Early Design Cycle Timing Simulation of Caches

Preliminary Exam Report

Edward S. Tam
estam@eecs.umich.edu

Advisor: Edward S. Davidson
davidson@eecs.umich.edu

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan

September 9, 1996

Table of Contents

List of Figures

List of Tables

Abstract

Cache performance is a key component of a microprocessor's overall performance, as it is the cache that buffers the high speed CPU from the much slower main memory. Behavioral cache simulators indicate the performance of a given cache configuration in terms of the number of hits and misses experienced when running a piece of code. One can attach a leading edge penalty or "effective" penalty estimate to each miss to get a first order idea of run time. However, individual timing penalties are not assessed within these models, nor are the various factors that can affect each memory access' latency. Our Latency Effects (LE) cache model accounts for additional latencies due to trailing edge effects, bus width considerations, the number of outstanding accesses allowable, and port limitations.

A tool implementing the LE cache model has been built on top of a behavioral cache simulator, DineroIII, in the spirit of the Resource Conflict Methodology developed by J-D Wellman. The tool was combined with Wellman's RCM_brisc processor simulator to provide a realistic interaction of the cache with the processor (including the latency masking effects of processor activity) and to assess the accuracy of the model when simulating the execution of actual programs. The combined tool accurately mirrors the effects of changing a cache's configuration for a given processor configuration running a variety of programs. While the reported execution times do not exactly match the total execution times of the same programs running on actual hardware, the tool provides enough useful information to guide processor/cache designers early in the design cycle toward optimal configurations for target applications. This addition of cache modeling to the RCM_brisc instruction-level processor simulator with perfect cache increases simulation time by only 17% (less than 5% over a constant miss penalty cache model), which is reasonable given the added benefits of using the LE cache model.

1.0 Introduction

Cache performance is a key component of a microprocessor's overall performance, as it is the cache that buffers the high speed CPU from the much slower main memory. Behavioral cache simulators indicate the performance of a given cache configuration in terms of the number of hits and misses experienced when running a piece of code. Timing penalties are not assessed within these models, giving a false perception of a cache's actual impact on a system. In a simple model, which can be applied to the behavioral simulator output, each miss is simply assigned a minimum (leading edge) penalty, or an effective (average) penalty. However, there are actually a variety of latency-adding effects that depend upon such things as which words within a cache block are accessed (upon and soon after a miss), the width of the busses between the CPU and cache and between the cache and next level of memory, the number of outstanding accesses the cache can sustain, and the number of available ports to the cache.

Detailed circuit level simulators (timers) do characterize these additional effects, but these simulators are specific to a given machine. As opposed to behavioral simulators, which sacrifice detail for configurability, circuit level simulators target a specific machine and simulate its actual operation. Typically, circuit level simulators can only be created after the cache design is near its final, detailed implementation. A simulator that achieves a compromise between behavioral simulators and timers would be very useful for assessing the benefits of using a particular cache configuration in a given processor. Such a simulator could be employed much earlier in the design cycle than a traditional timer and could run at comparable speeds, or much faster, while producing results that adequately reflect the execution of a given program running on actual hardware. Furthermore, it would retain the flexibility of a behavioral simulator, allowing many configurations to be evaluated in a reasonable time while producing a much more realistic performance assessment. If the tool adequately models a majority of the effects that would be seen in normal execution, the performance estimate should be quite accurate, helping designers to identify the changes that are needed to obtain high performance at a reasonable cost.

We have implemented such a configurable cache simulator that is more realistic, and thus more accurate, than a behavioral simulator, yet is more paramaterizeable than a machine-specific timer. The tool uses an experimentally developed Latency Effects (LE) cache model for a machine of interest and some parameterized extensions to determine the access latency for each memory access in a trace. The access latency depends not only upon whether the access is a hit or miss, but on its relationship to other accesses in process, the width of the memory busses, the number of outstanding accesses that the cache can sustain, and the number of ports to the cache. Specific latency-adding effects are derived from the experimental model as a function of the specific reference patterns in the trace.

As a first step in assessing the correctness of the LE cache model, the cache simulator has been combined with J-D Wellman's RCM_brisc tool [Wellman95], which is based upon his Resource Conflict Methodology (detailed in Section 4.2). Together, the combined tool, RCM_brisc+LE, simulates an RS/6000-like (POWER) microprocessor with cache. Our simulation results show that for memory stressing codes, the output of the resulting simulator, RCM_brisc+LE, closely follows the trends seen by the same programs running on actual hardware. Furthermore, detailed analysis of those programs and the resulting output of the simulator show that the output corresponds closely to the results we would expect to see on a corresponding processor/cache configuration running those programs. In addition, a variety of useful statistics are provided by the new tool which serve to aid computer architects and programmers in the design and use of caches for target applications. These added benefits are provided with a maximum increase in simulation time of 17% over a processor-only simulator that assumes a perfect cache, and only 4.5% over a combined simulator that assumes a constant miss penalty.

Before we discuss the LE cache model and its implementation, we present an overview of caches and cache simulation in Sections 2.0 and 3.0, respectively. Then, the implementation of the LE cache model is presented in Section 4.0, followed in Section 5.0 by experimental results from the LE cache model implementation and an assessment of their accuracy. The cost of using the LE cache model is presented in Section 6.0, followed by a description of the simulator's output in Section 7.0. Conclusions are drawn in Section 8.0 and future work is discussed in Section 9.0.

2.0 Cache overview

A cache is a small, fast memory that is managed so that it contains recently accessed blocks of memory. The first level of the memory hierarchy encountered once the address leaves the CPU is generally a cache [Hennessy96]. The use of caches is based on the principle of locality, which says that most programs do not access all code or data uniformly. Instead, code or data is accessed in groups (spatial locality) or it is accessed repeatedly in a short period of time (temporal locality). To make use of locality, this small, fast memory is placed between the CPU and the slower, larger next level of memory. In general, several levels of cache may be used and separate caches at the same level may be used for instructions and data. The level of hierarchy targeted in this paper is the first level of data cache, known as the L1 data cache.

The L1 data cache (herein referred to simply as cache) is the smallest, fastest memory that the CPU can access. It can service memory requests at or near the CPU's execution frequency, but may incur longer latencies when requested information is not present in the cache. When an access is made to data not already allocated in the cache, the data must be obtained from the next level of the memory hierarchy. This process is called servicing a cache miss, and usually takes many more cycles to satisfy than a cache hit. Requests that reference data that is allocated in the cache return the desired data in less time than the full miss latency and are called hits to the cache. As we will see in Section 2.2, there are actually several types of hits with various access delays that they may suffer; for Section 2.1, the simple concept that a hit returns data immediately to the CPU will suffice.

2.1 Components of cache design

There are many well-known components to cache design. The cache is broken into chunks called blocks (also referred to as lines). A block is a collection of contiguous data that is treated as a single entity of cache storage. Blocks often consist of multiple words, with the typical word size being 32 or 64 bits for today's microprocessors. Larger blocks (e.g. blocks consisting of a greater number of words) take more advantage of spatial locality, as more data close to recently accessed data is stored in the cache within one atomic cache allocation unit. However, blocks that are too large sacrifice temporal locality, as having larger blocks for a fixed size cache reduces the number of different blocks that can be stored in the cache at once. Furthermore, using larger blocks can increase the miss latency, as a greater number of cycles may be required to fill a cache block with data from the next level of memory for a fixed size cache-to-memory bus. Larger blocks can also increase cache pollution because superfluous data may be brought into the cache within the referenced blocks. Blocks that are too small sacrifice spatial locality, as data close to, but not in the same block as recently requested data already present in the cache may have to be fetched from the next level of memory when it is requested. Typically, block sizes in today's caches range from 32 to 128 bytes.

Each block maps into a single set in the cache. A set is a group of blocks in the cache; in a k-way associative cache, a set can simultaneously hold any k of the blocks that map to it. Higher associativity provides more flexibility as to which blocks may simultaneously reside in a fixed size cache, and hence generally results in a higher hit ratio (the fraction of memory accesses found in the cache). However, increasing the associativity of a cache increases the number of tags that must be checked to see if the referenced data is in the cache. Thus, the time to access requested data in a highly associative cache grows due to the increased time required to perform and resolve the increased number of comparisons within a set to find the desired data. Due to its faster access and simpler implementation, the associativity of caches is kept low in most cache designs. Typically, today's caches are direct mapped, two-, or four-way set associative.

Every time memory is accessed, a check must be made to see if the referenced data is in the cache. If the data is in the cache, the reference is a cache hit, and the data is returned directly from the cache. If the desired data is not in the cache, the reference is a miss and the transaction must access a higher level of the memory hierarchy. In detail, a miss may be handled in different ways, depending upon the configuration and management policies of the cache and whether the access is a read or a write.

When a read miss occurs, a block in the cache is replaced with the desired data. There are several methods to determine which block should be replaced (Least Recently Used, random, and optimal, among others) [Hennessy96]. Some approximation of the LRU replacement algorithm is usually used in today's caches. Random replacement is sometimes used in large caches to reduce the implementation cost. Our simulations assume LRU replacement, but other strategies can be selected. The optimal algorithm may be used to determine an upper bound on cache performance; however, it is not implementable as it requires knowledge of the future in order to make its replacement decisions.

For all write accesses, be they hits or misses, the next level of memory must eventually be updated with the new changes. Two write policies are commonly used: write through and write back. In the write through policy, the information written to the cache is written to the next level of memory at the same time or soon thereafter. In write back, the information is initially written only to the cache; the modified cache block is written to the next level of memory only when it is replaced. For uniprocessors, the write back policy is normally used. In many cases, the data in a block may be changed multiple times before the block is replaced and then written out to the next level of memory. By delaying the update, multiple writes to a block can be grouped into one update, which generally results in higher performance.

When a write miss occurs, the referenced block need not be loaded into the cache. In write allocate caches, the block is loaded on a write miss; a block replacement process similar to that for read misses is then used, followed by the update of the block in the cache. In no-write allocate caches, however, the referenced block is modified directly in the next level of memory and is not loaded into the cache. Write back caches generally use write allocate, hoping that subsequent writes to that block will be captured by the cache. Write through caches often use no-write allocate, since subsequent writes to that block will still have to go to memory.

The width of the busses between the CPU and cache and between the cache and the next level of memory can affect the latencies of the memory accesses. Most accesses are word-sized, a word typically being 32 or 64 bits (4 or 8 bytes) in size. The busses between the levels of the memory hierarchy are usually some multiple of the word size, though they are rarely as wide as the size of an entire cache block. As a result, portions of the block, called subblocks, are filled on consecutive bus cycles by data returning from the next level of memory when satisfying a cache miss. For instance, if the cache-to-next-level-of-memory bus width is 1/4 of the size of a block, it will take four bus cycles to fill the entire cache block. The subblock containing the desired word is normally returned first; in this case, the block starts “filling” at the desired subblock and wraps around to the beginning of the block to complete the block fill. While this minimizes the time to access the desired (missing) word, the other words in the cache block must wait additional cycles before they are present in the cache; this additional wait can adversely affect the latency of future memory accesses, as we will see in Section 3.2.

Since the advent of pipelining, multiple memory accesses are often in flight at once. In a blocking cache, an access that follows a miss must wait to begin execution until the miss completes its access. A cache allowing hit-under-miss, on the other hand, would allow hits to complete while a miss is outstanding; a new miss would still have to wait until the earlier miss completes. A cache capable of sustaining more than one outstanding miss is called a non-blocking cache. The number of allowable outstanding accesses is defined as the maximum number of uncompleted misses that the cache can support while still allowing new memory requests to begin execution. Once this threshold is crossed, all future accesses, be they hits or misses, must wait until at least one of the outstanding misses completes before they can begin execution.

The number of ports to the cache also affects cache performance. A port is a point of access to the cache – it can either be a read port, write port, or both. Transactions can take place only when a port of the desired transaction type is available. If a read (load) is requested and no read ports are available, the access must wait until a read port becomes free; in most cases, the port becomes free the very next cycle, as a port is normally used for only one cycle per transaction. However, if a transaction uses the port for multiple cycles or there are older (pending) accesses waiting to use the desired port, the new access may have to wait additional cycles to obtain the use of the port. Increasing the number of ports to the cache can alleviate this problem, though this approach is usually avoided due to its high cost. Today's caches often have two ports so that potentially two accesses can be completed in each cycle.

2.2 Terminology

In this section, we review some of the terminology that will be used throughout this paper:

Cache hit – an access to the cache that requests data that is presently in the cache. For reads, the desired data is returned in `CACHE_HIT_LATENCY` cycles.

Latency-adding effects – effects a memory access may experience that add to its nominal execution time, e.g. trailing edge effects, bus width considerations, the number of outstanding accesses allowable, and port limitations.

Delayed hit – a hit that experiences latency-adding effects. These are called hits because they do not incur the full read or write miss latency; furthermore, they do not generate any additional miss traffic.

Miss – an access to the cache that incurs the full read or write miss latency. (Misses are further categorized below).

Cold-start miss – a miss access to the cache that occurs because the block that contains the desired data has never been accessed before. Also known as a compulsory miss.

Capacity miss – a miss access to the cache that is not a cold-start miss, but occurs because the entire working set of the program cannot simultaneously reside in the cache. The number of capacity plus cold-start misses is the total number of misses that would occur in a fully associative cache of the same size and block size with optimal replacement.

Conflict miss – all other miss accesses to the cache, which occur because more than k blocks of the working set map to the same set of a k-way associative cache, and additional misses that may be due to the nonoptimality of the replacement policy, thereby causing some data to be replaced during execution.

Read miss latency – the nominal time to satisfy a read (load) request that misses in the cache, e.g. the time to return the desired data from the next level of memory to the cache under ideal circumstances.

Write miss latency – the same as read miss latency, except for write (store) requests.

3.0 Overview of cache simulation

Cache simulation is widely used to determine the performance of a given cache configuration for the execution of a target application. This evaluation can be done using behavioral cache simulators, circuit level simulators (timers), and via the measurement of actual systems (using hardware monitors). Behavioral cache simulators are highly paramaterizeable, but they do not represent the access latencies of a target machine accurately. Circuit level simulators, on the other hand, are extremely accurate for a single machine, but are not paramaterized to evaluate a variety of dissimilar machine implementations. Finally, the measurement of an actual system, while useful in gauging performance, requires a completely implemented, fully operational system, and even then may not give designers or programmers a good idea of the underlying causes of the measured performance. The Latency Effects (LE) cache model and its implementation improves upon these techniques by incorporating more of the effects that a memory access can experience than a behavioral cache simulator does while providing the flexibility to change the configuration of the target cache for each simulation run. Furthermore, the statistics output by the LE tool aid the designers and programmers in determining the bottlenecks and underutilized resources of the configuration.

3.1 Currently available behavioral cache simulators

Many behavioral cache simulators are currently available, including DineroIII [Hill85], ACS [PARL95], Fast-Cache [Lebeck95], and others. These simulators take cache design parameters, such as cache size, block size, and associativity, together with a sequence of memory accesses as input and determine the number of cache hits and misses that would occur if the code were run on a processor with a cache of the corresponding parameters. Each memory access is analyzed individually, and its result (whether it is a hit or miss to the cache) is dependent upon the state of the cache at the time of the access. The job of the simulator is thus to maintain the cache state and decide whether each successive reference is a hit or a miss. After each access is evaluated, its effect on the cache is immediate, e.g. if a block of data is loaded into the cache by one access, all of that data is considered to be present and immediately accessible in the cache when the very next access arrives.

3.2 The LE cache model

While knowledge of the number of cache hits and misses is useful, knowing the effects of those hits and misses on a program's execution is essential for guiding the design process. One way to add this functionality to existing cache simulators is to attach a latency to each access. This addition enables us to determine the number of cycles required to execute a given sequence of memory accesses.

However, this approach assumes that all memory accesses are independent and decoupled, i.e. once the earlier access to the cache line is evaluated, the requested data is immediately present in the cache until it is replaced. In actuality, if an earlier access has not fully completed before a new access to the same cache line occurs, a trailing edge effect may be seen. The new access to the cache line will then require more than `CACHE_HIT_LATENCY` cycles to complete because the referenced data, although allocated, is not yet actually present in the cache yet. However, since an earlier access to the cache line is already in flight, this new access will not incur the full read or write miss penalty. Furthermore, performing this new access to the cache line does not generate any additional miss traffic, as the desired data is already in transit from the next level of memory to satisfy the earlier request. Current general simulators do not address trailing edge effects, as they assume that after an access is made to a cache line, all subsequent accesses to that line are cache hits (until the line is replaced). We call such a reference that experiences trailing edge effects (and other hits that experience other latency-adding effects) a *delayed hit.* Thus we divide accesses into cache hits, delayed hits, and misses.

Trailing edge effects can have an enormous effect on a program's actual execution time. The greatest impact is made when there is a series of memory accesses as in Program 1:

```
1 LDF A
2 LDF B
3 LDF C
4 LDF D
```

Figure 1: Program 1

Suppose that the data in memory is laid out as follows:

Byte Offset:	0	8	16	24	32	40	48	56
Data:	A	B	C	D	E	F	G	H

Figure 2: One block-aligned portion of memory (blocksize = 64B)

For simplicity, suppose that access requests are sent to the cache in the first Execute stage of the processor pipeline and that loads complete in the same cycle that the data returns from the cache. Assuming that the access to address A is the first access to that cache line, the LDF (floating-point load) at line 1 will always miss (it is a cold-start miss). In a cache with a block size of 64 bytes, the three subsequent LDFs will be recorded as hits to the cache in a cache simulator such as DineroIII, since each load requests 8 bytes of data and all the desired data is in the same block. This situation is shown below in Figure 3:

```
Cy IX--------W
1) A
2) BA------->A (miss)
3) CB------->B (hit)
4) DC------->C (hit)
5)  D------->D (hit)
```

Figure 3: Execution behavior of Program 1 according to unmodified DineroIII

In the figure above, Cy indicates the cycle number, I indicates the issue stage of the pipeline, X shows the start of the execution phase, and W is the writeback stage.[1] Without any modification, DineroIII would indicate that all the accesses complete in the same cycle that they begin execution, regardless of whether they hit or miss. The information about whether the access actually hits or misses in the cache is recorded for statistical purposes. If DineroIII were used to model memory accesses in conjunction with a processor simulator, all Execute and Writeback stages for these loads would be merged into one cycle, and this sequence of instructions would require five cycles to execute.

[1] The target machine in this example is a uniprocessor with a single load/store execution unit that can handle one memory request per cycle. This machine model will be used throughout this study to simplify the analysis; the following examples can easily be extended to execute on a machine allowing multiple memory accesses per cycle.

The first step to improving this model would be to add leading edge latencies to each memory access. These latencies would describe the time that the access spends in the Execution and Writeback stages of the pipeline, with the Writeback stage usually requiring a single cycle. For instance, the access latency for a load miss could be 10 cycles (9 cycles for Execute and 1 cycle for Writeback), whereas a load hit would only require 2 cycles (1 cycle each for Execute and Writeback). The simulation of Program 1 with DineroIII augmented with these latencies is shown in Figure 4.

```
Cy IX--------W
1) A
2) BA
3) CBA
4) DC A      B
5)  D  A     C
6)      A    D
7)       A
8)        A
9)         A
10)         A
11)          A
```

Figure 4: Execution behavior of Program 1 for leading edge latency-augmented DineroIII

Here, the program execution is more realistic in terms of overall execution latency (11 cycles total). However, the accesses to B, C, and D cannot complete before the access to A, since A, B, C, and D all reside in the same block of memory and the earliest datum to return to the cache is A! At the earliest, the accesses to B, C, and D can return to the processor in cycle 11 along with the access to A; completing any earlier would not be possible because the data would not be in the cache yet!

As introduced earlier, the accesses to B, C, and D suffer from trailing edge effects caused by A: since the block brought in by the access to A is not available when B, C, or D execute, and since all four accesses reside in the same cache block, B, C, and D must wait additional cycles to complete. If trailing edge effects were incorporated into the cache model, the execution of Program 1 would look like:

```
Cy IX--------Wbck
1) A
2) BA
3) CBA
4) DCBA
5)  DCBA
6)   DCBA
7)    DCBA
8)     DCBA
9)      DCBA
10)      DCBA
11)          ABCD
```

Figure 5: Execution of Program 1 when trailing edge effects are taken into account and 4 read ports are available

Here, we see that all four accesses complete in cycle 11. This makes sense because it is possible that all the data in the cache line is available at cycle 11 and the data can be returned to the desired functional units in the same cycle (e.g. there are four available read ports to the cache for that cycle). The time to wait for the desired data is represented in the additional stages of Execute that those accesses must experience. As shown in Figure 5, the Execute and Writeback stages represent the miss pipeline in the cache subsystem. Different machines may handle the actual misses in this pipeline differently, but the end result is the same: the desired data is not available until the required latency has passed. Thus, this representation of the access' execution shall suffice for understanding the movement of different memory accesses through the cache subsystem.

The accesses to B, C, and D are all delayed hits, since they do not incur the full miss latency, yet they do not complete in the time required for a cache hit. Also, these accesses do not generate any additional miss traffic, as the required data is already in transit to satisfy the earlier request to A.

While a machine could be built to load an entire cache block in one bus cycle, it is likely that the cache block is loaded from main memory in multiple bus cycles (e.g. one word per cycle, with the requested word returning from memory first). Thus, in addition to trailing edge effects, a memory access can experience increased execution times due to bus width considerations. If successive data words of the block are returned to the cache in successive cycles, the program execution would look like:

```
Cy IX--------W
1) A
2) BA
3) CBA
4) DCBA
5)  DCBA
6)   DCBA
7)    DCBA
8)     DCBA
9)      DCBA
10)      DCBA
11)       DCBA
12)        DCB
13)         DC
14)          D
```

Figure 6: Execution of Program 1 with words from memory returning in sequential cycles or with only 1 read port to the cache

There may also be port limitations that affect an access' latency – in the program execution of Figure 5, four read ports from the cache are assumed to be available at once. Typically, processors today have one read/write port to the cache; some have more than one port and in such cases, some ports may be read-only or write-only. If the target processor has one read port to the cache, regardless of whether data is returned from memory simultaneously or sequentially, Program 1 would execute as shown in Figure 6, above. Since there is only one read port, only one of the words can be returned to the processor from the cache in each cycle. In this example, subsequent requests must wait for earlier ones to complete before a port is freed and the desired word can be transferred from the cache to the processor.

Words in memory are often accessed nonsequentially. For instance, if we run Program 2 with the memory layout shown in Figure 8,

1 LDF I
2 LDF J
3 LDF K
4 LDF L

Figure 7: Program 2

Byte Offset:	0	8	16	24	32	40	48	56
Data:	I	K	L	J	M	N	O	P

Figure 8: Another block-aligned portion of memory (blocksize = 64B)

the program's execution might look like:

```
Cy IX--------W
1) I
2) JI
3) KJI
4) LKJI
5)  LKJI
6)   LKJI
7)    LKJI
8)     LKJI
9)      LKJI
10)      LKJI
11)       LKJI
12)        LJK
13)         JL
14)          J
```

Figure 9: Execution of Program 2 with words returning from memory in sequential cycles

While the overall program execution time remains the same as Program 1's execution time (14 cycles), the desired data returns in a different order; this is entirely a function of the location of the requested words of data in memory. For Program 1, the data returned in the order it was requested. In Program 2, the data returned in the order that the data was retrieved from memory (first I, then K, L, and J). Typically, caches fill their blocks from the next level of memory in the order that the data is found in memory, not the order the data is requested. Thus, instead of returning one cycle after I, the access to J returns three cycles after I returns since it is the last word of the four in the block to be returned when word I is requested first. The increased latency required to complete the access to J is shown by the increased time that J spends in the Execute stage; the accesses to I, K, and L complete as their desired data is loaded into the cache block. This effect is not very apparent in a simple program consisting of only memory accesses; however, if there were a fixed- or floating-point operation that depended upon the data at address J, that dependent instruction would have to wait three additional cycles: one cycle due to the cache being unable to load the entire cache block in one cycle and two more cycles due to the placement of J within memory.

There is one more effect that we must consider. Most caches cannot continue accepting requests if there are a number of misses that have yet to be satisfied (these misses are termed "outstanding accesses"). Once the threshold of outstanding accesses is reached, subsequent accesses may be delayed until the earlier accesses complete; until at least on of these outstanding accesses completes, the cache is said to be *blocked*. If Program 1 were run on a processor whose cache could only sustain two outstanding accesses at a time, its program execution would look like:

```
Cy IX--------W
1) A
2) BA
3) CBA
4) C BA
5) C  BA
6) C   BA
7) C    BA
8) C     BA
9) C      BA
10)C       BA
11)DC       BA
12) DC       B
13)  D       C
14)          D
```

Figure 10: Execution of Program 1 on a processor whose cache can only sustain two outstanding accesses at a time with 1 read port to the cache

Since neither A nor B are complete when C tries to enter the Execute stage, C must wait because the cache is blocked. C does not enter Execute until the first outstanding access, A, exits Execute in cycle 11. Since C was blocked in the Issue stage, D does not get to enter Issue until C exits that stage. (In more complex processor configurations, C and D would be blocked together in a processor buffer somewhere outside of the Execute stage).

Once C enters Execute, it finds that the data it wants is already in the cache, thanks to the earlier miss to the block by A. Thus, C finishes one cycle after B (due to port conflicts) and D completes one cycle after that. Since all the accesses were to the same block of the cache, the blocked cache does not adversely affect the execution time of Program 1, as its execution time remains 14 cycles.

If we were to execute Program 3 on the same (two outstanding accesses sustainable) processor and cache with the memory layouts shown in Figures 2 and 8, the program execution would look different:

1 LDF A
2 LDF B
3 LDF I
4 LDF K

Figure 11: Program 3

```
Cy IX--------W
1) A
2) BA
3) IBA
4) I BA
5) I  BA
6) I   BA
7) I    BA
8) I     BA
9) I      BA
10)I       BA
11)KI       BA
12) KI       B
13)  KI
14)   KI
15)    KI
16)     KI
17)      KI
18)       KI
19)        KI
20)         KI
21)          K
```

Figure 12: Execution of Program 3 on a processor whose cache can only sustain two outstanding accesses at a time

In Program 3, the misses to A and I are to different blocks in the cache. I must wait until A completes before it enters execute, since A and B are outstanding when I enters the issue stage. But, once I enters execute, it finds that it, too, is a miss, and must wait 10 cycles before completion. K, being a trailing edge access to the same block as I, must wait 10 cycles to complete as well. Thus, Program 3 requires 21 cycles to execute, as opposed to 14 cycles for Program 2 running on a similar machine.

The Latency Effects (LE) cache model accounts for the nominal hit and miss times, plus the added delays due to each of the aforementioned effects of prior accesses on the timing of the next access (trailing edge effects, bus width and port limitations, and the number of outstanding accesses allowable). By using the LE cache model instead of a currently available behavioral cache simulator like DineroIII, a more accurate simulation of the execution of memory accessing instructions can be obtained.

4.0 Implementing the LE cache model

In order to test the correctness of the LE cache model, a cache simulator incorporating this model was built. Instead of writing a basic cache simulator from scratch, DineroIII was used as a basis, as it is a widely-used and highly paramaterizeable behavioral cache simulator. While there are many ways that the delays associated with the LE cache model can be realized, an approach based on the Resource Conflict Methodology was used. DineroIII and the Resource Conflict Methodology are described in the next two sections, followed by the implementation of the LE cache model.

4.1 DineroIII

DineroIII [Hill85] is a parameterizeable, trace-driven cache simulator developed by Mark Hill. DineroIII takes a trace of memory accesses as input and determines, for each access, whether the access hits or misses based on the state of the cache at the time of the access. The cache that is simulated can be modified based on associativity, cache block size, overall cache size, and update policy, among other parameters. Statistics reported at the end of the simulation include the number of read and write hits and misses to the cache, the number of words transferred, and the total number of memory accesses.

4.2 Resource Conflict Methodology

The Resource Conflict Methodology (RCM) [Wellman95] was proposed by J-D Wellman as a technique for modeling and simulating computer systems early in the design cycle. Each element of the simulated processor is viewed as a resource that may be unavailable at a given time. For instance, an RCM model analyzes the effect of each instruction's execution on a given machine's resources. Each instruction requires a certain set of resources to execute. As the instructions are executed, the availability of each resource that is used is updated, delaying the times at which those resources are available to subsequent instructions. Given the resource constraints, the execution time of a program can thus be determined by keeping track of when instructions are allowed to execute and when they complete.

4.3 Implementing LE on top of DineroIII

DineroIII keeps the current state of the cache in memory during simulation. The cache is "updated" when each new access is made: If the access is a hit, the data is already in the cache, but it may be marked to reflect that it is the most recently accessed datum (for a replacement policy such as LRU). If the access is a miss, any block replacement that is required is performed and the desired data is placed in the cache immediately. Any subsequent accesses to that data block (until it is replaced) are thus recorded as hits, since according to DineroIII's cache state, the data for that block is in the cache.

However, due to leading and trailing edge effects, data does not return to the cache from the next level of memory immediately. These effects can easily be incorporated into DineroIII by controlling when DineroIII's state is updated, i.e. when DineroIII "sees" a memory access. Given the cycle in which the access begins execution, we can determine the effects that it sees during execution and when it completes – and thus when DineroIII's state should be updated.

Each memory access is evaluated in turn, since we assume only a single memory access is permitted per cycle.[2] First, the DineroIII cache is checked to determine whether the access is a hit or a miss. If DineroIII says that the access is to data currently in the cache, the access is tentatively marked a cache hit. This access must wait `CACHE_HIT_LATENCY` cycles before the data is available for the requesting functional unit. Usually, data is returned in same cycle that it is requested when a cache hit occurs, so the `CACHE_HIT_LATENCY` is normally equal to one. Thus, the cache hit completes in the same

2 The single memory access per cycle limitation is due to the target machine we chose to emulate in this study. However, due to its implementation based on the RCM model, the LE cache model can inherently handle multiple accesses per cycle.

cycle in which it begins execution and the data is available for use in the processor in the very next cycle.

If an access is tentatively marked as a hit and a dependency of the second type is found (e.g. the requested data will be replaced by a currently outstanding miss access), the access is then handled as a cache miss, as explained in Section 4.4.5. If no dependencies are found, the access is indeed a cache hit, and the DineroIII cache state is updated to reflect that that block is the most recently accessed reference.

If an access is tentatively marked a cache miss, the access is not necessarily a miss that would incur the full read or write miss latency. The access may be a delayed hit, as an outstanding miss access may already have requested the desired block. Thus, a check is made to see if the new access is dependent on any currently outstanding accesses. If the access does depend on a currently outstanding access, the access is a delayed hit, which to DineroIII seems like a miss, as the desired data is not present in DineroIII's cache due to the delayed update that we have imposed on the DineroIII cache state. However, since the desired block is already in transit from the next level of memory (due to the earlier dependent miss to that block), the new access should not automatically incur the full read or write miss latency. This is the beginning of the implementation of the LE cache model outside of DineroIII.

Once the access has been completely evaluated by the LE cache model, the completion cycle for that access is known. The effects of the access on the state of the DineroIII cache are enacted (i.e. the cache is updated) when the cycle of the simulation is greater than or equal to the completion cycle of the access. The update of the DineroIII cache state is deferred for each access until the data is actually resident in the cache. This way, if an access is determined to be resident in the DineroIII cache state at any given time, the data will be available immediately (except in the case where a dependency of the second type is discovered) as a cache hit; otherwise, the access is a delayed hit or cache miss. The updating of the DineroIII cache state will be explained in greater detail in Section 4.4.5.

4.4 The LE cache model outside of DineroIII

As discussed in Section 3.2, there are four main sources of additional delay for a memory access:

- Trailing edge effects
- Bus width considerations
- Number of allowable outstanding accesses
- Port limitations

Each one of these effects potentially affects the completion time of an access to the cache (and the subsequent access to the next level of memory for cache misses). Each effect and its implementation is now discussed in turn.

4.4.1 Trailing edge effects

In order to quantify the trailing edge effects experienced by a cache access, the status of each outstanding access to the cache must be retained. Cache hits, on the other hand, can be evaluated as they occur, as they can only be affected by previous instructions. A cache hit can only be affected by prior delayed hits or misses because of its port requirements (discussed in Section 4.4.4). For instance, if a cache hit is issued in cycle X, it will complete in two cycles (using the example latencies from Section 3.2), i.e. in cycle X+2. For our model machine, the earliest time that a new memory accessing instruction can be issued is in the very next cycle, cycle X+1. If the new access is also a cache hit, it will complete in cycle X+3, and will not depend in any way on the earlier cache hit. If the new access is a delayed hit or a miss, the access will complete in cycle X+N, where $N \geq 4$. However, the first cache hit will complete in cycle X+2, which is earlier than any subsequent access (be it a hit, miss, or delayed hit) can complete. As a result, the cache hit will not be affected by any accesses that issue after the cache hit is issued, regardless of the outcome of the subsequent accesses. However, we will see in Section 4.4.4 that cache hits can be affected by prior delayed hits or misses whose latencies are greater than the latency of a cache hit.

Misses can affect the completion times of many subsequent accesses. The number of future accesses affected increases in proportion to the duration of the full miss latency. To determine these effects, the status of a cache miss is kept until the access is satisfied, i.e. until the requested data is finally resident in the cache. All outstanding accesses are ordered chronologically in a linked list called the `update_buffer`. When an instruction accesses data not currently in the cache, the `update_buffer` is checked to see whether the desired data is currently in transit from the next level of memory to the cache. A *dependency* is found if one of two situations occur: 1) the desired data coincides with a cache block that is currently being brought into the cache from the next level of memory or 2) the desired data is currently being replaced by an outstanding access. For dependencies of the first type, instead of incurring the full miss latency of the cache, the new access will be satisfied in some shorter period of time related to the remaining time to satisfy the earlier outstanding access to the same cache block. Dependencies of the second type will require the nominal miss latency for that access type to complete, as the desired data will need to be fetched from the next level of memory to satisfy the access. As we will see in subsequent sections, this is only the first part of determining the completion time of a delayed hit or miss; once the earliest time to completion is determined (when the earlier conflicting miss completes), bus width considerations, the number of allowable outstanding accesses, and port conflicts must be taken into account. These will be discussed in detail in Sections 4.4.2, 4.4.3, and 4.4.4, respectively.

The `update_buffer` corresponding to the execution of Program 1 at cycle 4 is shown in Figure 13. At this point in time, the access to C is beginning execution. We see that the first entry in the `update_buffer` linked list is the load from address A. The next entry in the linked list, which corresponds to the next outstanding access to complete, is the load from address B.

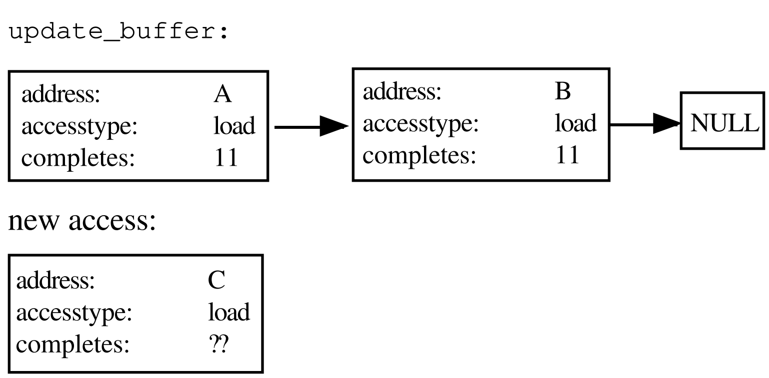

Figure 13: State of update_buffer when the access to address C is being evaluated

The load from address C will check the `update_buffer` and find that its desired data will be present in the cache at cycle 11, thanks to the earlier request to that cache block by the access to A. Since it is only cycle 4, this access will also be placed in the `update_buffer`, as it will complete in a later cycle and may affect the access time of a future memory accessing instruction. Assuming there are enough ports to satisfy three read requests from the cache in one cycle and the bus between the cache and the next level of memory is wide enough to return all three access to the cache in the same cycle, the load from address C will also complete in cycle 11. The updated `update_buffer` is shown in Figure 14.

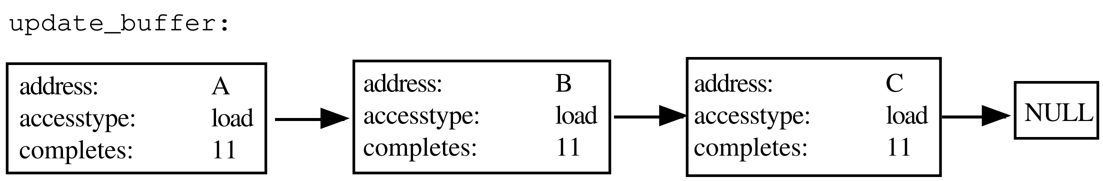

Figure 14: State of update_buffer after access to address C has been evaluated

Every delayed hit and miss goes through this process and eventually ends up in the `update_buffer`. If the access is a miss, i.e. the desired cache block is not present in the DineroIII cache state, no dependencies will be found with the entries currently in the `update_buffer`. Dependencies of the second type cause hits or delayed hits to become misses, as the desired data will be replaced by an outstanding access. These accesses will incur the full miss latency for its type (read or write), plus any additional cycles due to port conflicts (see Section 4.4.4). An entry is made for the new access and it is placed in the `update_buffer` so that future accesses can check against this new outstanding access as well.

By assigning differing latencies for each access depending upon its time of execution, we can more accurately predict the time to execute a series of memory access instructions. Using its first cycle of execution as the starting point for each access, the completion time for that memory access can be determined by looking at the outstanding accesses to the cache, combined with the knowledge of the read and write miss latencies for the given cache. Using varying latencies is the first step toward a more realistic cache performance estimate.

4.4.2 Bus width issues

Another parameter that affects when an access completes is the bus width between the cache and the next level of memory. The data bus of a port between the CPU and the cache is normally at least the width of one access (typically a 32-bit word or a 64-bit doubleword), so requests will complete without being affected by this bus width. However, the bus width between the cache and the next level of memory is normally smaller than the block size of the cache. For instance, in the RS/6000 Model 320H [Hardell90], the block size is 64 bytes, but the cache-to-memory bus is only 8 bytes wide, causing a block of data to return to the cache in multiple cycles.

Typically, machines are designed to return the requested word within the cache block first, with subsequent subblocks (a subblock being the width of the cache-to-memory bus) arriving in subsequent cycles to the cache. The other alternative is to always return the first subblock of the block first, forcing all but the first subblock in the block to wait additional cycles for a miss access to them to be satisfied. Since most of today's machines use the former technique, the following discussion will concentrate on that method; the analysis can easily be modified to use the second cache fill policy.

Given the requested-word-first cache fill policy, the first miss to a cache block will take the required full read or write latency to complete. A subsequent miss to the same cache block that occurs while the earlier miss is in flight may or may not be satisfied in the same cycle as the earlier miss. If the subsequent miss is to the same subblock as the earlier miss, both accesses will complete in the same cycle (ignoring port conflicts for now). If the subsequent miss is to a different subblock, that access will have to wait additional cycles to be satisfied. Since the cache block is filled in a wrap-around fashion, the later access will have to wait at most an additional [(block size in bytes)/(cache-to-memory bus width in bytes) - 1] cycles to complete. If the later access is in the subblock to return immediately after the first requested subblock, the later access will complete one cycle later than the earlier access, pending port conflicts. If the later access returns in the third subblock, it will wait two additional cycles, etc. An example of bus width issues affecting memory access latencies was shown in Figure 9 with the execution of Program 2.

Given a cache's block size and cache-to-memory bus width, we can easily determine these additional delays given when the accesses occur relative to one another. The additional cycles are simply added on to the completion cycle that was determined for this access in the initial trailing edge analysis. These additional cycles will not and should not affect any earlier access' completion times, but will affect the completion times of the current and future accesses. The completion cycle of the access is stored in the access' `update_buffer` entry so that future accesses can determine when this outstanding access completes.

4.4.3 Number of outstanding accesses

The number of outstanding accesses that a cache can sustain affects the completion time of any memory access. Caches that do not support hit-under-miss will stall whenever there is a miss in the cache. This means that any subsequent memory accesses, even if they were to access data that was present in the cache at that time, would be stalled until the outstanding access completed. Obviously, future misses to currently absent data would also be stalled, though the delay due to the blocked cache would likely remove some of the trailing edge effects. (For an example of this situation, see Figure 10, which shows delayed hits turning into cache hits due to the blocked cache). When the cache becomes unblocked, the first waiting access begins execution. Since there are no outstanding accesses when this new access executes, it will not experience any latency adjustments due to trailing edge effects. While this case is handled by the LE model, the case where hit-under-miss is allowed is much more interesting. The case where a miss occurs while the cache is blocked was shown earlier in Figure 12 with the execution of Program 3.

There can be a varying "number of outstanding accesses" (NOA) when hit-under-miss is allowed. If hits are allowed when only one miss is outstanding, the NOA is equal to one. This means that we can have one miss access to the cache outstanding at a time, but we can still service hits to the cache. If another miss occurs while an earlier miss access is outstanding, the processor stalls all memory accesses until the earlier miss is satisfied. After the first miss is completed, there is once again only one miss access outstanding, and hits can once again be handled if they occur. Increasing the NOA increases the number of memory accesses that can be in flight at one time, giving a corresponding increase in the number of accesses that could potentially be affected by trailing edge effects. Still, once the threshold is crossed, all accesses, be they hits or misses, must wait until at least one earlier miss is satisfied before they can execute.

To account for NOA, we simply need to check to see how many misses are outstanding when we evaluate a new memory access miss. If the number of outstanding misses is less than the NOA threshold, the current access' completion time is wholly dependent upon trailing edge effects, bus width effects, and port limitations. If the number of outstanding accesses is greater than the NOA threshold, this access must wait until enough misses are satisfied to reduce the number of outstanding accesses to less than the NOA threshold. If the new access depends on an outstanding access, the new access will complete `CACHE_HIT_LATENCY` cycles after it is allowed to execute, as the data would be in the cache by the time the request is actually made to the cache. If the new access does not depend on any outstanding access and is simply delayed due to the blocked cache, the normal trailing edge and bus width effects analysis will be applied, once the cache is no longer blocked and the access is allowed to begin execution.

If the new access is a miss, the new access would have to incur the full miss latency before it would complete. This latency would be added on to the cycle that the access was finally determined to execute to determine its completion time. As above, the new access would execute when enough misses are satisfied to reduce the number of outstanding accesses to less than the NOA threshold.

4.4.4 Port limitations

The final element to consider when determining an access' completion time is the availability of an appropriate port to the cache. After the completion cycle of an access is determined by considering trailing edge and bus width effects, and the number of outstanding accesses, we must check to see if the desired port for the transaction (read or write) is available in the desired completion cycle. If an appropriate port is available, the completion cycle remains unchanged and we have found the actual completion time of the access. If an appropriate port is not available, we must check subsequent cycles to see when an appropriate port does become available. This is done by scanning through subsequent cycle times, starting at the access' completion cycle + 1. The search is continued until an appropriate port is found and that cycle count is returned as the actual completion time. Thus, the access completes as soon as it possibly can, considering the trailing edge and bus width effects, the location of the word in memory, the number of outstanding accesses sustainable, and the port limitations.

Port conflicts affect hits to the cache as well. If there are earlier misses to the cache that complete in the same cycle as the new hit and there are not enough ports available to satisfy the hit to the cache, the hit must wait for one or more additional cycles. The same method described above for misses is used to find the hit's completion time.

Obviously, increasing the number of ports will reduce the likelihood that port conflicts occur. With a greater number of ports available, more requests to the cache can be returned to the processor each cycle. On a related note, if the number of outstanding accesses allowed is less than the number of ports to the cache, port limitations will only affect hits to the cache. This occurs because we would never have more misses outstanding than there are available cache ports, so the only time there will be contention for cache ports is when there are hits to the cache that complete in the same cycle that a pending miss completes. Memory accesses are evaluated sequentially in time, so older accesses reserve their use of a port before newer accesses do. Thus, even though an access may be a cache hit, if all the ports for the cycle in which the hit should complete are reserved for prior accesses, the hit must wait at least one additional cycle to obtain an available port and complete its memory access.

4.4.5 Flow chart diagramming the operation of the LE cache model

All memory accessing instructions go through the same steps of evaluation in the LE model. Once a completion cycle has been determined, that cycle, minus the cycle that execution of the instruction started, is returned as the latency of the instruction. The flow chart of the operation of LE is shown in Figure 15. At each of the "endpoints" of the flow chart, the completion time for the current access is finalized and that value is returned to the calling program.

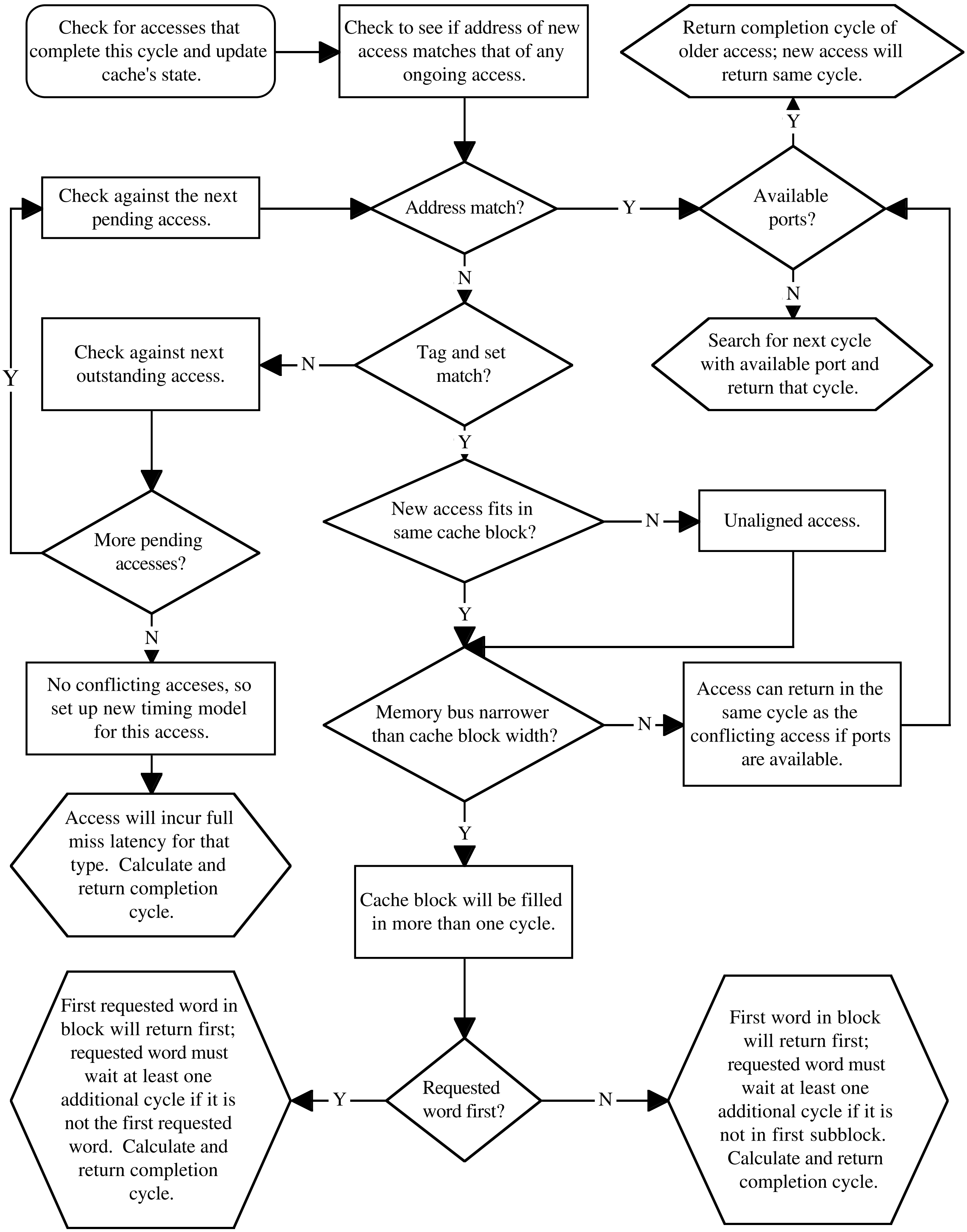

Figure 15: Flow chart showing the operation of the LE cache model

The state of the DineroIII cache is not updated until the cycle count of the currently evaluated memory accessing instruction is greater than or equal to the completion cycle of an outstanding access in the `update_buffer`. Delaying the update of the cache state until a new access occurs presents no problems, as the state of the cache is "corrected" to reflect the recently completed transactions before a new access is evaluated in LE. If the new access begins execution after an older, outstanding access completes, the new access should not depend upon that older access, and indeed it does not, since all outstanding accesses in the `update_buffer` are removed from `update_buffer` when they are completed and the cache's state is updated before each new access is evaluated. Since only memory accessing instructions can affect the state of the cache, delaying these updates reduces simulation time, especially during long stretches of compute-only code, as no checks need to be made on the cache unless a new instruction accesses memory. If the new access begins execution before some currently outstanding access completes, the new access may experience latency-adding effects caused by the outstanding access(es). Since the outstanding access is still in the `update_buffer`, a dependency may be found with the new access, causing the LE cache model to adjust the new access' completion time accordingly.

Currently, unaligned accesses are simply flagged in LE; special mechanisms by which to evaluate unaligned accesses have not been implemented. This case was not considered for two reasons: First, unaligned accesses are rare, undesirable, and avoidable, so not incorporating them into our model should not have a great impact on the model's performance. Secondly, there are many different methods used to handle unaligned accesses, so many so that incorporating a specific handling method would sacrifice the general purpose nature of the LE cache model. If a relatively simple, parameterizeable method for handling unaligned accesses can be derived, it will be incorporated into LE in the future.

4.5 Using the RCM model to implement LE

With regard to instruction execution, the LE cache is modeled as single resource. Memory accessing instructions check the cache (LE) resource to determine when execution can begin and when execution completes. The start and end times of each instruction's execution are determined by checking the various effects that an access can encounter, as detailed in Section 4.4. Given when a memory accessing instruction completes, dependent instructions are delayed accordingly in the corresponding portions of the RCM simulator (e.g. dependent memory accessing instructions delay their issue times with respect to the cache and dependent execute instructions delay their issue time to the functional units). Since LE is modeled as a single resource, it can easily be joined to an RCM-based processor model to simulate the execution of code on a processor with cache.

Within the LE cache model itself, the cache blocks, busses to and from the cache, the ports, and the cache itself are viewed as resources. If a desired cache block is not available, the requesting instruction must wait a at least a nominal number of cycles until the data is returned to the cache. If the bus to the cache from the next level of memory is currently used to return an earlier requested word, the newer requested word from same cache block must wait additional cycles. Transfers between the cache and the processor are taken care of by the port resources; if a machine has a certain number of ports to the cache, there will be a corresponding number of simultaneously available busses between the cache and the processor. If a port of the desired transaction type is not available, the instruction must again wait additional cycles before it can complete execution. Also, if the cache itself is blocked, all further accesses must wait until some earlier miss completes before execution can continue. Thus, the LE cache model is implemented using the RCM methodology by modeling each effect accounted for in the LE cache model as determined by the availability of some resource.

5.0 Experimental results

In order to test the validity of the new cache model, a tool was built to simulate the memory access performance of a given program. The tool was written in the spirit of J-D Wellman's Resource Conflict Methodology, where instructions take differing amounts of time to execute depending upon resource availability. With regard to memory accesses, resources within the LE model include the ports, the cache (available unless blocked), and data (which becomes available when each access to it completes and remains available until the next access to it begins execution in the cache).

The processor and cache simulator combines J-D Wellman's RCM_brisc tool, which simulates an RS/6000-like [Bakoglu90] machine, with the LE cache model. Based on the REAP tool, RCM_brisc simulates the execution of instructions fed to it in the form of a trace of the program's execution on an actual machine, which in this study is an RS/6000. The RCM_brisc tool by itself simulates the execution of all instructions, but assumes a perfect cache model, where all data from memory is available in a fixed, predetermined amount of time. However, the perfect cache model is an unrealistic estimate of program performance in today's processor/cache designs; cache and memory effects must be included in any processor simulation if it is to realistically evaluate a program's performance on a system under consideration.

The LE cache model, in its current implementation, could easily have been combined with any other currently available instruction-level simulators such as Talisman [Bedichek95], SimICS [Magnusson95], and others. This is possible because this implementation of the LE cache model maintains the state of the caches itself and does not take into account virtual memory or TLB effects. It models the first level of data cache and assumes a perfect memory thereafter, regardless of the number of level of caches beyond that. This simplification still provides accurate pictures of program execution, as we will see later in this section. The RCM_brisc tool was chosen since the LE cache model was implemented in the same spirit using the RCM model, and, more importantly, the creator of the RCM_brisc tool was readily accessible (his desk is next to mine).

The combination of the LE cache model implementation with the RCM_brisc processor simulator involved several pre-existing simulators, namely RCM_brisc for core processor simulation and DineroIII for behavioral cache simulation. The LE cache model implementation was built on top of, and thus includes, DineroIII in this realization. LE was then fused to RCM_brisc to "service" memory accesses and determine their execution times. A high level picture of the overall processor/cache simulator is shown in Figure 16; the white portions detail currently existing simulators while the gray, highlighted portion indicates the new work discussed in this paper.

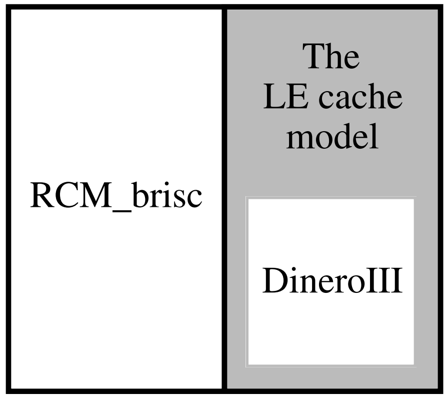

Figure 16: A high-level picture of the overall processor/cache simulator, RCM_brisc+LE.

The LE cache model implementation handles a wide variety of cache configurations, as determined by the inputs to the simulator. A list of the inputs handled by the LE cache simulator are shown below in Table 1.

cache size	number of read ports
block size	number of write ports
associativity	NOA
replacement policy	CPU-to-cache bus width
read miss latency	cache-to-memory bus width
write miss latency	word size
return policy (requested word first/first subblock first)	

Table 1: Inputs used to configure the LE cache model simulator

A more detailed picture of the interaction between LE and RCM_brisc is shown in Figure 17:

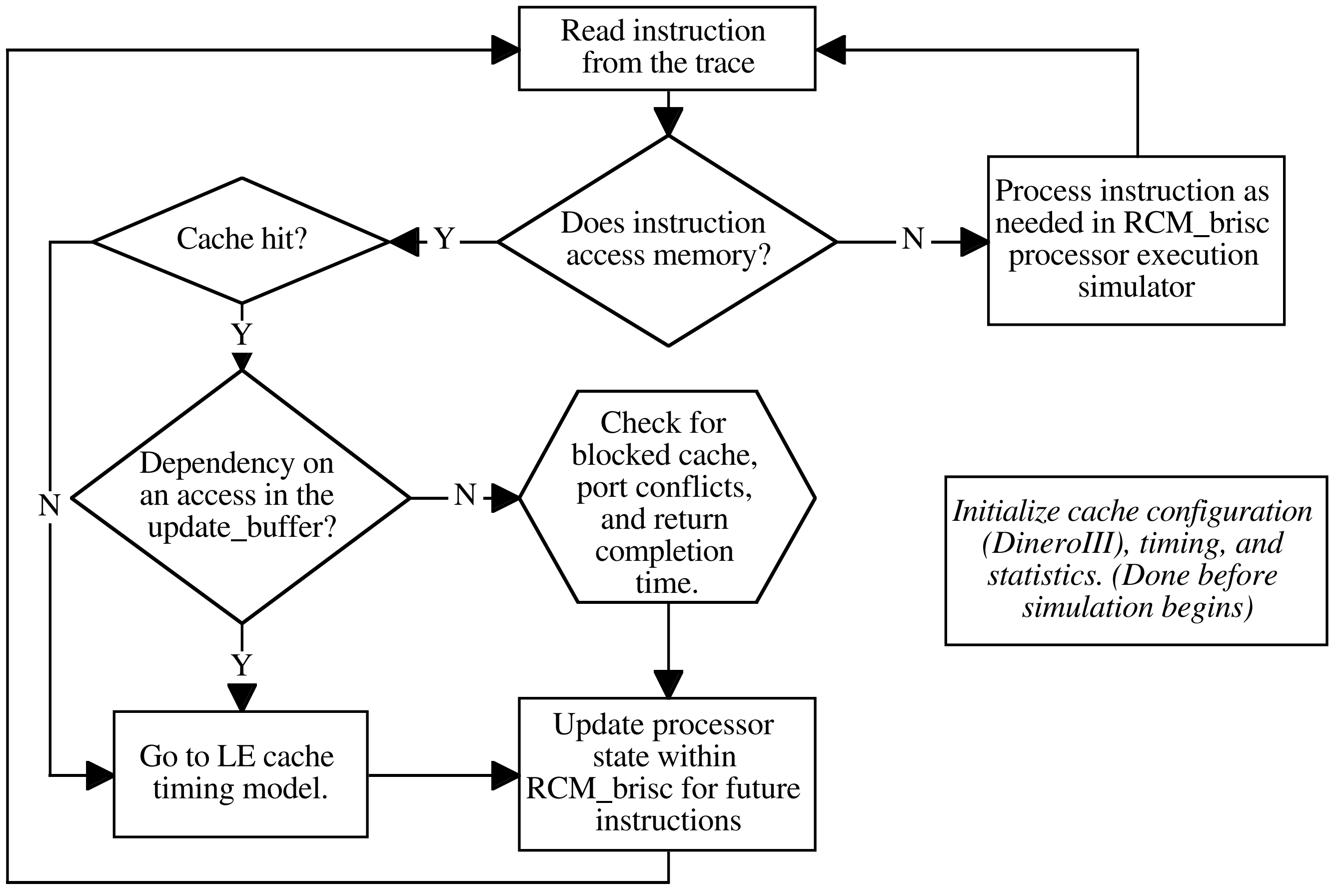

Figure 17: Interaction between the processor simulator, RCM_brisc, and LE

There is a potential situation in which desired data is located in the DineroIII cache state when a memory access commences, but an outstanding access in the `update_buffer` will modify or replace that block. If the new access does not know about the pending update to that data, it will proceed as if it were a cache hit and receive old data and/or be assigned an incorrect latency! To address this problem, even if an access has been determined to hit in the cache, the `update_buffer` is checked for conflicts. If a conflict is found, an earlier, outstanding access must first update the referenced cache block, and this new access must wait until these changes are made. Thus, if the outstanding access is a miss that replaces this block, this access must then be evaluated as a cache miss, since the desired data will not be present in the cache after the earlier access completes. If, on the other hand, the outstanding access modifies the desired block, this access must then be a delayed hit.

In order to determine the correctness of the model, three approaches were used. First, a comparison of the reported execution times of a set of representative memory accessing kernels was made between the combined simulator output and the actual execution time of those kernels on the target modeled machine; these results are presented in Section 5.2. The second means of verification, presented in Section 5.3, compares the simulator output to the expected behavior for those programs. The final check for the correctness of the tool was to vary the parameters taken by the tool to see if the resulting program execution behavior was as expected. These results are presented in Section 5.4.

Since an RS/6000-like processor was chosen as the target machine, an RS/6000's cache was examined to determine its characteristics for input to the LE model.

5.1 Characterizing the RS/6000 cache

In order to accurately model the cache performance of the machine we were emulating, we measured the latencies for different cache access patterns. The memory subsystem performance description provided by the vendor was too simple to accurately represent the machine's real performance. Thus, the load/store kernel cache characterization method presented in [Shih92][Shih93] was used to learn more about the target cache's characteristics.

The load/store kernels used were similar to those used in the original study. The basic program used was:

```
double *x(array_size)
:
start_time = wtime();
for(i=1;i<loops;i++)
  (load/store kernel);
end_time = wtime();
```

Figure 18: Basic cache parameterization program

The kernel was run many times to obtain steady state performance. An example load kernel is:

```
for(i=1;i<array_size;i+=stride*4)
  p += x(i) * x(i+stride);
  p += x(i+stride*2) * x(i+stride*3);
```

Figure 19: Example load kernel

By varying the parameters `array_size` and `stride`, enough data was obtained to characterize the cache.

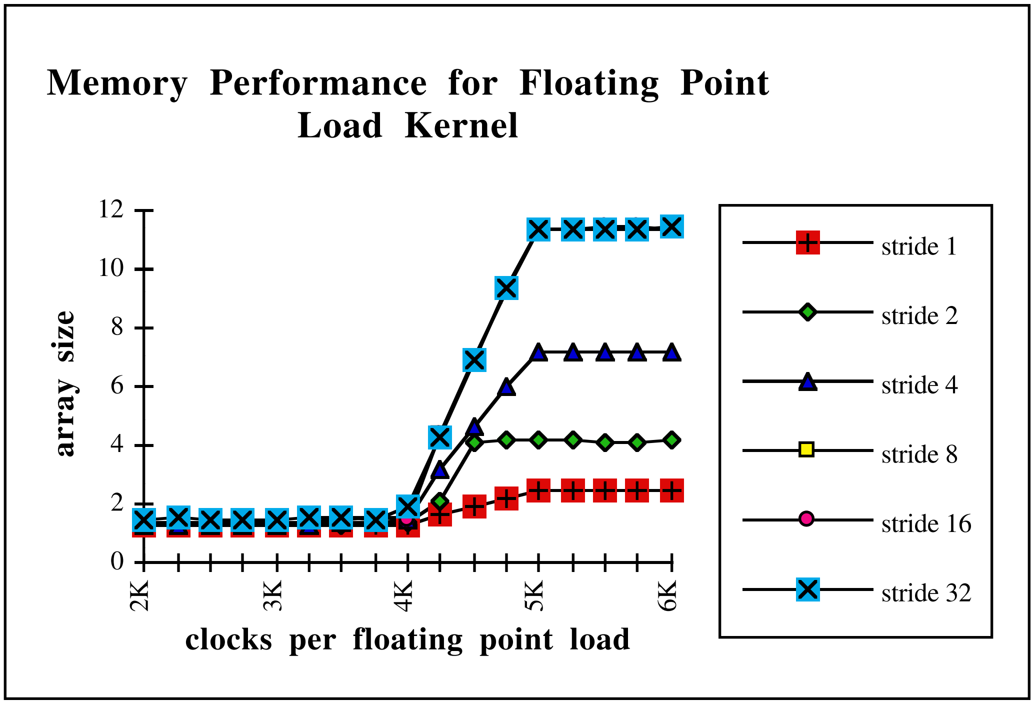

Figure 20: Load kernel performance

Figure 20 shows the average time to perform a floating point load for a given `array_size` and `stride`. When `array_size` is less than 4K, the average access rate is one clock cycle per data element, so we conclude that all accesses are cache hits (in the *cache* region [Gallivan90]). The RS/6000 Model 320H being parameterized in fact has a 32 Kbyte data cache (= 4K * 8 bytes).

When the array size increases beyond 4K, some misses will occur (in the *transition* region [Gallivan90]). If `array_size` exceeds the cache size by less than the size of one full cache column (one block of each associative set), some of the data will never be replaced. The access times level off at `array_size` = 5K (the *memory* region [Gallivan90]) for each stride used, so we conclude that a column is (5K - 4K)* 8 bytes = 8 K and that the cache is therefore 32K/8K = 4-way set associative, as documented [Hardell90].

The average miss penalty is also dependent upon the stride used to access the array. Since stride 8, 16, and 32 all have the same performance, the cache block size must be 8 * 8 = 64 bytes, as documented for the Model 320H. From the stride 8 curve, we also conclude that cache misses due to consecutive loads can be serviced at a maximum rate of one miss per 11.4 clocks.

The bus width between the cache and the next level of memory can also be determined from this data, as in [Mangionne-Smith93]; the cache-to-memory bus width for this machine is 8 bytes.

Using the following store kernel, we can obtain the store performance of the cache:

```
for(i=1;i<array_size;i+=stride*4)
   x(i) = p;
   x(i+stride*1) = p;
   x(i+stride*2) = p;
   x(i+stride*3) = p;
```

Figure 21: Sample store kernel

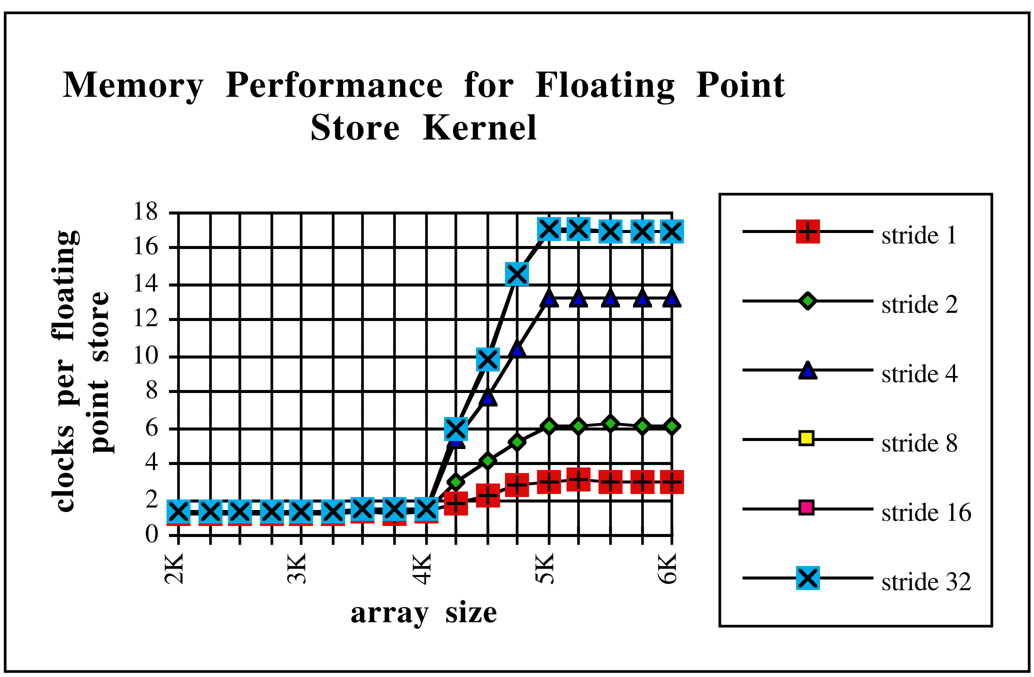

Figure 22: Store kernel performance

Figure 22 shows the cache performance for the store kernel. For consecutive store float instructions in the cache region, the average access time is 1.25 clocks per data element. The reason that the machine does not obtain 1 store per cycle is discussed in [Shih92]. From this figure, we can draw the same conclusions about the cache configuration as we did using the load kernel: the cache is 4-way set associative with 64 byte blocks and a 32 Kbyte capacity. Also, the maximum rate of servicing consecutive stores is one miss per 17.1 clocks.

The cache's configuration was verified using the load/store kernel approach; these parameters were used to configure the DineroIII base cache simulator. The full load and store miss latencies obtained from the load/store kernels were used to characterize the impact of every cache miss of the corresponding type. While this generalization may produce overly pessimistic memory performance evaluations (as all misses seem to incur the full latency penalty), the numbers obtained would only be suspect if the access patterns were not also taken into account. Due to the trailing edge effects model incorporated into the LE cache simulator, we will see that this simplification does not adversely affect a program's execution time estimate.

Table 2 summarizes the cache characteristics of the RS/6000 Model 320H; these characteristics were obtained via the load/store kernels, as described above, and literature detailing the RS/6000's cache [Hardell90][Olsson90]. The RS/6000 has one read port and one write port to the cache, and it is a write back, write allocate configuration, as described in [Olsson90]. The RS/6000 cache uses a cache reload buffer (CRB) and a store data buffer (SDB) to buffer the newly requested block and the evicted block, respectively [Hardell90]. This leads to an NOA value of 1, i.e. a hit-under-miss cache where a miss access that occurs while an earlier miss is outstanding will stall the cache. Note furthermore that accesses to the SDB while the CRB is being filled from the next level of memory can still proceed as if they were hits to the cache. Similarly, accesses to the CRB can be satisfied as the requested data returns from the next level of memory.

Size:	32K
Associativity:	4-way
Blocksize:	64B
CPU-to-cache bus width:	8B
Cache-to-memory bus width:	8B
Ports:	1R, 1W
NOA:	1
Read miss latency:	11 cycles
Write miss latency:	17 cycles

Table 2: Cache characteristics used to model the RS/6000 Model 320H

The read and write miss latency were rounded down to the nearest integer value for use in RCM_brisc+LE. Integer latencies are desired, as each access requires an integer number of cycles to execute; if fractional, average values were used, the tool, when combined with RCM_brisc, would not produce correct results. The latencies were rounded down to give more optimistic cache performance numbers; since the load/store kernels from whence this data was derived tend to represent the worst-case access patterns, rounding the latencies down may be reasonable for programs with more common memory accessing patterns.

5.2 Comparing the simulator output to actual machine performance

Since the load/store kernels were used to characterize the machine's cache, it made sense to determine the accuracy of the LE cache model by simulating these same kernels. The load/store kernels exercise a broad range of regular access patterns that could be seen by the cache. If the simulator is accurate for these kernels, the accuracy should carry over as well to workloads that are less stressful of the memory.

To gauge the usefulness of the LE cache simulator vs. other, currently available behavioral cache simulators, three other simulators were also run with the same inputs:

perfect cache: In this simulator, a perfect cache is assumed. This is the RCM_brisc processor simulator running unmodified.

adjusted: In this simulator, DineroIII is used to determine the number of cache hits and misses experienced during program execution. However, the simulator itself does not change the latencies of memory accesses on the fly. Instead, after the program execution time is determined using the RCM_brisc processor simulator, the cycles due to cache misses are simply added on to that program execution time to obtain an adjusted, cache miss latency-incorporating simulation output. The number of cycles due to cache misses is determined by multiplying the number of read misses and write misses by the corresponding nominal miss latency. The sum of these numbers is then added to the program execution time to obtain the final, adjusted program execution time.

LE-nominal: This simulator determines the effect of each memory access on program execution as it is simulated. DineroIII is queried on the fly to determine if the memory access is a cache hit or miss and the corresponding nominal latency is assigned to the access. Additional latency-adding effects are ignored. The difference between this simulator and the adjusted simulator is that this simulator accounts for the masking, if any, of some of the total memory latency by activity or stalls for other reasons in the processor. This simulator handles this overlap between memory access latencies and other processor activity more realistically than the adjusted simulator does.

LE cache: This is the fully implemented LE cache model simulator that has been developed in this paper, including all the additional latency-adding effects.

For reference, *actual* numbers were obtained by timing the execution of each kernel on an actual RS/6000 Model 320H; the lowest time in five trials was used in an attempt to minimize extraneous unmodeled interference due to the operating system and/or the network.

In the rest of Section 5.0, the LE cache model results are compared with the results of these three alternatives and the actual run time. Section 5.2.1 shows the results of simulating the load/store kernels on the three simulators, while Section 5.2.2 presents the simulation results for the first nine Livermore Loop Kernels [McMahon86].

5.2.1 Load/store kernel simulation performance

We traced the execution of each load and store kernel running on an IBM RS/6000 Model 320H using the *Atrace* tracing package developed by Ravi Nair of the IBM T.J. Watson Research Lab. The output of *Atrace* was converted into a form readable by the RCM_brisc tool, e.g. one where instruction semantics, register requirements, memory addresses, etc. were understood by RCM_brisc. Each of the converted traces was then input into each simulator along with the cache parameters listed in Table 1.

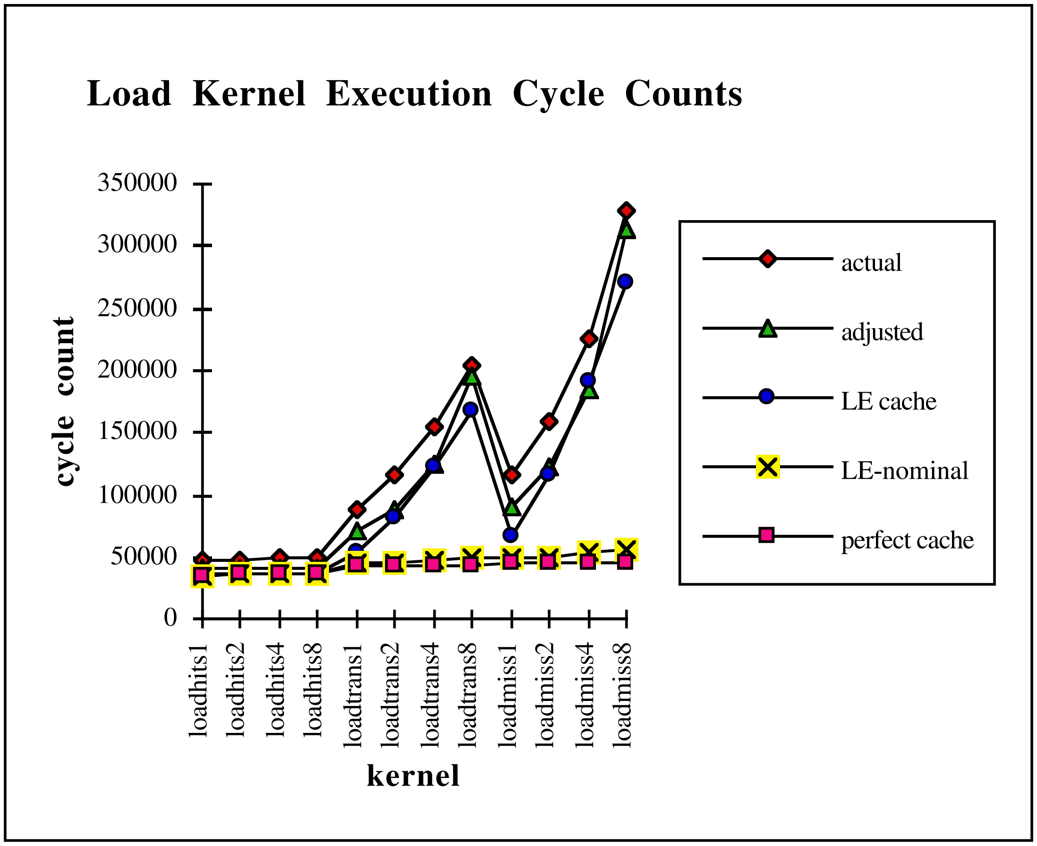

Figure 23: Execution times of each of the load subkernels as simulated by RCM_brisc with the 4 cache models and as measured on an actual RS/6000

The reported execution cycle counts of the load and store kernels is shown in Figures 23 and 24, respectively. The load/store kernels from Section 5.1 were run with strides of 1, 2, 4, and 8 and with one array size each in the cache, transition, and memory regions of cache activity. The kernels with an array in the cache, transition, and memory regions are denoted by the “hit,” trans,” and “miss” kernels, respectively, in these figures; the 1, 2, 4, and 8 in the kernel names indicate the stride. Each subkernel performs exactly the same number of memory accesses (after the arrays are initialized).

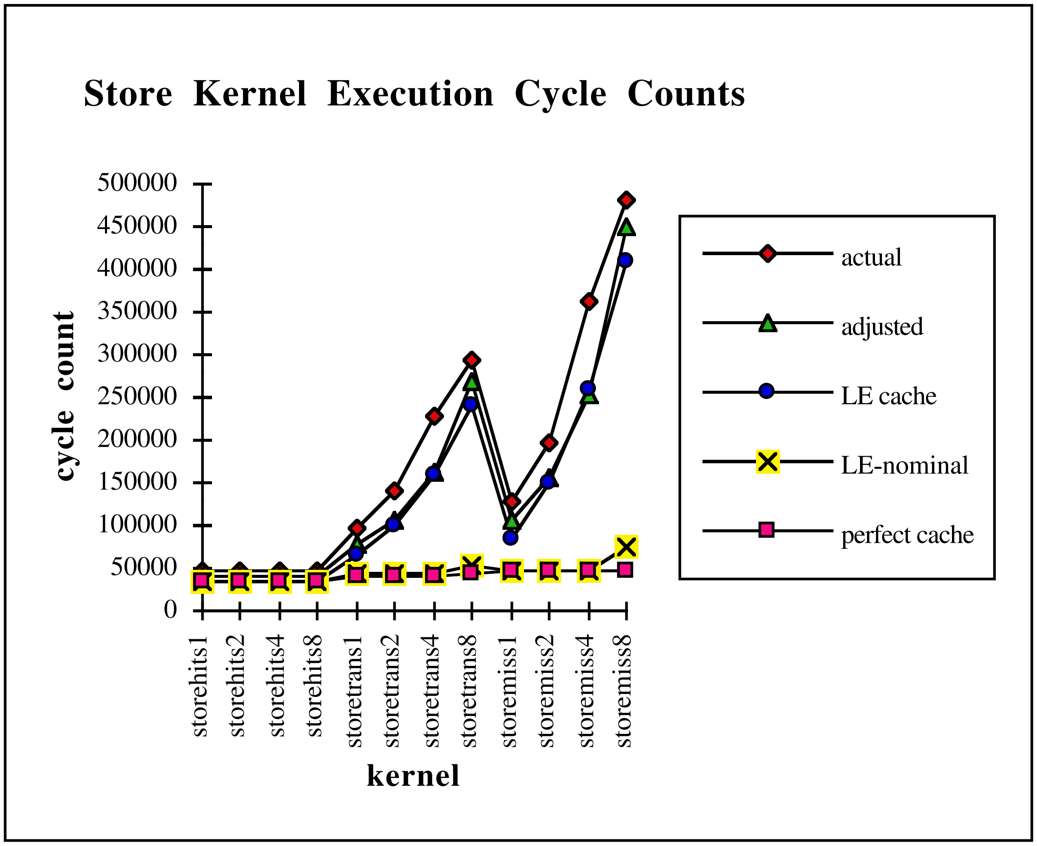

Figure 24: Execution times of each of the store subkernels as simulated by RCM_brisc with the 4 cache models and as measured on an actual RS/6000

The performance of the four simulators was similar for both the load and store kernels, so the following discussion will relate to the load kernels only. Two types of execution cycle count curves are seen in the graphs above. One type is created by the perfect cache and LE-nominal simulators. For these simulators, the reported execution times remain fairly constant across all of the kernels. This makes sense for the perfect cache simulator, as the same number of memory accesses is performed in each of the kernels and all accesses are hits. As a result, with a similar number of compute instructions in each kernel, the overall reported execution time should remain fairly constant.

The fact that the LE-nominal simulator also follows this trend is interesting, though not surprising. The latencies that are incorporated in this simulator do not simply add to the overall execution time, as many of the memory accesses overlap. Since DineroIII is maintaining the cache state with no deferred updates, once an access is made, the cache state is updated to indicate that that block is resident in the cache. Future accesses to that cache block (until it is replaced) are deemed to be hits to the cache, regardless of when they are actually issued. As a result, we see many hit-under-miss conditions that do not actually exist, but due to the simplicity of the model, are present in the program simulation. Note that there are fewer such hit-under-miss opportunities as the stride increases in the trans and miss regions, and the LE-nominal curve rises above the perfect curve accordingly. Thus, the overall reported execution time does not differ much from the perfect cache times, but the difference does increase for the larger strides in the trans and miss regions.

The other type of curve present in the graphs above follows the actual execution times quite well. The adjusted and the LE cache simulators both fall in this category. It is interesting that the adjusted model performs so well compared to both the LE cache model and the actual machine performance. This is an artifact of the kernels that were used in this experiment. These kernels stress memory to the extent that they are completely memory bound. Thus, virtually every access' latency will directly affect overall program execution time. Adding the effects of each of the memory access' to the perfect cache simulator's reported time thus gives a reasonable picture of program execution. This approach works for these kernels, where the memory accesses' latencies are not masked out by processor execution. However, we shall see in Section 5.2.2 that, for more common, non-memory stressful codes where there is ample opportunity for latency masking, this simplification of simply adjusting the processor's total run time adversely affects simulated execution times, and produces an output that tends to overestimate the machine run time relative to the LE cache model.

The LE cache model also produces program execution cycle counts which closely mirror the performance of those codes running on an actual machine. While the LE cache simulator cycle counts are sometimes lower than those reported by the adjusted simulator, the gap between the LE cache model output and the actual numbers is more consistent. Each of the changes in the LE execution time are easily explainable from the graph, and they correspond directly to the cache characterization graphs presented in Section 5.1. Within each kernel, the stride is varied between 1, 2, 4, and 8. As the stride is increased, the execution time must increase because there are more misses. If each access is to a doubleword (8B) and the cache has a 64B line size, a maximum of 8 consecutive accesses can hit in each block. Only the first of the consecutive accesses to the same block can be a miss. Depending upon the stride used to access the array, there can be as many as 7 subsequent delayed hits to that same block (for stride = 1) or as few as 0 (for stride = 8) for a cache-to-memory bus width of 8B. For a set number of overall accesses, fewer delayed hits results in a greater number of cache misses, leading directly to longer execution times. Looking at the trends within each kernel (`loadhit`, `loadtrans`, `storemiss`, etc.), we see that the execution times reported by the LE cache model and the adjusted simulator do indeed increase in the trans and miss regions, as expected.

The trend of execution times among the kernels is also as we should expect. The kernels in the hit (cache) region of memory have lower overall execution times than the comparable kernels (e.g. those with equivalent strides) in the transition region (denoted as the `<load/store>trans` kernels). The transition region kernels in turn have lower execution times than the comparable kernels performed in the miss region (denoted by the `<load/store>miss` kernels). These results correlate directly to the graphs characterizing the cache that were presented in Section 5.1 – each successive region incurs a higher average miss penalty per access.

We can see from Figure 23 that, while the execution cycle counts reported by the LE cache simulator are lower than the actual execution times, the shape of the curve across all the kernels closely follows the same curve for the actual execution times. The simulator's execution times are lower than the actual execution times because the simulator we have built is not nearly detailed enough to accurately capture all of the effects seen during program execution on an actual RS/6000 machine. For example, a TLB was not modeled; calculations for these kernels indicate that TLB misses could contribute up to 3,000 additional cycles to the overall program execution time. Another factor that would affect the overall execution time is the presence of system interrupts during program execution. While we attempted to minimize the effects of extraneous interrupts (by using the lowest run time obtained among five runs), some interrupts do inevitably occur. System interrupts can upset the cache state and cause register spillage, both of which can lead to increased actual program execution time being reflected in the user run time. Due to these simplifications, execution times will generally underestimate the run times on an actual machine. The adjusted model may thus appear to be more accurate, although this is most likely only a spurious coincidence since the additional cycles of the adjusted model over the LE cache model occur for the wrong reason.

Note that early design cycle simulators are not used to determine the exact performance of a desired final implementation in a total system context; they are used instead to obtain an accurate idea of machine performance on a target application while providing flexibility early in the design process. In this regard, the RCM_brisc+LE simulator is most appropriate. The fact that the LE cache model performance follows the shape of the actual performance curves for these kernels attests to the LE cache model's accuracy with regards to the additional effects that it incorporates. From these arguments and further evidence in the following sections, we conclude that the LE cache model presents a more realistic performance estimate than the other three simulators while providing designers with much more useful information with which to make design decisions.

5.2.2 The Livermore Loop Kernels

In order to determine the performance of RCM_brisc+LE on more typical, less memory stressful code, we chose to simulate the performance of a subset of the Livermore Loop Kernels, specifically `lfk1` - `lfk9`. These programs were traced on an RS/6000 Model 320H as were the load/store kernels in Section 5.2.1.

	actual (cycles)	*perfect cache*	*adjusted*	*LE-nominal*	*LE cache*
lfk1	5261491	-1.35%	7.54%	-0.09%	0.99%
lfk2	5070502	2.40%	11.61%	3.70%	4.82%
lfk3	4832365	-9.09%	0.57%	-7.72%	-6.55%
lfk4	4605971	-6.84%	3.29%	-5.41%	-4.18%
lfk5	5073453	-1.83%	7.39%	-0.52%	0.59%
lfk6	44452627	0.96%	2.43%	1.34%	1.43%
lfk7	5608557	-0.33%	8.01%	-0.18%	0.83%
lfk8	12765638	-6.46%	-2.80%	-7.49%	-7.05%
lfk9	6707654	-3.20%	3.79%	-2.20%	-1.34%

Table 3: Comparison of relative execution times of each of the Livermore Loop Kernels as simulated by the four simulators vs. the actual execution times

We can see that even though our simulator is very high level and is intended for early design cycle use, the reported execution cycle times from the RCM_brisc+LE simulator differ from the actual execution times by a maximum of 7.05% for these benchmarks. Other simulators which are much more detailed have reported success when their simulation output has come within 32% of actual execution times [Lebeck95]. Considering the fact that this simulator is supposed to give a good picture of program/processor performance and not the actual runtimes, the results look very good. With the additional statistics output by the LE cache model (see Section 7.0), along with those already output by the RCM_brisc processor simulator (e.g. basic block size, CPI, functional unit usage, etc.), the RCM_brisc+LE simulator adequately serves its purpose.

Interestingly, we see that not all of the simulator predicted execution times underestimate the actual run times – in `lfk2` the LE cache simulator overestimated the run time by nearly 5%. Still, this performance is quite acceptable given the intended use of the simulator in the early design cycle. This performance is particularly impressive since it has been asserted that industrial processor simulators rarely accurately model the final product [Bose96]. Having a highly configurable, semantically accurate simulator can be of great use in deciding what parameter changes to investigate early in the design cycle while they are not yet fixed in stone.

As expected, the perfect cache simulation results normally underestimate the program execution time (7 out of 9 kernels). Using the perfect cache simplification yields execution times that report execution times that are up to 9.09% lower than actual.

The adjusted simulator, on the other hand, normally *overestimates* program execution times (8 out of 9 kernels). As discussed earlier, this is due to the fact that many memory access latencies are at least partially masked by processor activity and therefore do not directly add to overall program execution time. As a result, the outputs of the adjusted simulator consistently show significantly more cycles than the LE cache simulator (up to 7.8% more), for reasons that are inappropriate in the actual machine. Using the adjusted simulator yields reported execution times that are up to 11.61% higher than actual.

The LE-nominal simulator performs similarly to the LE cache model simulator for these codes. However, as we saw in Section 5.2.1, for more memory stressful codes, the LE-nominal simulator gives poor simulation results. The only simulator that performs well in both environments (memory stressful and non-memory stressful) is the LE cache model. Because of its flexibility and proper behavior, the LE cache model is the most attractive of these four models.

5.3 Accuracy of the output

To help determine whether the LE cache simulation output was indeed correct and not a fortunate fluke, we examined the output of the load/store kernels in detail (a description of the full output provided by the LE cache simulator is presented in Section 7.0). This could easily be done because we controlled the number of memory accesses that occurred in each of these programs. Furthermore, knowing the access patterns and the cache configuration, we can determine the number of cache hits, misses, and delayed hits we would expect to see if the program were run on the target cache.

All of the load/store kernels performed the same number of memory accesses, as stated in Section 5.2.1. To be exact, there are 23,040 memory references after the memory array is initialized. The size of the array changes to control the memory region which is accessed. Small arrays (< 4K in size) serve to stress the hit (cache) region, arrays greater than 4K and less than 5K service the transition region, and arrays greater than 5K model the miss region. The arrays sizes we used were 2,034, 4,608, and 5,760 doublewords for the hit, transition, and miss regions, respectively.

Analyzing `loadtranstride4`, we know that there must be 4,608 write accesses to initialize the memory array. The initialization process strides through the array by 1, so we expect to find 4,608/8 = 576 compulsory cache misses to load the array into the cache. Furthermore, since each write miss brings in a cache line consisting of 8 doublewords, we expect to see 7 delayed write hits for each cache write miss. Totaling the two numbers gives 576*7 = 4,032 total delayed write hits. This corresponds closely to the 4091 delayed write hits reported in the output of the `loadtranstride4` simulation. The differences are due to two main factors. The first results from the compiler's placement of the data array in memory. For the compiler we used, `cc` on the RS/6000, the array was allocated in memory on a word boundary, but not on a cache block boundary. As a result, the first access to the data array did not bring in a full block's worth of useful data. For all of the kernels whose expected and simulated delayed hit numbers differ by exactly one, the first access to the array was to the second word in the cache block. This resulted in a maximum of 6 delayed hits for the initial cache block access, which is one less than what we would expect for a block-aligned array. This placement of the array in memory results in an additional cache miss to access the last data word in the array and consequently reduces the number of delayed write hits by one. Thus, this difference between the expected and simulated numbers is due to the fact that the compiler did not force the array to have a block-aligned memory placement.

The other, larger gaps between the expected and simulated delayed hit numbers (for both read and write) are due to the RS/6000's method of handling data variables. Despite allocating variables in registers, the RS/6000 always keeps copies of those variables in memory in preparation for context switches, etc. When the working set of the program is small, these variables remain in the cache and the registers and no additional misses of any kind are seen. However, when the cache is filled with data accesses in a program, as in the load/store kernels in the transition and miss regions, these variables are flushed from the cache. When it comes time to update these variables, the RS/6000 reloads them into the cache before writing to them. These status variables are updated in the load/store kernels in every loop iteration, resulting in larger differences between the expected and simulated delayed hit numbers.

The number of read misses and delayed hits can also be explained. For an array size of 4,608, accessed using a stride of 4, we get one delayed hit for each cache miss, since two accesses will hit in a cache line size of 8 doublewords. The size of one cache set in the RS/6000 is 4,096 doublewords, which means there will be 4,608 - 4,096 = 512 conflict misses seen when accessing the entire memory array. Striding through the array by 4 results in 512/2 = 256 misses for each time the array is completely accessed, since two doublewords from each cache block are actually used. Each time those 512 doublewords are brought in to the cache, they replace 512 doublewords that will be needed later to access another portion of the array. The cache is 4-way set associative, which means there will be 5 groups of 512 double words that are displaced each time the array is accessed. (This is true even for the first access to the array after initialization, because the initialization causes the same displacement). Since the outer loop causes the array to be accessed 5 times, we would thus expect to find 256x5x5 = 6,400 delayed read hits, which follows closely to the 6,488 delayed read hits reported in the program output. The difference between these two numbers is due to the additional cache accesses used by the RS/6000 to update variables, as explained above.

The same analysis was performed on each of the load/store kernels tested, giving similar results – the reported numbers follow the expected numbers very well, as seen from Table 3. Thus, the simulation of delayed hit read and write accesses appears to be modeled accurately subject to the differences explained above. The correctness of the other portions of the LE cache model, e.g. port conflicts, bus width considerations, and NOA, are discussed in the following section.

kernel	*expected # of read delayed hits*	*# of read delayed hits in sim*	*expected # of write delayed hits*	*# of write delayed hits in sim*
loadhits1	0	1	2016	2015
loadhits2	0	1	2016	2015
loadhits4	0	1	2016	2015
loadhits8	0	1	2016	2015
loadtrans1	11200	11479	4032	4046
loadtrans2	9600	9834	4032	4061
loadtrans4	6400	6488	4032	4091
loadtrans8	0	4	4032	4151
loadmiss1	20160	20160	5040	5051
loadmiss2	17280	17276	5040	5063
loadmiss4	11520	11524	5040	5087
loadmiss8	0	4	5040	5135
storehits1	0	1	2016	2015
storehits2	0	1	2016	2015
storehits4	0	1	2016	2015
storehits8	0	1	2016	2015
storetrans1	0	1	15232	15394
storetrans2	0	1	13632	13753
storetrans4	0	1	10432	10439
storetrans8	0	1	4032	4031
storemiss1	0	1	25200	25195
storemiss2	0	1	22320	22311
storemiss4	0	1	16560	16559
storemiss8	0	1	5040	5039

Table 4: Comparing the expected number of delayed hits with the number of delayed hits seen from program simulation

5.4 Analyzing the flexibility of the LE cache model

One of the strong points of the LE cache model is its flexibility. Its ability to model caches by varying parameters including and beyond those parameters already modeled by existing behavioral cache simulators is what makes this model unique. Along with the default cache size, associativity, block size, and other flexibility provided by the underlying DineroIII cache simulator, the LE cache model adds parameters for read and write miss latencies, the number of ports to the cache, the width of the busses to and from the cache, and the number of outstanding accesses allowed before the cache blocks. Each of these additional parameters was varied for a specified kernel, `loadtranstride4`, with the RS/6000 cache configuration, to determine two things: 1) whether the effects were modeled reasonably and 2) the effect that changing these parameters has on program execution.

5.4.1 Varying latencies

Changing the miss latency input for a memory access should have the obvious effect on program execution time – decreasing the latency should decrease program execution time. In `loadtranstride4`, the interesting part of the kernel is during the reads of the memory array (the other memory accessing portion of the program initializes the data array). The load miss latency determined for the RS/6000 cache in Section 5.1 was 11 cycles. Running the simulation multiple times with decreasing load miss latencies resulted in the following graph:

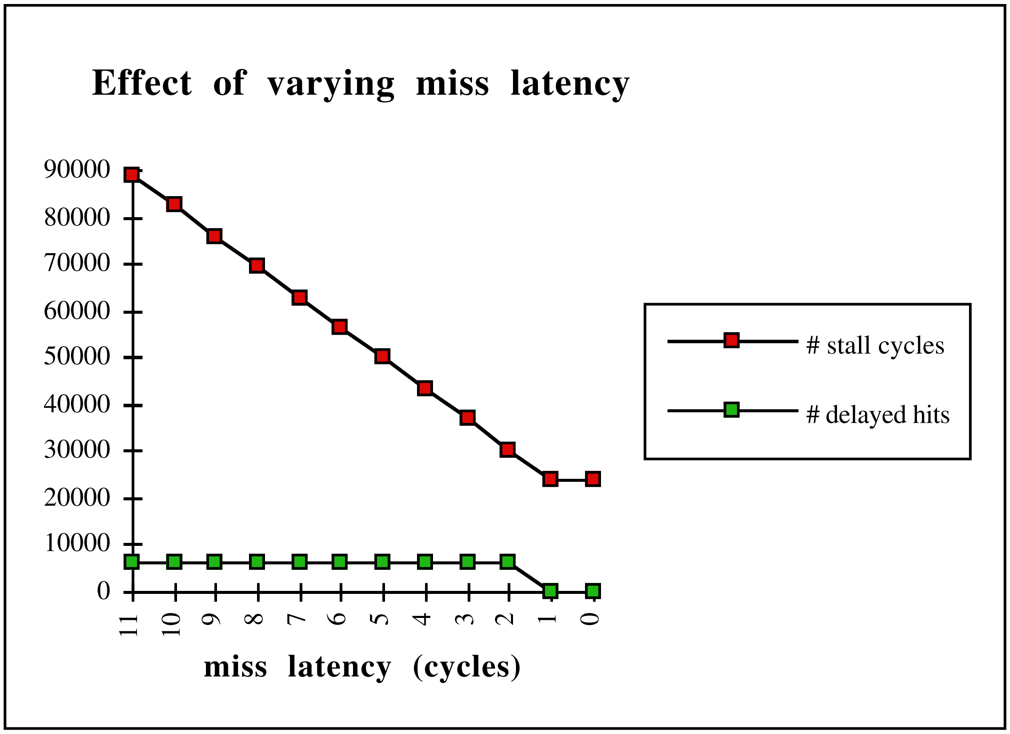

Figure 25: The effect of varying the load miss latency on the number of stall cycles due to trailing edge effects and the number of read delayed hits for the kernel `loadtranstride4`

As we can see from Figure 24, the number of stall cycles due to trailing edge effects decreases linearly with the decrease in miss latency, as expected. Also, the number of delayed hits is 6,488 for all latencies except 1 and 0. This makes sense given the RS/6000's cache configuration (whose other parameters were not changed in this experiment). Given the same number of delayed hits, a lower miss latency will result in a lower number of stall cycles. At a latency of 1 cycle, each cache read miss takes only a single cycle longer to execute than a cache hit, which has an effective latency of one cycle (requested data resident in the cache is returned the same cycle it is requested). At this latency, 19 delayed read hits occur. (These delayed hits are due to the additional variable accesses, as discussed in Section 5.3.) Reducing the latency to 0 cycles resulted in no delayed read hits, as expected, since all read accesses are as fast as cache hits. This is true regardless of the bus width, as long as the bus between the cache and next level of memory is wide enough to return one access per cycle. Notice that the number of stall cycles remains constant for both read miss latencies of 1 and 0 at approximately 24,000 cycles. The reason the number of stall cycles is not zero is that the write miss latency was left at its original value of 17; at read miss latencies of 1 and 0, all of the stall cycles due to trailing edge effects are caused by delayed write hits that occur during the initialization of the data array.

5.4.2 Varying the number of ports

Varying the number of ports available to the cache is also a useful feature for simulation, as ports are costly, but having only a single cache port may lead to poor performance. The number of cache ports directly affects the number of stall cycles due to port conflicts. Increasing the number of ports has the obvious effect of decreasing the number of these stall cycles, potentially decreasing the overall program execution time.

However, if only one access returns to the cache from the next level of memory each cycle, increasing the number of cache ports may have little effect on misses and delayed hits– the bottleneck in this case is the cache-to-memory bus width, not the number of available ports. Port conflicts will only occur when a greater number of requests can complete in a cycle than there are available cache ports; if the number of words that can be returned from memory each cycle is less than the number of cache ports, increasing the number of ports will only benefit cache hits.

We simulated the `loadtranstride2` kernel, varying the number of read ports and the width of the bus between the cache and the next level of memory. (`loadtranstride2` was chosen over `loadtranstride4` because it placed greater stress on the availability of cache ports). The bus between the CPU and the cache was assumed to be wide enough to return one access per cycle, with one bus dedicated to each of the available cache ports. As in Section 5.4.1, only the number of read ports was varied, as the interesting portion of the kernel involved only loads.

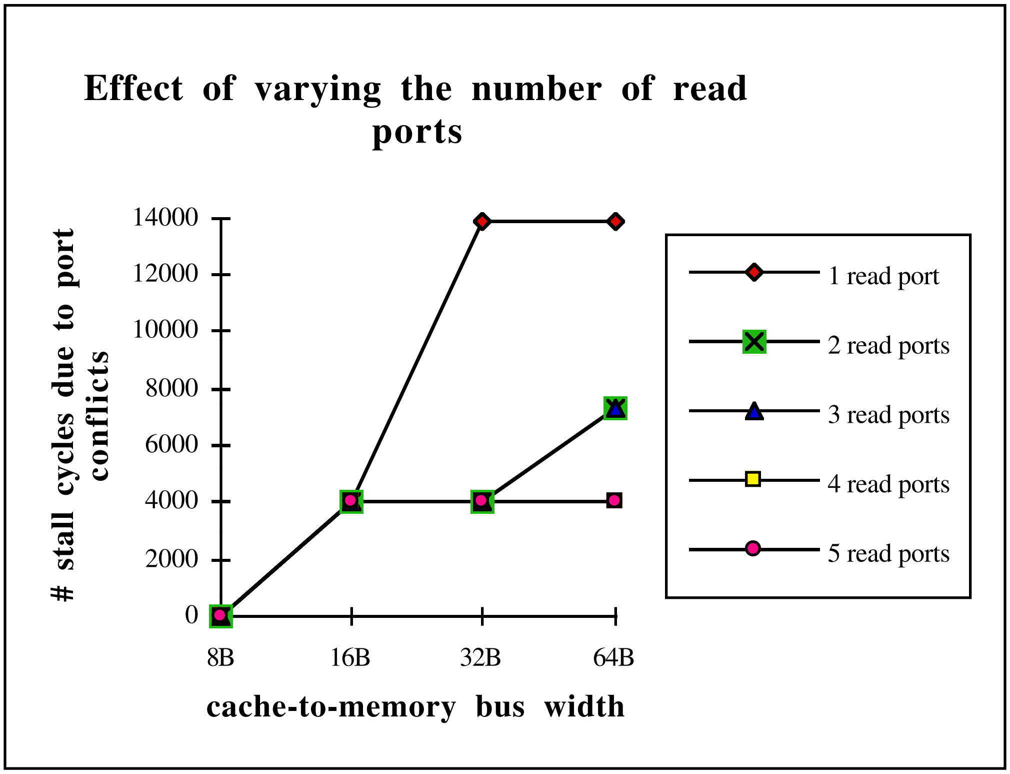

Figure 26: The effect of varying the number of read ports and the cache-to-memory bus width on the number of stall cycles due to port conflicts for kernel `loadtranstride2`

As we can see in Figure 25, increasing the number of cache ports decreases the number of stall cycles due to port conflicts, as expected. Three trends can be seen in this graph as the cache-to-memory bus width is increased. At a bus width of 8B, only one request is returned to the cache per cycle, so only one port is needed to satisfy that request. As a result, there are no port conflicts regardless the number of ports available. At a bus width of 16B, the number of port conflicts is still the same at ~4,000 regardless of the number of ports. This is because we are striding through the array by two; consequently, there is only one doubleword requested in each group of 16B (since a doubleword is 8B). Increasing the bus width to 16B results in a constant number of port conflicts because these port conflicts are due to the initialization of the array, which strides through the array by one. We will see that 4,000 port conflicts is the minimum required by this kernel to satisfy the initialization phase. With the 16B bus width, there are no read port conflicts, as expected – only one port is required to satisfy all the returning read requests as they arrive.

At a bus width of 32B, up to two accesses can return to the cache, and thus the processor, every cycle. We therefore need at least two ports to satisfy all of the requests that are returning without incurring any port conflicts. This expectation is verified in the above graph – with only one read port available, the number of stall cycles due to port conflicts shoots up to nearly 14,000. Every access is issued in a separate cycle, e.g. the first access to the cache block is issued in cycle X, the second in cycle X+1, etc. For a bus width of 32B, data for two accesses can return to the cache each cycle after the initial miss latency is satisfied. For a miss latency of N cycles, the first two requests to the cache block (to the first and third doublewords) will return in cycle X+N. The second 32B of the cache block will return to the cache in cycle X+N+1. With a sufficient number of ports, the third access to the block would complete in cycle X+N+1. However, the second access is completing in cycle X+N+1 due to the port conflict caused by the initial access. Since there is only a single read port available, the third access must wait an additional cycle, to cycle X+N+2, to complete. The fourth access is likewise delayed, resulting in three stall cycles for every read access to a block. However, for two or more ports, the number of stall cycles due to port conflicts remains at 4,000, as expected.

At a bus width of 64B, up to four accesses can return to the cache each cycle, which means we need four or more read ports to satisfy all of the requests without stalling. Thus, at four and five ports, we see that the number of stall cycles due to port conflicts remains at 4,000. However, the performance for one, two, and three ports is interesting, and merits some discussion.

The processor we are modeling, an RS/6000, can only handle one outstanding access before the cache blocks. All subsequent accesses to the current pending block can continue, however; blocking only occurs when a new block is accessed. Striding through an array by two, we touch four of the eight doublewords in each cache block. At a bus width of 64B, all four words return in the same cycle. With a single cache port available, the first access completes without any port conflicts, but the next three accesses suffer port conflicts. Each of these stalled accesses must wait an additional cycle to complete, as there is only one available cache port. Here, for each group of four accesses, three stall cycles due to port conflicts will result. This is the worst case for this cache configuration for this program – the maximum number of accesses is returning to the cache each cycle but the minimum number are actually allowed to complete. This is also the same number of stall cycles per cache block load that was experienced for a bus width of 32B and a single port, which explains the similarity in the number of stall cycles due to port conflicts for those two bus widths.

For two ports, two of the four accesses can complete each cycle, but there will be two port conflicts for the 3rd and 4th words returning from memory. This is because with all four accesses returning in the same cycle, the first two to return will use the two available cache ports; the subsequent two in the cache block will suffer port conflicts. For each group of four read accesses, we will have two port conflicts – both of these conflicts are satisfied in the very next cycle. Thus, we incur one stall cycle due to port conflicts for each group of four read accesses.

For three ports, three of the four accesses can complete each cycle, but there will still be a single port conflict. The first three accesses to the cache block will complete using the three available cache ports, but the fourth access returning to the cache from memory will have to stall a single cycle to obtain a usable port. Thus, like the two port case, we incur one stall cycle due to port conflicts for each group of four read accesses. Consequently, the overall number of stall cycles due to port conflicts is the same for both two and three read ports, as we can see from the graph.

Each of the trends in the above graph directly follow the expected behavior of the program for the given cache configuration, which indicates that the handling of port conflicts is done correctly in the LE cache model.

5.4.3 Varying bus widths

Concurrent with the gathering of data for Section 5.4.2, we looked at the number of stall cycles due to bus conflicts.

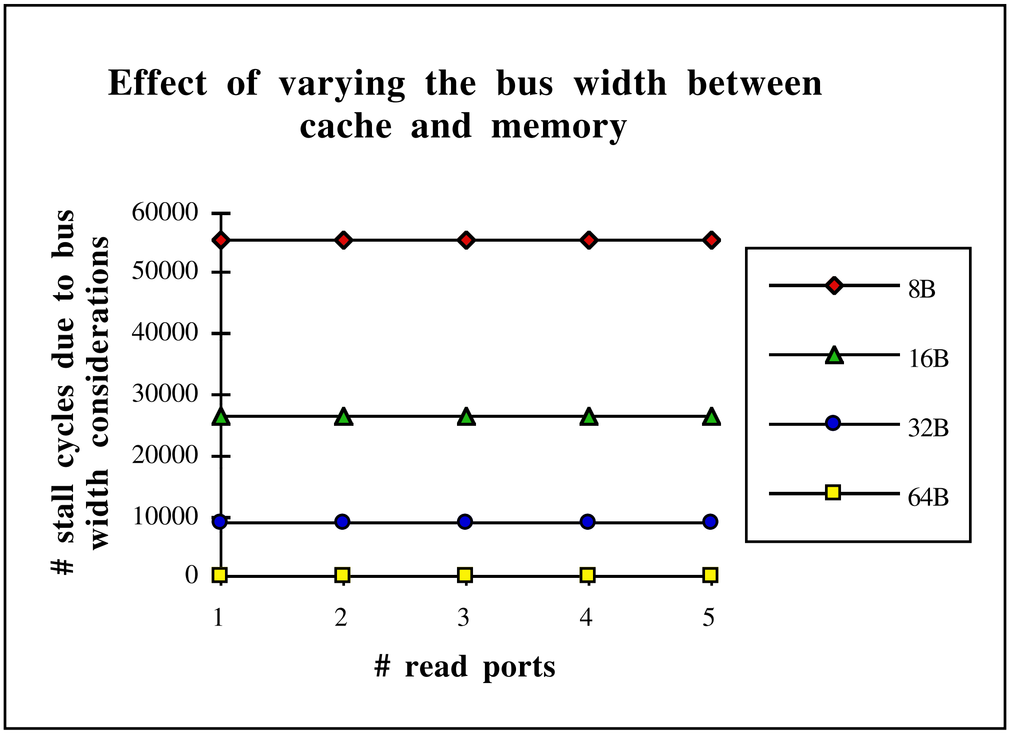

Figure 27: The effect of varying the number of read ports and the cache-to-memory bus width on the number of stall cycles due to bus width considerations for kernel `loadtranstride2`

As we can see from Figure 26, the output is as we expected – the narrower the bus width, the greater the number of stall cycles due to bus conflicts. The number of stall cycles does not decrease in inverse proportion to the bus width, however, due to the kernel we simulated. Since we are striding through the array by two, every doubleword in the cache block does not necessarily incur a stall cycle due to bus width considerations. For an 8B bus width only one access can return every cycle, so we would expect to see the greatest number of bus conflicts here. For a 16B bus, only one read access will return each cycle. However, the initialization phase strides through the array by one, so a 16B bus will halve the number of bus conflicts for the initialization. Thus, the number of stall cycles due to bus width considerations drops from ~55,000 to ~27,000 cycles.

At the next jump in bus width to 32B, two read accesses will return to the cache each cycle, effectively halving the number of read stalls due to bus width conflicts compared to the 8B and 16B bus widths. The effects on the initialization phase are also of the same magnitude, halving the number of stall cycles compared to the 16B bus width. As a result, the number of stall cycles due to bus width considerations drops to ~8,900 cycles.

Finally, at a bus width of 64B, an entire cache block is returned to the cache from the next level of memory each cycle. As a result, no stall cycles are incurred from bus width considerations because an entire block is loaded each cycle.

Each of these numbers remain constant regardless of the number of cache ports available. Changing the number of cache ports has no effect on the number of stall cycle due to bus conflicts because the bus width considerations for an access are taken into account before the number of available ports is considered.

5.4.4 Varying the number of outstanding accesses allowable

The last parameter that can be varied in the LE cache model is the number of outstanding accesses allowed before the cache stalls further accesses. This number (NOA) is equivalent to the degree of nonblocking of the cache. Caches range from fully blocking (NOA = 1) to fully non-blocking (NOA > maximum miss latency). The RS/6000 has an NOA value of 1. We simulated `loadtranstride4` with varying degrees of NOA to obtain the results shown in Figure 27:

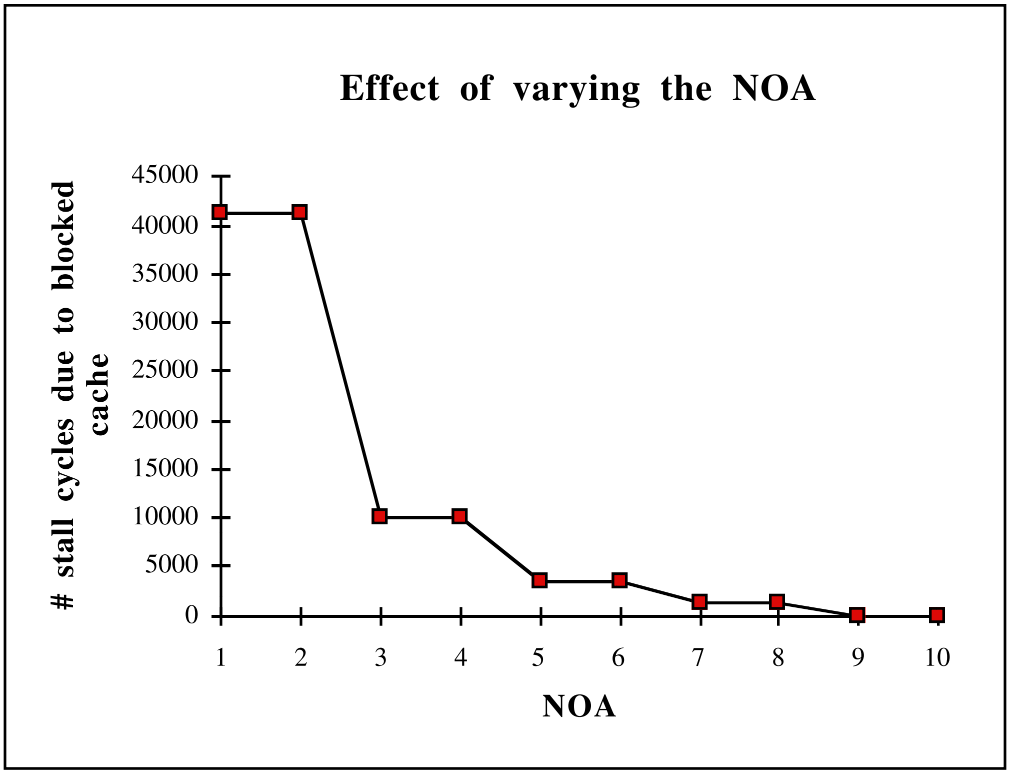

Figure 28: The effect of varying the number of outstanding accesses allowable on the number of stall cycles due to a blocked cache for kernel `loadtranstride4`

As expected, increasing the NOA decreases the number of stall cycles due to cache blocking. There is, however, an interesting staircase effect seen in these numbers. This is due to the pattern by which the memory array is accessed. As stated before, with a stride of 4 and a cache block size of 8 doublewords, two accesses are made to each cache block before moving on to the next one. The first access to a block will miss. The second access, however, is to the same pending block as the earlier outstanding miss. In this implementation of the LE cache model, delayed hits count toward the NOA; the implementation can easily be changed to have only misses count towards the NOA. As a result, the NOA must increase by two for this kernel before we see a drop in the number of stall cycles due to a blocked cache. For instance, when the NOA = 3, the first two accesses to a block can be in flight and a third access to a different block can also start. Thus, we see the staircase decline in the number of stall cycles for each increase by 2 of the NOA. The decreases in the number of stall cycles due to blocked cache as the NOA increases by two are progressively smaller and smaller. The first jump is the largest because given the access pattern, two different blocks could potentially be outstanding at a time.

Increasing the NOA to the point where the second block can begin execution dramatically decreases the number of stall cycles. However, as the NOA increases, fewer new block accesses benefit, as only a certain number of accesses can be in flight at a time given the nominal miss latency and the distance between memory accesses in the program stream. Again, this effect is what we would expect from the model when simulating the `loadtranstride4` kernel.

6.0 Execution times

Using a new model normally incurs some type of cost – some solutions are never implemented because their value added is small in comparison to the cost of use. For the LE cache model, the cost of use is the increased time required to simulate a target program compared the same time to simulate the program on a simulator lacking the LE cache model. The most dramatic comparison is between the RCM_brisc+LE simulator and the original RCM_brisc simulator, which assumed a perfect memory model and, as a result, did not incur any additional processing time when evaluating memory accessing instructions.

The greatest difference between the execution times of the two simulators is obviously for applications that have a significant fraction of instructions that access memory. For programs in this category, i.e. the load/store kernels of Section 5.2, RCM_brisc+LE is ~17% slower than the perfect cache simulator, i.e. RCM_brisc alone. For more typical, less memory stressful benchmarks such as the Livermore Loop Kernels, RCM_brisc+LE is only ~10% slower than RCM_brisc alone. For these more common codes, RCM_brisc+LE simulates ~1540 instructions per second, whereas RCM_brisc alone simulates ~1720 instructions per second. (The simulations were run on a Sun SPARCstation20 workstation with 32MB of memory).

	lfk3	***loadtranstride4***
LE vs. LE-nominal	1.003	1.058
LE vs. perfect	1.103	1.174
LE-nominal vs. perfect	1.100	1.124

Table 5: Relative simulator run time for the LE cache and LE-nominal models over RCM_brisc alone

This relative increase in simulation time is reasonable, however, given the increased accuracy of the overall simulation provided by the inclusion of the LE cache model. For an additional comparison, we compared the simulation time for the LE-nominal processor/cache simulator running the same codes. The fourth simulator, the adjusted model, runs the same speed as the LE-nominal simulator. This is because both simulators query DineroIII on the fly to determine the result of a memory access; LE-nominal incorporates the latencies of each access during simulation, while the adjusted model calculates the impact of the memory accesses at the end of simulation and adds the resulting number of cycles to the total execution cycle count. Thus, for the following discussion, all references to the LE-nominal model with regards to simulator run times apply similarly to the adjusted simulator.

As seen in Table 4, using the full LE cache model instead of simply using nominal latencies for cache hits and misses requires only a 4.5% increase in simulation time for memory-stressful codes. For more typical, non-memory stressful codes, the difference in simulation times is almost negligible, an increase of 0.3%. A processor-only simulator in today's design environments is of limited use by itself, as the CPU's interface to memory is the most likely bottleneck to the processor's overall performance. The addition of a semantically correct cache simulator such as the LE cache model is well worth the additional 10% to 17% increase in simulator run time. Even a simple, nominal latency-incorporating simulator, like the LE-nominal cache simulator, does not help the designers visualize program performance realistically, yet it increases the simulator run time almost as much as the full LE cache model. Given the detailed output provided by the LE cache model (see Section 7.0), designers can make informed decisions regarding the cache's design to improve processor/cache performance on targeted applications. These added benefits easily outweigh the minimal increase in execution time over either of the two simpler models.

7.0 LE output

When the LE cache model is used to simulate the execution of a sequence of memory accesses, a number of useful statistics are output at the end of the simulation. An example of the output is shown below.

The program that was simulated in this case was Livermore Loop Kernel 3, which calculates the inner product of a large array. The first statistics listed in the output are the number of memory accessing instructions in the program, along with the average number of cycles per memory access. For `lfk3`, this number is quite low, at about 4 cycles per access, since there are many cache hits in the program. Accesses to the cache that hit complete in the same cycle that they begin execution, so the effective cache hit latency is one cycle.

```
Number of memory accessing instructions:    796306
Average number of cycles per memory access: 3.802

The execution (in cycles) requires:
         4516043 cycles

Additional cycles due to trailing edge effects:
         273080
Additional cycles due to bus width effects:
         26256
Additional cycles due to port conflicts:
         45389
Additional cycles due to pipeline stalls:
         56798

Number of cache hits:                          714634
Number of cache misses and delayed hits:       81672
Number of port conflicts:                      45389

Number of delayed read hits:                   14638
Average number of cycles per D.H. read:        4.166
Number of delayed write hits:                  30790
Average number of cycles per D.H. write:       7.741

D.H.+miss ratio:                               0.1026
Number of cycles due to D.H.+miss:             878468
Average number of cycles per D.H.+miss:        10.756

Number of read D.H.+miss:                      39539
Number of cycles due to read D.H.+miss:        405911
Average number of cycles per read D.H.+miss:   10.266

Number of write D.H.+miss:                     42133
Number of cycles due to write D.H.+miss:       472557
Average number of cycles per write D.H.+miss:  11.216
```

Figure 29: LE simulation output for Livermore Loop Kernel 3

The next statistic is the total number of cycles required to execute the whole program. This number is obtained from the RCM_brisc portion of the simulator, as the LE portion only simulates memory accessing instructions.

Next, the effects of trailing edge effects, bus width effects, port conflicts, and cache blocking are detailed for the given program. These are the same numbers that we verified in Section 5.4. We see that there are 45,389 cycles due to port conflicts; looking further down in the LE output, we see that that is exactly the number of port conflicts that were experienced in the program. Thus, each port conflict was resolved by making the memory access wait one additional cycle.

The effects due to delayed hits are further broken down into delayed read hits and delayed write hits. The average time for each type of delayed hit is less than the corresponding full miss latency, as expected. Following the delayed hit information, the normal cache simulation characteristics are displayed. These include the D.H.+miss ratio (the fraction of memory accesses that are delayed hits or misses), the number of reads and writes that are either delayed hits or misses, and the average latencies for each. Since there are a good number of delayed hits in the execution of `lfk3`, the average latencies for read and write accesses that are not cache hits are both lower than the corresponding miss latencies (11 and 17 cycles, respectively). This also makes sense. If the statistics reported did not include the delayed hits, the average miss latency for a read and write miss would be 11 and 17 cycles, respectively, which would not give an observer any new insight into the program's execution.

Each of these statistics can be used by a computer designer or programmer to improve the performance of a given program on a particular machine. For instance, for `lfk3`, we see that a significant number of cycles are due to port conflicts. If this phenomenon is seen during the execution of other target programs, it may be beneficial to increase the number of ports to the cache to reduce the number of port conflicts, decreasing execution time and increasing the machine's performance. Also, knowledge of the number of trailing edge accesses can help the programmer lay out data and access memory in a more efficient manner so as to avoid these additional delays during program execution [Shih96].

8.0 Conclusion

A new cache model has been developed that incorporates various effects into the simulation of a sequence of memory accesses. In addition to nominal hit and cold-start/capacity/conflict misses, delayed hits are now modeled. The latency for each access is determined by examining the access based on the current state of the cache and any misses that are outstanding at the time of the new access. The Latency Effects (LE) cache model determines the effects of trailing edge effects, bus width considerations, the number of outstanding accesses allowable, and port limitations on each access to determine the number of cycles required to complete a given access.

The LE cache model was implemented on top of the DineroIII cache simulator so that a basic cache simulator did not need to be written. The LE model was realized in an approach based on the Resource Conflict Methodology proposed by J-D Wellman, permitting the model to be flexible and general-purpose. To verify the model, LE was combined with RCM_brisc, an RCM-based processor model, and configured to simulate an RS/6000-like processor. A comparison of the reported execution times of this processor/cache simulator, RCM_brisc+LE, and the actual execution times on the target machine were made to determine the correctness of the model. Our results show that while the simulated execution times are lower than the actual times, the simulator output mirrors the effects of the differences among various memory access patterns.

Further verification was done by varying the parameters which the LE cache model can take and determining their effects on program execution. For each of the four effects, the outputs of the model closely follow the expected performance of those programs running on the target, paramaterized machine.

The cost of using the LE cache model is a 10% to 17% increase in simulation time over the same RCM_brisc simulator that assumes a perfect cache. This additional simulation time is well worth the price, as the outputs of the RCM_brisc+LE simulator are much closer to reality, and thus more usable, than the RCM_brisc simulator's output alone. Furthermore, the LE model outputs many useful statistics, including the effects of each of the latency-adding effects modeled in LE. With this knowledge, the processor and cache designers and programmers of the machine can make appropriate adjustments to improve the delivered performance of target applications on the machine.

9.0 Future work

While the LE cache model greatly improves on the performance of a behavioral, early design cycle cache simulator, there are other parameters than can easily be incorporated into the model to further the usefulness of the tool. Other common or proposed cache features, such as interleaving, hardware prefetching, etc. will be examined and incorporated into future versions of the LE cache model implementation. Also, the current simulator concentrates only on a single level of cache. Even though the simulated performance is currently relatively accurate, handling multiple levels of caches, TLB effects, and the latency effects between successive levels of the memory hierarchy is another area that will be considered in the future to improve the accuracy of the model.

Since the LE cache model is implemented on top of the DineroIII behavioral cache simulator, any other work that has been built on the DineroIII platform will be easily incorporated into the LE cache simulator. An example is the nontemporal streaming (NTS) cache presented by Rivers [Rivers96]. This and other novel cache configurations can easily be incorporated into the LE simulation model, providing designers with effective performance estimates of a wider variety of target cache configurations, regardless of whether they are existing or new, novel configurations. The NTS cache is currently being incorporated into the LE cache simulator, and the effectiveness of the NTS cache when presented with a latency-incorporating model will be evaluated.

Another area of future work will be to evaluate the effects of superscalar execution on memory access patterns and cache design. This can easily be done given the RCM_brisc+LE simulator, both of which are highly paramaterizeable. Various processor/cache configurations will be evaluated to determine potential high performance processor/cache configurations. Previously, this type of performance evaluation would not be available so early in a design stage; prototypes or machine-specific simulators would normally need to be built before these comparisons could be made. However, using the RCM-based RCM_brisc processor and LE cache model simulators, these evaluations will be made simply by changing the parameters to the simulator and rerunning the simulation.

Currently, the performance of the simulator is bound by the input method. Using a trace of execution, the simulation time of the program varies directly with the length of the trace. For long traces, this simulation is tremendously time consuming. Methods of modifying the input method, such as the use of reduced or sampled traces, will be examined in the hope of decreasing simulation time by several orders of magnitude while maintaining a high correlation between simulated and actual performance. Other trace reduction methods, such as basic block analysis, profiling, etc., will also be evaluated for use in improving the simulator speed of the RCM_brisc+LE tool.

10.0 Acknowledgments

This research was partially supported by a gift from IBM. I would like to thank J-D Wellman for all his help and support in understanding the RCM model, using the RCM_brisc tool, and fixing the many bugs that arose from interfacing the two simulators. I would also like to thank Doug Ranz of IBM for his help in understanding the actual operation of the RS/6000 cache and how to obtain the best program performance using the cache. Finally, I would like to thank Jude Rivers for his thoughtful insights and encouragement during the later portions of this work.

11.0 References

[Bakoglu90] H. B. Bakoglu and T. Whiteside, "RISC System/6000 Hardware Overview," IBM RISC System/6000 Technology SA23-2619, International Business Machines Corporation, 1990. pp. 8 - 15.

[Bedichek95] R. C. Bedicheck, “Talisman: Fast and Accurate Multicomputer Simulation,” *Proceedings of the 1995 ACM SIGMETRICS Conference*, 1995.

[Bose96] Personal communication, June, 1996.

[Gallivan90] K. Gallivan et al., “Experimentally Characterizing the Behavior of Multiprocessor Memory Systems: A Case Study,” *IEEE Transactions on Software Engineering*, vol. 16, February, 1990, pp. 216-223.

[Hardell90] W. R. Hardell et al., "Data Cache and Storage Control Units," IBM RISC System/6000 Technology SA23-2619, International Business Machines Corporation, 1990. pp. 44 - 51.

[Hennessy96] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers Inc., San Francisco, California, 1996. pp. 373 - 427.

[Hill85] M. D. Hill, DineroIII Documentation, Unpublished UNIX-style Man Page, University of California, Berkeley, October 1985.

[Lebeck95] A. R. Lebeck and D. A. Wood, “Active Memory: A New Abstraction for Memory-System Simulation,” *Proceedings of the 1995 ACM SIGMETRICS Conference*, May, 1995.

[Magnusson95] P. Magnusson and B. Werner, “Efficient Memory Simulation in SimICS,” *Proceedings of the 28th Annual Simulation Symposium*, April, 1995.

[McMahon86] F. H. McMahon, "The Livermore FORTRAN Kernel: A Computer Test of the Numerical Performance Range," Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[Olsson90] B. Olsson et al., "RISC System/6000 Floating-Point Unit," IBM RISC System/6000 Technology SA23-2619, International Business Machines Corporation, 1990. pp. 34 - 43.

[PARL95] Parallel Architecture Research Lab, README file, New Mexico State University, New Mexico, 1995.

[Rivers96] J. A. Rivers and E. S. Davidson, “Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design,” *Proceedings of the 1996 ICPP*, vol. I., Bloomingdale, IL, August 12-16, 1996, pp. 151 - 160.

[Shih92] T-P Shih, "Performance Evaluation of IBM RS/6000," Directed Study Report, University of Michigan, Ann Arbor, Michigan, 1992.

[Mangionne-Smith93] W. Mangionne-Smith et al., “Approaching a Machine-Application Bound in Delivered Performance on Scientific Code,” *Proceedings of the IEEE: Special*

Issue on Computer Performance Evaluation, vol. 81, August, 1993, pp. 1166-1178.

[Shih96] T-P Shih, "Goal-Directed Performance Tuning for Scientific Applications," Ph.D. Dissertation, University of Michigan, Ann Arbor, Michigan, 1996.

[Wellman95] J-D Wellman and E. S. Davidson, "The Resource Conflict Methodology for Early-Stage Design Space Exploration of Superscalar RISC Processors," *Proceedings of the 1995 ICCD*, Austin, Texas, October 2-4, 1995. pp. 110-115

[Wellman96] J-D Wellman, "Processor Modeling and Evaluation Techniques for Early Design Stage Performance Comparison," Ph.D. Dissertation in progress, 1996.