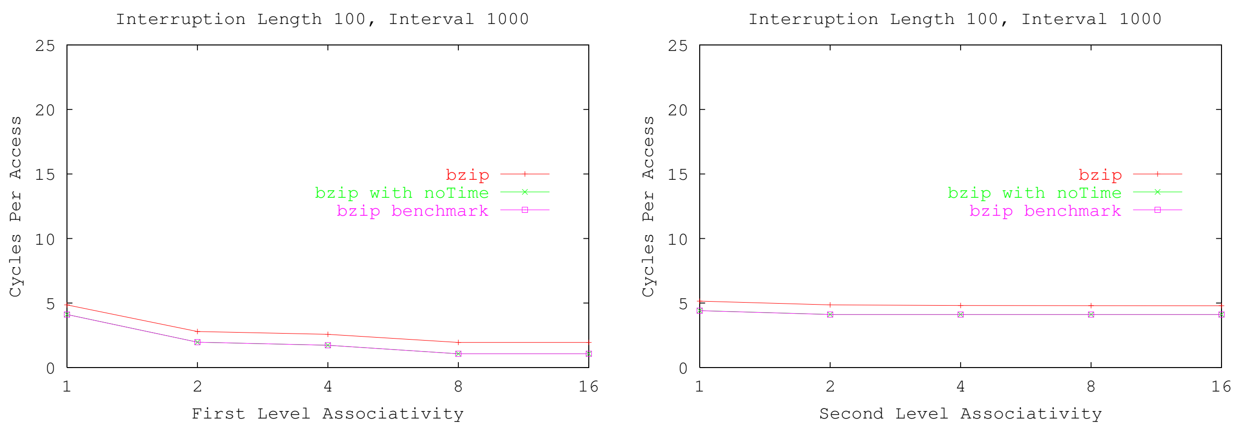


Figure B.7: Bzip variable first and second level associativity tests with interruption length 100, and interval 1000, with no cost for increased associativity.

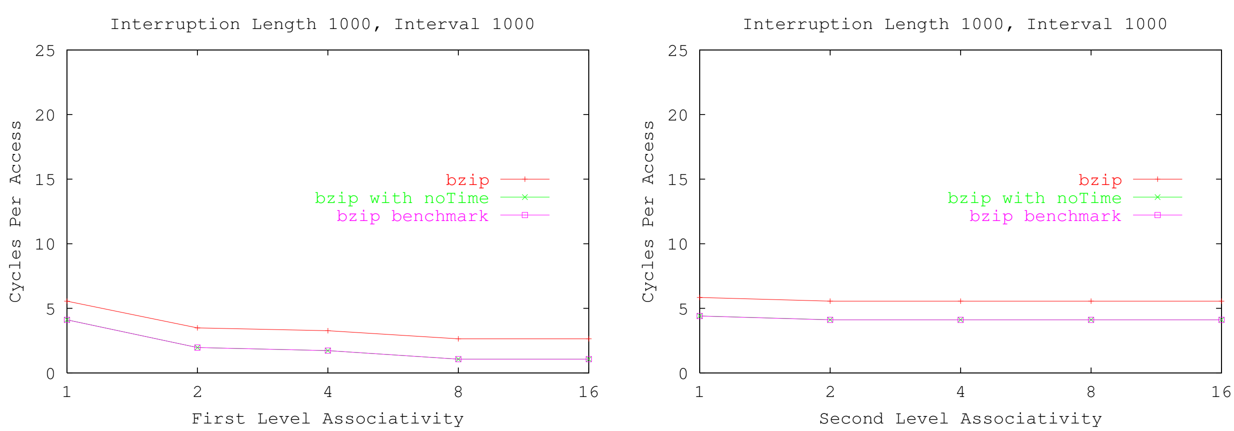


Figure B.8: Bzip variable first and second level associativity tests with interruption length 1000, and interval 1000, with no cost for increased associativity.

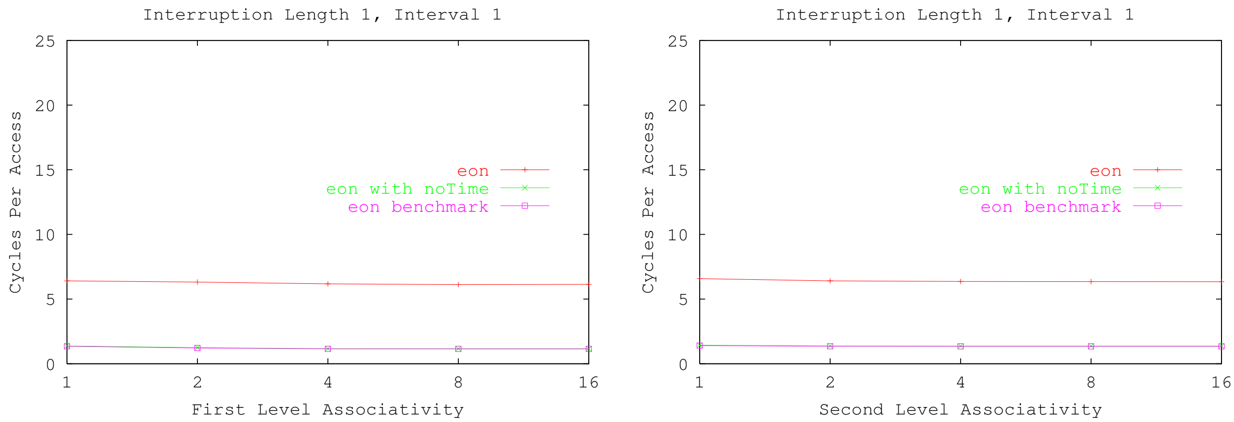


Figure B.9: Eon variable first and second level associativity tests with interruption length 1, and interval 1, with no cost for increased associativity.

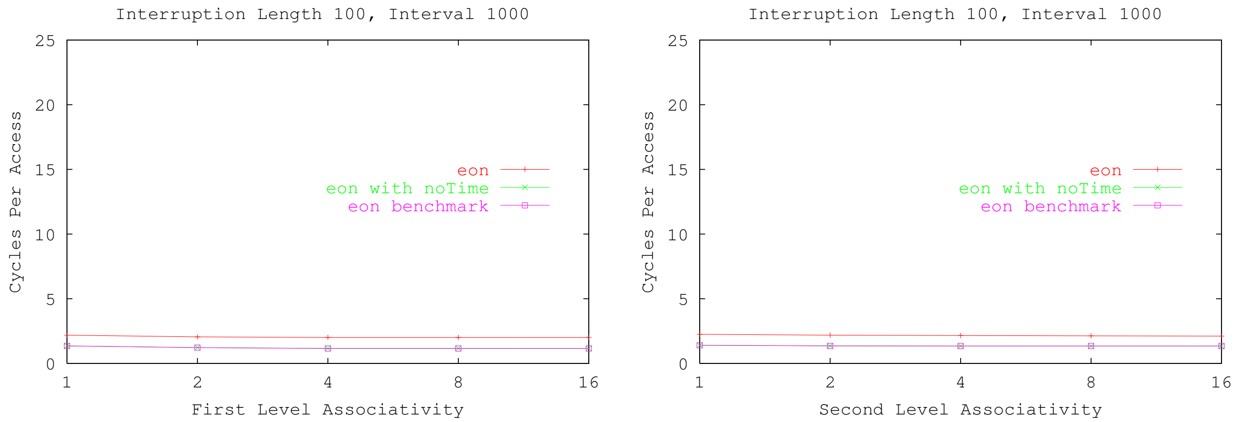


Figure B.10: Eon variable first and second level associativity tests with interruption length 100, and interval 1000, with no cost for increased associativity.

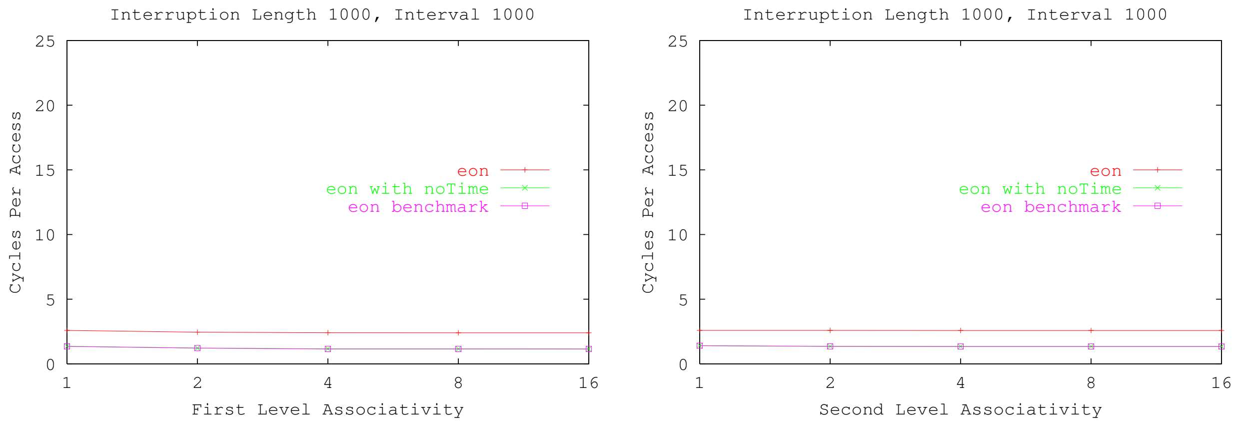


Figure B.11: Eon variable first and second level associativity tests with interruption length 1000, and interval 1000, with no cost for increased associativity.

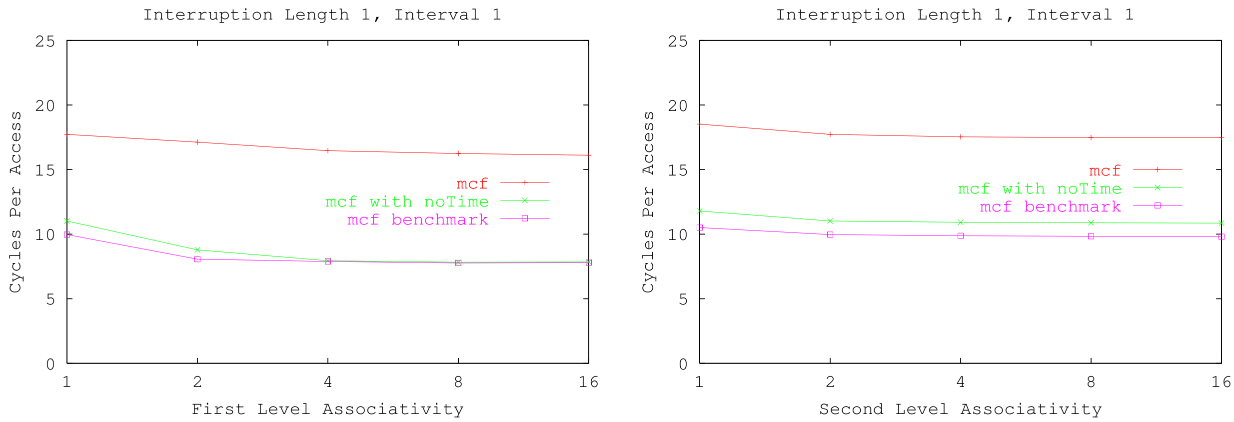


Figure B.12: Mcf variable first and second level associativity tests with interruption length 1, and interval 1, with no cost for increased associativity.

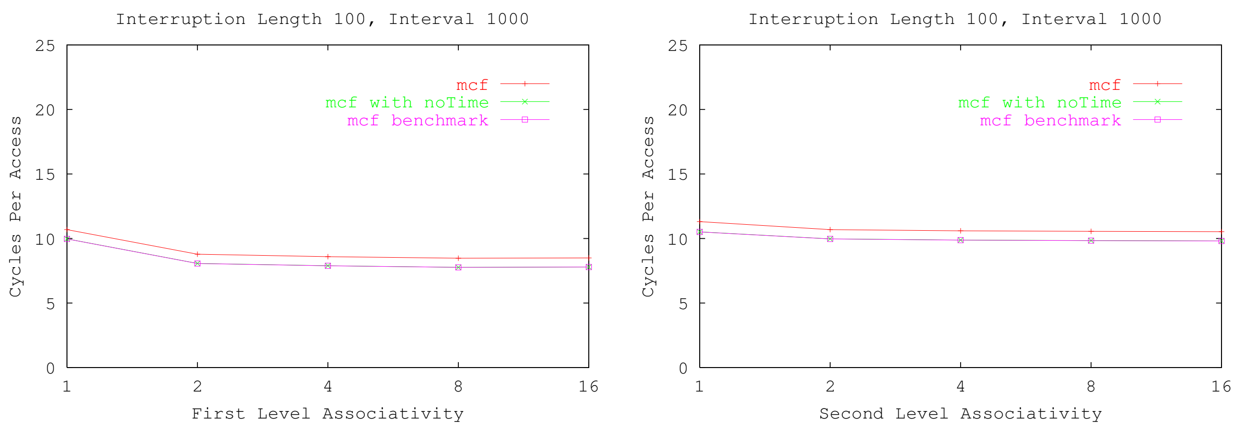


Figure B.13: Mcf variable first and second level associativity tests with interruption length 100, and interval 1000, with no cost for increased associativity.

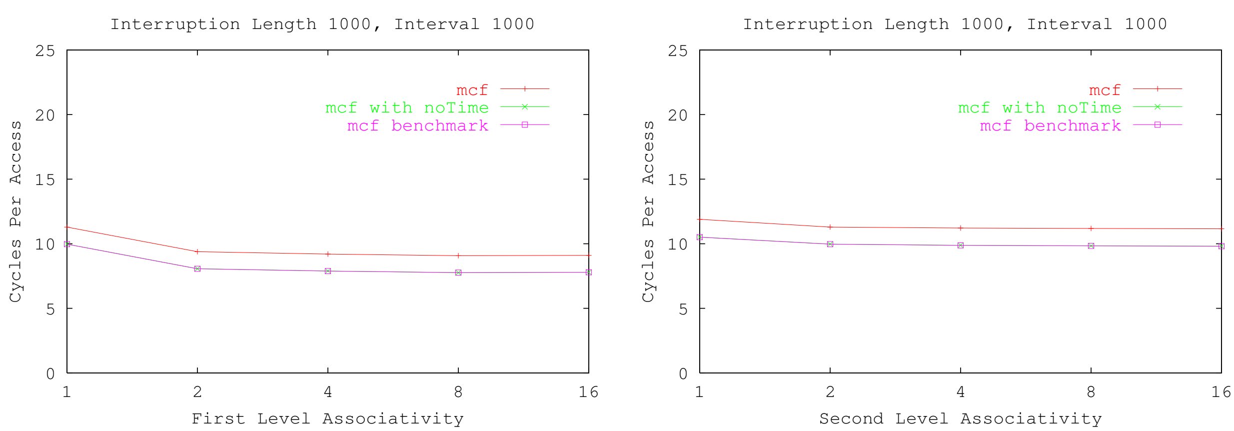


Figure B.14: Mcf variable first and second level associativity tests with interruption length 1000, and interval 1000, with no cost for increased associativity.

Appendix C

Read-Once Labeling

C.1 Read-Once Label Approximation

We developed a Perl based application to approximate read-once memory accesses. This application takes an integer x as a parameter. The application checks to see if each memory reference is accessed again within the next x accesses or the previous x accesses. If the memory reference is not, then it is labeled read-once.

Through trial and error, we found a conservative x of three times the cache size was sufficient for a reliable result. While a specific solution to this labeling problem is possible with a two-pass cache simulator, this approximation was deemed sufficient for use with our memory traces.

Appendix D

BYU Trace Documentation

The following sections are reproduced from the BYU trace archive website. They are protected under the following provision:

D.1 Address Trace Collection Technique

Technique for Obtaining Traces

We have a number of Tektronix TLA720 logic analyzers gathering data from the pins of Intel Pentium, Pentium II, and Xeon processors. The software we want to trace runs on the processor, under any of several different operating systems. When the buffer on the data analyzer is full, it sends a signal over to the Pentium's parallel port, triggering an interrupt. A special driver written for it sends it into a tight loop until the data analyzer is ready for more data. Meanwhile, the data analyzer sends the contents of its buffer over the network to a workstation that saves it into our tape array drive. A program on the workstation then goes through the raw data, saving out only the desired bits in a specific format, and then compresses it back to disk. This program then signals the data analyzer that it is ready for more data, causing the data analyzer to unlock the processor.

Currently, we are taking address traces from the chip, meaning that we record memory requests that the chip generates. This is useful for research involving caches.

D.2 Address Trace Format

Address Trace Format

- **BYU Address Trace Format v1.1**
  A bug in the size field was corrected. It now shows lengths of 16 and 32 bytes (not just 8 and smaller). Length 8 still accounts for over 90% of the references.

- **BYU Address Trace Format v1.0**
  Each trace record contains twelve (12) bytes of information in the following format:

```
typedef struct BYUADDRESSTRACE
{
        unsigned long addr;
        unsigned char reqtype;
        unsigned char size;
        unsigned char attr;
        unsigned char proc;
        unsigned long time;
} p2AddrTr;
```

- **Address Field**

  The address field of the trace is a 32 bit physical address. This field is in little endian format and represents the address generated by the processor being traced.

- **Request Type Field**

  The request type field is one byte long. It describes the type of request: instruction fetch, memory read or write, etc. For a complete listing, see byutr.h.

- **Size Field**

  The size or length field contains the size of the data transfer in bytes.

- **Attribute Field**

  The attribute field is one byte long and refers to the cacheability of the block. In complete address traces all instruction fetches and data reads are noncacheable because the first-level instruction and data caches are disabled. In trace data that is acquired with the first-level caches enabled only those references that are truly noncacheable are marked as such. The lowest two bits have the following meaning:

00	uncacheable
01	write through
10	write protect
11	write back

- **Processor Field**

  The processor or agent id field tells which processor submitted the request. The value of this field varies in a multiprocessor system and remains constant in a single processor system.

- **Time Field**

  The time field indicates the delta time since the last request in clock ticks.