Flexible Timing Simulation of Multiple-Cache Configurations

Edward S. Tam, Jude A. Rivers, and Edward S. Davidson
Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, MI. 48109-2122
{estam,jrivers,davidson}@eecs.umich.edu[1]

Abstract

As the gap between processor and memory speeds increases, cache performance becomes more critical to overall system performance. Behavioral cache simulation is typically used early in the design cycle of new processor/cache configurations to determine the performance of proposed cache configurations on target workloads. However, behavioral cache simulation does not account for the latency seen by each memory access. The Latency-Effects (LE) cache model presented in this paper accounts this nominal latency as well as the additional latencies due to trailing-edge effects, bus width considerations, port conflicts, and the number of outstanding accesses that a cache allows before it blocks. We also extend the LE cache model to handle the latency effects of moving data among multiple caches. *mlcache*, a new, easily configurable and extensible tool, has been built based on the extended LE model. We show the use of *mlcache* in estimating the performance of traditional and novel cache configurations, including odd/even, 2-level, Assist, Victim, and NTS caches. We also show how the LE cache timing model provides more useful, realistic performance estimates than other possible behavioral-level cache timing models.

Keywords: *cache timing simulation model evaluation*

1 Introduction

Cache performance becomes ever more critical to overall system performance as the gap between processor and memory speed increases. The performance of a particular cache configuration depends not only on the miss ratio incurred during the execution of a particular workload but also on where in the program's execution the misses occur and the latency of each miss. However, useful timing simulation of caches is typically unavailable until late in the design stage. Using today's behavioral simulators, simple, traditional caches are evaluated early in the design cycle; however, novel cache designs are often not considered since they are difficult to model.

The issue of providing more useful cache timing simulation analysis early in the design cycle has been addressed by the Latency-Effects (LE) cache model [Tam96], which incorporates latency-adding effects into a behavioral-level simulation, particularly trailing-edge effects, bus width considerations, the effects of port conflicts, and the number of outstanding accesses that a cache can handle before blocking. Existing methods of modifying behavioral cache simulators to incorporate timing effects include adjusting the total cycle count reported by a perfect cache simulation by adding an estimated number of cycles due to cache misses (the adjusted model) or assigning a nominal leading-edge penalty to each miss as it occurs (a model we will refer to as LE-nominal). To illustrate the advantages of the LE cache model, we will compare the LE cache model's results to the results of using these other models.

---

1. This research was funded in part by a gift from IBM.

While our previous work concentrated on single, traditional caches [Tam96] and two multi-lateral caches [Rivers97], the LE cache model is easily extended to incorporate other novel cache designs. When multiple caches are present in a system, organized either as parallel (multilateral) L1 caches or sequential (multilevel) L1 and L2 caches, the time to move data among caches must be accounted for in any realistic cache timing simulation. This paper will detail the extension of the LE cache model to handle multiple cache systems by accounting for the additional latency-adding effects experienced by accesses that require data movement among the caches. We also present *mlcache*, an easily configurable multiple-cache simulator based on the extended model, which currently handles any processor system with a processor and two caches backed by a "next" level of memory. For multi-lateral cache configurations (where the two caches are accessed by the processor in parallel), the "next" level is typically a second level of cache; for multi-level caches, the "next" level is either a third level cache or main memory. This next level can clearly be incorporated similarly into *mlcache*, but is omitted for simplicity in this initial feasibility study.

Section 2 presents a brief overview of cache timing simulation. Section 3 discusses several behavioral-level cache timing models. Section 4 presents the extended LE cache model and the *mlcache* simulator and Section 5 explains our implementation and testing of *mlcache*. Section 6 presents the results of using *mlcache* as well as a comparison of the LE cache model to the other cache timing models. Conclusions are presented in Section 7.

2 Timing simulation of caches

There are many ways that cache performance has been evaluated during the design cycle. Miss ratio has been used to indicate the potential performance of a system using a given cache when running target workloads. Miss ratios are easily obtained using behavioral simulators such as DineroIII [Hill85], Tycho [Hill93], ACS [PARL95], and others. While a lower miss ratio usually indicates higher performance, the effect of the cache on overall processor/machine performance is difficult to quantify using behavioral simulators. However, behavioral simulators are very fast and easy to configure, allowing many different cache configurations to be roughly evaluated in a short period of time.

At the other end of the cache evaluation spectrum are cycle-by-cycle circuit-level simulators. These simulators are very accurate in modeling a cache's latencies and consequent effects on a machine's overall performance. However, since circuit-level simulators are complex and hard to build and modify, they normally are not developed until late in the design cycle when the processor/cache design is relatively stable. Circuit-level simulators are thus not suitable for evaluating many different, potentially novel, cache designs early in the design cycle.

The Latency-Effects (LE) cache model [Tam96] splits the difference between these two extreme approaches to cache simulation and performance evaluation. The LE cache model and its implementation incorporate the flexibility and speed provided by a behavioral simulator while providing results that are more useful, akin to the performance estimates provided by circuit-level simulators. Our LE model incorporates the parameters of traditional behavioral cache simulators, including cache size, block size, associativity, replacement policy, etc. In addition, it accounts for latencies due to trailing-edge effects, bus width considerations, port conflicts, and the number of accesses that the cache allows before blocking.

3 Behavioral-level cache timing models

Behavioral simulators are typically used to evaluate a large number of cache configurations early in the design cycle. The Latency-Effects (LE) cache model was developed to obtain more realistic, useful timing simulations. Other simpler, but generally less accurate behavioral-level cache timing models exist, including the perfect cache model, the LE-nominal model, and the adjusted model, presented below.

3.1 The LE cache model

The LE cache model accounts for nominal access latencies for misses to the caches. Furthermore, the model accounts for additional latencies due to trailing-edge effects, bus width considerations, port conflicts, and the number of outstanding accesses that a cache allows before blocking.

First, consider a single cache with a perfect backing memory. When the processor makes an access while the cache is "quiet," i.e. there is no activity still in process for any earlier access, the cache state is interrogated and this access is assigned a nominal latency depending on whether it is a read or write, or a hit or miss. In this model, a single-cycle cache (in which a read hit access to a quiet cache returns data to the processor at the end of the same cycle in which the access request is made) is assigned a latency of 0.

An access is said to commence in the cycle in which it is accepted by the cache and a read access is said to complete in the cycle in which data is returned to the processor (even though some activity related to finishing this access may still be pending in the cache). A read access to a quiet cache thus commences at (the beginning of) the same cycle, X, in which the processor makes the access request and completes at (the end of) cycle X+L, where L is the nominal latency for this access. In general, activity of prior accesses that is still in process when an access is made may delay the commencement of an access or increase the latency of the access beyond its nominal value due to other "latency-adding" effects.

Trailing-edge effects may increase the latency of accesses made to cache blocks that are currently moving to the cache from the next level of memory as the result of a previous miss. For example, if an access is made to block B at cycle Y, but a previous access to block B was made at time X and assigned a latency (including latency-adding effects) of L, then this access can complete no earlier than cycle X+L and its latency is increased, if necessary, to insure this. Since such requests may not incur the full miss penalty and they do not cause any additional traffic between the cache and the memory they are reported as hits (or "delayed hits"); such hits will, however, generally experience a latency that is greater than the nominal hit latency due to this trailing-edge effect.

Furthermore, if the bus between the cache and the next level of memory is not wide enough to return a full cache block in one cycle, then the latency is further increased by one or more cycles according to the distance between the portion of the block referenced by this access relative to the portion referenced by the original miss. The trailing-edge effects, including these bus width considerations, may thus increase the minimum completion time for this reference as a function of when the data will be made available by a previous miss.

Another factor that influences the completion time of a memory access is the number of outstanding accesses (NOA) that a cache can handle before it blocks. Once the cache is blocked, no new accesses can commence until at least one of the outstanding accesses completes. Typically, caches allow "hits under misses," i.e. hits can be completed while the cache is still serving a previous miss; many caches also allow multiple misses to be outstanding. The LE model uses NOA to model this feature. A blocking cache has NOA=1; a fully nonblocking cache has infinite NOA (the cache never blocks regardless of the number of accesses in flight). If the number of outstanding accesses equals NOA when an access is made, then the commencement of this access is delayed until one of the outstanding accesses is fully served and no other access may be made until the cycle in which this access commences. If the previously assigned access completion time is less than this commencement time plus the nominal latency, this difference is then added to the completion time.

Finally, port conflicts are considered. If no appropriate port to the processor is available at the current completion time of this access, then the completion time is further increased to the first cycle at which such a port is available.

The Latency Effects (LE) cache model thus accounts for nominal hit and miss times, plus the added delays due to each of the aforementioned effects of prior accesses on the timing of an access. Some timing effects are not accounted for in the model, including the effects of write-through vs. write-back caches, write-back buffers, and the time required to obtain the system bus in the presence of other users. These latency-adding effects will be incorporated in future refinements of the LE model. The effects of TLB misses, page faults, etc. could also be added. However, we chose to concentrate on the modeling of multiple cache systems first, as described below.

In our past work [Tam96], we used the LE cache model for performance evaluations of "traditional" single level caches backed by main memory. However, in today's machines an on-chip L1 (first level) cache is generally backed by a L2 (second level) cache, and often an L3 cache, before reaching main memory. We therefore extended the LE cache model to simulate multiple caches in a memory system. In addition to multiple levels of caches, we also evaluated multi-lateral L1 caches, i.e. parallel L1 caches Multi-lateral cache designs include the Assist Cache, as used in the HP PA-RISC 7200 [Kurpanek94][Rashid94], the NTS Cache [Rivers96], and the Victim Cache [Jouppi90].

With multiple caches, access latencies now depend not only on the cache-to-processor and cache-to-memory interactions, but must now include cache-to-cache interactions. These latencies can include the time to "promote" (move) a block from one cache to another, the time to save a replaced block from one cache in another cache, and the time to swap entries between caches. The number of cycles for these operations may vary with the cache's configuration and its nominal access times. Depending upon the bus width between the caches, different parts of the cache line will be available in the destination cache at different times. Furthermore, if an access to the cache line is initiated while it is in transit from one cache to another, this access will suffer trailing-edge effects. These latency-adding effects can easily be incorporated into the LE cache model. In this extension of the LE model, a dedicated bus was assumed to be present between the caches, so port conflicts were ignored. Also, as latencies for data movement between caches is likely to be short, we did not consider blocking either cache for accesses requiring inter-cache data movement.

3.2 Other behavioral cache timing models

Other behavioral cache timing models can easily be derived based on a behavioral cache simulator with no timing analysis, like DineroIII:

perfect cache: In this simulator, a perfect cache is assumed. All memory accesses have zero latency and there are no latency-adding effects. This model can be used to determine the best-case execution time of a processor without any cache or memory effects included.

adjusted: This is an "adjusted" processor simulator that performs a behavioral cache simulation and adds p_R*m_R + p_W*m_W cycles to the processor simulator cycle count, where p_R and p_W are some constant number of penalty cycles for read and write misses, respectively, and m_R and m_W are the number of read and write misses, respectively.

LE-nominal: This simulator (and the next) simulate a processor with cache during the simulation. While the adjusted simulator performs processor and cache simulation separately and then combines their results to obtain program execution times, the LE-nominal simulator adds the nominal latency for each access to execution time as the access occurs during simulation. Additional latency-adding effects are ignored.

Full-LE: This is the fully implemented LE cache model simulator that is described above, including all the additional latency-adding effects.

The performance predicted by each of these four models is presented in the experiments below.

4 Modeling and simulating multiple caches

We now present a high-level picture of multiple cache interactions and discuss how these caches are actually modeled in simulation.

4.1 High-level description

Figure 1 shows a "fully-connected" memory system with two caches backed by a main memory. Depending upon the specific configuration being evaluated, some of the paths will be deleted. Note that the direct path between memory and processor is not included in the figure, as it is assumed that data that returns to the processor directly from memory must still go through the cache unit. The effects of a memory to processor transfer can be obtained by assigning appropriate parameter values to it for traversing the corresponding memory-to-cache and cache-to-processor paths.)

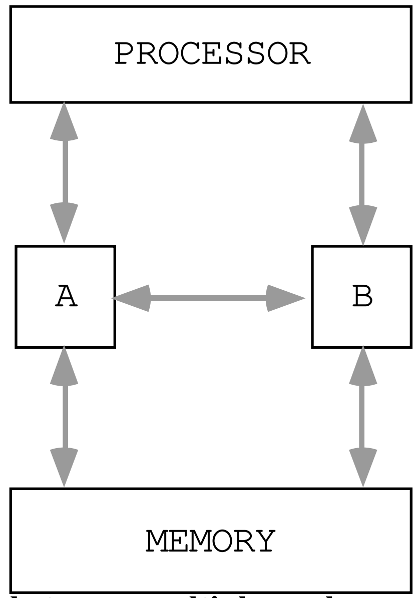


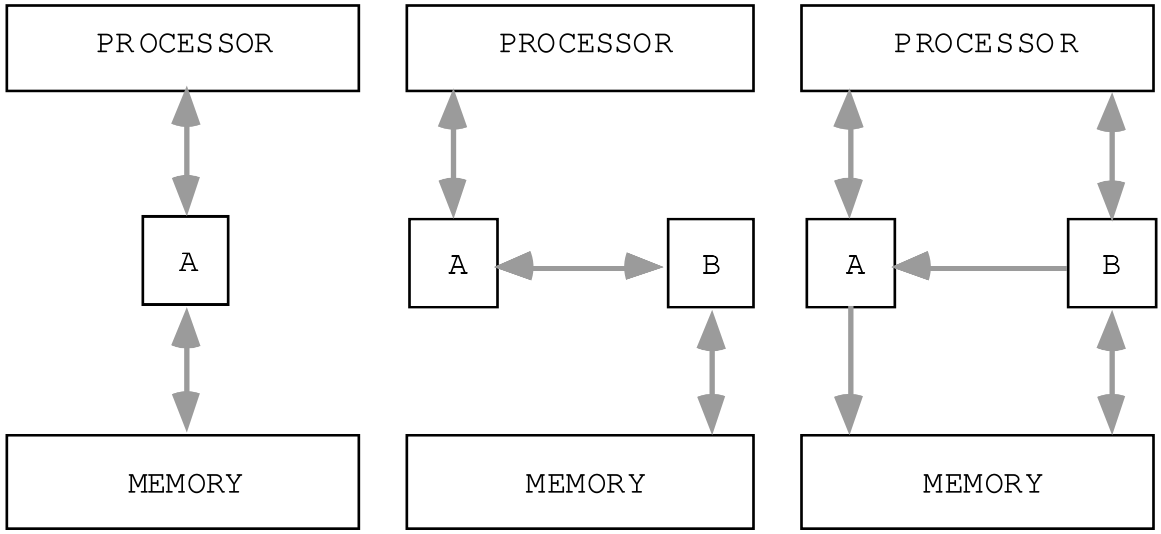
Figure 1: Interaction between multiple caches, processor, and memory

This figure can thus represent, at a high level, practically any system consisting of two caches, a processor, and memory. By removing particular arcs and elements from Figure 1, different cache configurations are represented, e.g. a traditional single cache backed by main memory (Figure 2a) or a 2-level cache, where the L2 cache is B and the L1 cache is A. The A and B caches can have different sizes, associativities, replacement policies, etc., which are specified separately by assigning parameter values.

Figure 2c shows a novel, multi-lateral cache configuration -- the Assist cache, as used in the HP PA-7200 [Kurpanek94][Rashid94]. All accesses that enter the cache system must enter through the Assist buffer (in this figure, the B cache). Note that there is no direct memory-to-A cache transfer path in Figure 2c. Whenever a word of B demonstrates *temporality*[2] during this lifetime, its cache block is promoted to the A cache. Otherwise, it resides in the B cache until it is replaced. Once a block has been promoted to the A cache, it resides there until it is replaced. In the basic Assist implementation, blocks that are replaced in A return to main memory; thus, there is no direct path from the A cache to the B cache.

A latency is assigned to each of the relevant paths in the figure for each type of operation to be performed. For instance, when a block in the B cache is found to be temporal in the Assist cache configuration, the time to promote the element to the A cache must be included in the access' latency, as promoted accesses must be serviced out of the A cache. Thus, the latency of an access can be determined by adding up the time to traverse each of the paths from where the access is resident at the time of the request to its final destination in the processor. For example, for the Assist cache configuration, the nominal miss latency, trailing-edge effects, and bus width considerations are incorporated in the memory-to-cache path, while the latency between caches and trailing-edge effects are included in the cache-to-cache path. Regardless of the cache configuration, each access is subject to the added latencies, if any, due to port conflicts and the number of outstanding accesses allowable (NOA).

2. A word exhibits temporality if it is accessed more than once during a lifetime in the cache. A lifetime of a cache block refers to the time interval that the block spends in the cache from one of its allocations until its next replacement. A particular memory block may have many lifetimes and thus may exhibit temporality in some lifetimes, but not others.



a) Single-level cache b) 2-level cache c) Assist cache

Figure 2: Different cache configurations described using the Figure 1 model

4.2 *mlcache* -- an easily configurable tool

In our previous work, our implementation of the LE cache model for single caches (the LE cache simulator) was built by modifying the DineroIII cache simulator [Hill85]. Using the concept of delayed update of the cache state in conjunction with consideration of the latency-adding effects, we were able to create a trace-driven cache timing simulator. The use of delayed update causes the effects of an access, i.e. the access' placement into the cache, the removal of the replaced line, etc., to occur only after the specified latency of the access has passed. A running global cycle count defines exactly when each access is presented to the memory system. In the case of a hit in the cache, the access can "complete" in the same cycle if the cache hit latency is zero cycles. If the access misses in the cache, the accessed block should not be "placed" in the cache until the nominal miss latency plus any cycles due to latency-adding effects have passed. Updating the cache with the effects of the access without accounting for this latency ignores the time taken within the cache to process the access. Thus, traditional behavioral cache simulators that allow an access to have an immediate affect on the cache state cannot properly account for varying access latencies. The use of a delayed update of the cache state allows behavioral cache simulators to more accurately represent each access' latency.

Adapting the original LE cache simulator [Tam96] to model multiple caches involved substantial low-level changes to the source code to achieve the desired effects and interactions for a system containing two caches, a processor, and memory [Rivers97]. Furthermore, latencies for elements moving between two caches were not taken into account in the hand-modified multiple-cache LE cache simulator. With these shortcomings, we felt the need to make the tool both more easily retargetable and more accurate in representing the timing of the target configuration. Accordingly, *mlcache* was developed as a parameterized version.

4.2.1 High-level parameterization

To make *mlcache* easily retargetable, we chose to provide a library of routines that a user could choose from when deciding what actions take place in the cache at a given time. The routines are accessed from a single C file named `config.c`. The user simply modifies `config.c` to describe all of the desired interactions shown in Figure 1 between the caches, processor, and memory. The user also controls when the actions occur via the delayed update mechanism built into the first implementation of the LE cache simulator. If more interactions are needed than those provided, they can then be coded into the simulator by hand; however, the routines that we have already provided are adequate to model most conceivable dual-cache designs.

While *mlcache* addresses many of the effects seen by a memory access in a multiple-cache configuration, some key effects are still not accounted for. Multiple-cache configurations that incorporate prefetching, as with a streaming buffer [Jouppi90], cannot be dealt with because hardware prefetching has not been included in the current implementation. Also, some configurations, e.g. a smaller or less associative cache "backing" a larger or more associative (possibly multilateral) cache can potentially violate the multi-level inclusion principle [Baer88]; the potential for this violation is common in such caches and has not been addressed in our current studies.

Support Routine	Description
check_for_cache_hit()	check to see if accessed block is present in the cache
update()	place an accessed block into the cache
move_over()	move an accessed block from one cache to the other
do_swap()	move an accessed block from cache1 to cache2 and move the evicted block to cache1
do_swap_with_inclusion()	place an accessed block into both cache1 and cache2 and move the evicted block from cache2 to cache1
do_save_evicted()	move the block evicted from cache1 to cache2
find_and_remove()	remove a block from a cache
check_for_reuse()	determine if a block exhibits temporal behavior (word re-use)

Table 1: Support routines used to control cache state and interactions

Table 1 shows the routines provided for the user to choose from and a brief description of each. Figure 3 shows portions of the `config.c` file wherein an Assist cache configuration is modeled by using these routines. As can be seen, the operations in the `config.c` file are all very high level and easily understandable and relieve the user from learning the intricacies of the cache simulator's low-level operation in order to model a new cache.

4.2.2 Assessing latencies for multiple caches

Accounting for latencies between caches is a simple extension of the LE cache model - given that we know what operation is occurring, we can add the corresponding latency onto the access time and then account for any latency-adding effects. For this paper, we assume that dedicated busses (as wide as the smaller cache's blocksize) are present between the caches so that we may ignore bus width considerations between the caches for moves between A and B. We also assume that, given these dedicated busses, there are dedicated ports for accesses traveling between the caches; this permits a processor read from say, the A cache while a different element is being moved to A from the B cache. Implementations of these caches in a real, well-designed machine would likely satisfy these assumptions.

Different latencies can also be assigned to a path depending upon the operation that is being performed. For instance, we see that in Table 2, the latency assigned for the `move_time` differs among the cache configurations. For an Assist cache (illustrated in Figure 2c), moves between the caches are always in a single direction, from the B cache (buffer) to the A cache (main cache). Thus, a move in the Assist cache configuration in our experiments requires a single cycle, meaning an access that misses in the A cache and hits in the B cache and exhibits temporality is satisfied with a single cycle latency. Accesses that hit in the A cache are returned in the same cycle (zero cycle hit latency), as are accesses that hit in the B cache that do not exhibit temporality.

For a Victim cache, promotions from the B cache to the A cache require a swap to be performed: the block from the B cache is moved into the A cache and the block evicted from the A cache as a result of the move is moved to the B cache. Normally, this operation cannot complete in a single cycle, as there is only a single, albeit dedicated, bus between the caches, but two elements need to be moved using the common bus. Thus, we can assign a maximum latency of two cycles for a move between the caches for the Victim configuration or assign a one cycle latency and assume a 2 block wide bus; we assigned latency 2 in the following experiments. If there is an access to a block that is moving between caches, the trailing-edge effect seen by this latter access is properly accounted for by the LE cache model.

```
/*
   this is the standard handler for each access.  it checks in the
   to see if the access is there first.  if it isn't, it checks in
   B cache.  if it's present in a cache, it handles the appropriate
   if the access misses in both caches, a miss is processed.  this
   "sequentially" perform a parallel check of the two caches in the

   other designs may not need both caches checked (e.g. MLCOs that
   the memory access stream based on some criteria like address (oc
   functionality (integer/floting point), etc.).
   */
int handle_access(int cycle_count, UpdateEntry{ *Entry)
  /* check for hit in A cache "first" */
  if(!check_for_cache_hit(cycle_count,Entry))
    /* miss in A cache - check in B cache */
    if(!check_for_cache_hit(cycle_count,Entry))
      /* miss in both caches - handle the miss */
      access_time = handle_miss(cycle_count,Entry);
    else
      /* hit in B cache (after miss in A cache) - handle the B cach
      access_time = handle_B_cache_hit(cycle_count,Entry);
  else
    /* hit in A cache - handle the A cache hit */
    access_time = handle_A_cache_hit(cycle_count,Entry);
  return access_time; }

int handle_A_cache_hit(int cycle_count, UpdateEntry{ *Entry)
  /* hit in A cache, so just update stack, etc. for A cache */
  Entry->on_completion = DO_UPDATE;
  Entry->access_latency = cache_latency;
  Entry->which_cache = ACACHE;
  return(handle_hit_timing(cycle_count,Entry,Entry->A)); }

int handle_B_cache_hit(int cycle_count,UpdateEntry{ *Entry)
  /* hit in B cache, so do appropriate updates */
  /* for assist cache, update is to promote it to A cache */
  if(check_for_reuse((Entry->B))) {
      Entry->on_completion = DO_MOVE;
      Entry->move_direction = B_TO_A;
      Entry->access_latency = move_time + cache_latency;
      return(handle_miss_timing(cycle_count,Entry,Entry->B)); }
  else {
      /* just update this access to B cache */
      Entry->access_latency = cache_latency;
      Entry->which_cache = BCACHE;
      return(handle_hit_timing(cycle_count,Entry,Entry->B)); } }
```

Figure 3: Part of a sample `config.c` file, showing the basic evaluation process for each access

In a 2-level cache, the second level cache is typically much slower than the first level cache (which also permits it to be much larger than the L1 cache); in the following experiments, we have assigned a five cycle latency for an element to move from the B (L2) cache to the A (l1) cache.

	DM	2-LEVEL		ODD/EVEN		ASSIST		VICTIM		NTS	
Cache	A	A	B	A	B	A	B	A	B	A	B
Size	8/16K	8K	32K	4K	4K	8K	1K	8K	1K	8K	1K
Associativity	1/1	1	4	1	1	1	full	1	full	1	full
Replacement policy	–/–	–	LRU	–	–	–	LRU	–	LRU	–	LRU
move time	–	5		–		1		2		–	
r/w latency to next level	11/17	5	11/17	11/17	11/17	–	11/17	11/17	–	11/17	11/17

Table 2: Characteristics of six different cache configurations studied (times/latencies are in cycles)

The semantics of the other multiple-cache configurations are discussed in more detail in Section 6, where we present the results of our experiments. From these brief examples, however, it is easy to see that extending the LE cache model to handle multiple caches was accomplished in a straightforward manner and provides user-friendly, high-level interface. This modular, library-based approach to cache configuration allows a significant range of cache configurations to be examined early in the design cycle.

5 Implementation and testing of *mlcache*

We have implemented six different cache configurations using the *mlcache* simulator: a direct-mapped single cache, an Assist cache, an NTS cache, a Victim cache, an odd/even cache, and a two-level cache. The latencies used for the timing simulation of these caches are shown in Table 2.

5.1 Simulation environment

A timing simulation of caches is of limited use without considering the latency-masking effects of processor execution. Thus, we integrated our cache simulator with the RCM_brisc instruction-level processor simulator [Wellman95], as was done with the LE simulator in [Tam96]. 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 IBM RS/6000 [Bakoglu90]. 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 constant, prespecified amount of time. However, the perfect cache model yields an unrealistic estimate of program performance; cache and memory effects must be included in any processor simulation if it is to realistically evaluate a program's performance. To make the processor model more modern, we configured RCM_brisc to represent an eight-wide, in-order issue, eight functional unit processor (2 FXUs, 2 FPUs, 2 L/S units, and 2 branch units). Each functional unit has a number of reservation stations that buffer instructions between the issue and execution units, permitting out-of-order execution and completion of instructions. There are five register files in the system: a 32 register GPR file, a 32 register FPR file with 40 physical registers (with register renaming), a two register link register file (for branches), an 8 register condition register file, and a two register count register file. An infinite number of register ports is assumed to minimize instruction issue conflicts and constraints and thereby increase the demand on the cache. In the optimal case, this machine can issue up to two load/stores per cycle and complete two per cycle; it has two cache read ports and a single write port.

The memory hierarchy includes separate L1 caches for instruction and data. Since this paper focuses on multiple-cache structures for data, we assume a perfect L1 instruction cache for simplicity and treat all instruction fetches as hits with zero latency. Thus, all of the cache configurations we describe in this paper represent the data cache configurations of the processor/cache combinations that we evaluate.

The *mlcache* simulator 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 *mlcache* maintains the state of the caches itself and does not take into account virtual memory or TLB effects. It models up to the first two levels of data cache and assumes a perfect memory thereafter, regardless of the number of level of caches beyond that. The RCM_brisc tool was chosen for convenience since the LE cache model was implemented in the same spirit using the Resource Conflict Methodology [Wellman95] model.

5.2 Target benchmarks

To evaluate the performance of the caches, we chose from a variety of scientific, floating-point codes and memory-intensive integer benchmarks. The selected benchmarks and their descriptions are shown in Table 3.

Program	Program Description	Memory References (millions)		Perfect Memory Performance	
		Loads	Stores	Cycle Count (millions)	IPC
APPBT	navier-stokes eqns approx.	10.621	1.043	44.414	1.126
APPSP	navier-stokes eqns approx.	10.183	1.542	41.435	1.207
EQNTOTT	boolean -> truth table	14.994	2.474	59.807	0.836
FEMC	em object identification	10.730	4.548	51.507	0.971
FFTPDE	3-D fast fourier transform	9.229	5.506	35.507	1.408
SIMPLE	2-D Lagrangian hydrodynamics	11.277	6.302	44.180	1.132
SPHOT	monte-carlo particle transport	16.948	5.151	65.139	0.767

Table 3: Benchmark program characteristics

APPBT, APPSP, and FFTPDE are from the NAS Benchmark suite, EQNTOTT is from the SPEC92 suite, and SIMPLE and SPHOT are from the RICEPS suite. The seventh benchmark, FEMC [Chatterjee93], is a floating point code developed and in use at the University of Michigan Radiation Laboratory for evaluating electromagnetic backscatter from a distant object. Due to time and resource constraints, we traced and simulated the execution of 50 million instructions of each program; performance of the first 10 million instructions of each program was discarded to avoid program initialization effects. Trace collection was done on an IBM RS/6000 running AIX 3.2.4 with the ATRACE tracing package (developed by Ravi Nair of the IBM T.J. Watson Labs).

Table 3 also shows the number of cycles required to execute the code on our RCM_brisc processor simulator with a perfect cache (i.e. all memory accesses are satisfied in the same cycle in which they commence). These numbers are needed for determining the Relative Cache Effect Ratio, as defined in [Rivers97], which is:

$$RCR_X = \frac{\text{CycleCount}_X - \text{CycleCount}_{PERFECT}}{\text{CycleCount}_{DM:8K} - \text{CycleCount}_{PERFECT}}$$

This ratio provides the finite cache penalty of a given cache configuration relative to the base 8K direct-mapped cache penalty. The DM:8K cache thus has an RCR of 1, and caches that perform better than the DM:8K have RCRs between zero and one.

5.3 Experiments

Two main experiments were run using the combined processor/cache simulator (which we call *RCM_brisc+mlcache*) and the chosen benchmarks. The first experiment uses *mlcache* to compare the performance of different cache configurations when running the chosen benchmarks. The second experiment evaluates the use of the Full-LE cache timing model in these new, multiple-cache configurations compared to the use of a perfect cache simulator, an adjusted simulator, and an LE-nominal simulator.

6 Results

6.1 Multiple-cache configuration performance

Using the simulation environment described in Section 5, we obtained the cycle counts shown in Figure 4 for the various cache configurations and benchmarks. From this figure, we can see a general trend for the caches -- with the exception of the ODD/EVEN cache configuration, the larger and/or multiple cache configurations exhibit lower cycle counts and thus better performance than the base DM:8K configuration. Given that each of the multiple cache configurations (aside from the ODD/EVEN) have at least 1K more data store available to them, the increased performance is not surprising. What is interesting to see, however, is the amount of improvement seen using the different cache configurations. This difference is highlighted when we calculate the relative cache effect ratio (RCR) for each configuration running each benchmark. The RCRs are shown in Figure 5.

Consider each configuration's performance from worst (high RCR) to best. As expected, the performance of the ODD/EVEN:(4+4K) cache is similar to the performance of the base, DM:8K cache. The odd/even cache is an interleaved cache, with half of the cache handling odd-numbered block addresses and the other half handling even-numbered block addresses. This similar performance might be expected because the overall cache size is identical in the two configurations, as is the mapping policy (the same number of odd and even sets exist in both configurations).

The 16K direct-mapped cache was included in previous studies for comparison with the (8+1)K cache-and-buffer configurations (in this experiment, the Assist, Victim, and NTS caches). We felt that an (8+1)K cache would require at most the same amount of die area (including all necessary state-maintenance logic) as the DM:16K configuration would. Thus, if the (8+1)K configuration performed as well or better than the DM:16K configuration, it should be a clear win. As expected, the DM:16K configuration consistently performs better than the DM:8K configuration. However, the RCR shows that the performance improvement typically does not double when doubling the cache size; only in FEMC does the RCR of the DM:16K configuration drop below 0.5.

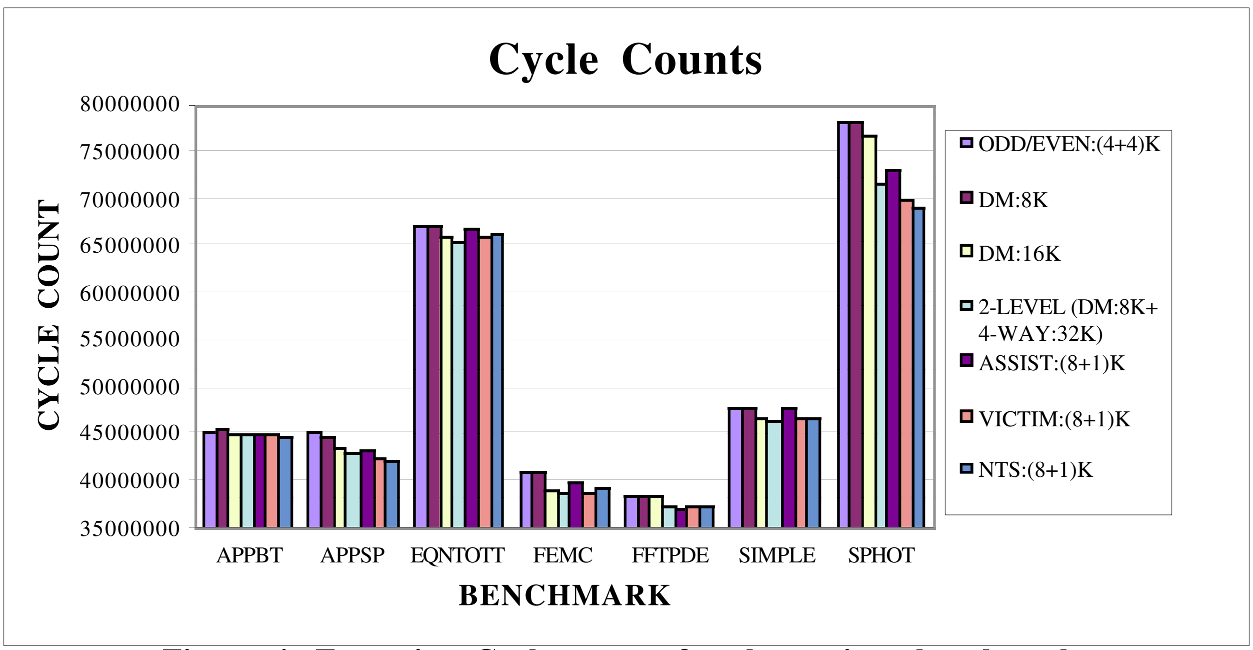


Figure 4: Execution Cycle counts for the various benchmarks using the various cache configurations

The first multiple-cache configuration is the traditional two-level cache: a direct-mapped 8K cache backed by a 4-way set-associative, 32K second level cache, with a latency of 5 cycles between the two levels. Despite this latency, the larger overall size of the cache structure allowed the 2-level cache to consistently perform better than either of the basic direct-mapped configurations.

The Assist cache configuration is the first of the "direct-mapped main cache with fully-associative buffer" configurations we evaluated. Its "main" cache is an 8K direct-mapped cache and the B cache is a 1K fully-associative buffer. All accesses to memory enter the cache system through the buffer. As in previous studies, the Assist cache configuration typically performs better than the DM:8K configuration, and sometimes (in APPSP, FFTPDE, and SPHOT) outperforms the DM:16K configuration. Its erratic performance relative to a larger DM cache may explain why the Assist cache is not widely used (or considered) in modern microprocessor/cache designs.

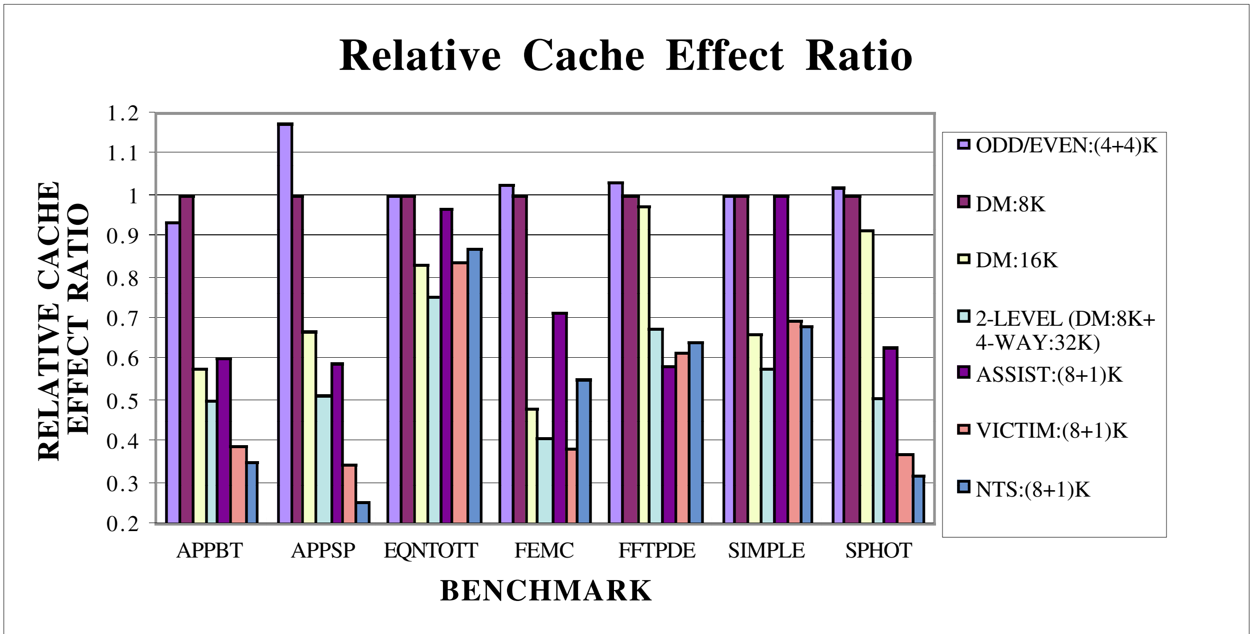


Figure 5: Relative Cache Effect ratio for the various benchmarks using the various cache configurations

The Victim cache is the second cache-and-buffer configuration. The Victim cache acts essentially as a small second-level cache to the direct-mapped main cache: it holds items evicted from the main cache. If there is a hit in the buffer, the hit block in the buffer is swapped with the block it replaces in the main cache. Unlike a two-level cache scheme, however, the 1K buffer is fully associative and fast -- including a reasonable swap time between caches. The latency to access an entry that resides in the buffer is two cycles. The full-associativity of the buffer also effectively increases the associativity of the main cache -- multiple items that map to the same set in the direct-mapped cache structure can all reside in the victim buffer simultaneously. From the RCR ratios, we see that the Victim cache performs very, very well. Among the three cache-and-buffer configurations, it has the lowest RCR in EQNTOTT and is never far from the lowest RCR. The Victim cache evens performs better than the DM:16K and the 2-level cache in all but EQNTOTT and SIMPLE, where the larger size of the two-level cache structure is clearly beneficial. However, particularly when die area is at a premium, an (8+1)K Victim cache appears to be a very viable alternative to a 16K direct-mapped cache or an (8+32)K two-level cache structure.

The NTS cache is the third cache-and-buffer configuration. The NTS cache manages memory accesses based on their access patterns -- temporal data is allowed to stay in the cache while non-temporal data is sent to the fully-associative buffer. Specifically, each block in the L1 cache and buffer and the L2 cache has an associated "nontemporal bit." This bit is set if the block is ever referenced temporally during a main (A) cache lifetime. On a miss access, the block is allocated to cache A if temporal and to the buffer (B) if nontemporal. Blocks return to L2 when replaced in A or B. In this configuration, it is less likely that useful, temporal data is displaced by nontemporal data. The small buffer is used to hold temporal data which generally has a shorter useful lifetime. Its performance is somewhat more variable than that of the Victim cache on this particular group of benchmarks, however. In APPBT, APPSP, SIMPLE, and SPHOT, the NTS cache has a lower RCR than the Victim cache, and is best overall in APPBT. APPSP, and SPHOT, achieving an RCR as low as .25 in APPSP. In EQNTOTT and FFTPDE, the performance of the NTS cache is slightly worse than the Victim cache, and significantly worse only in FEMC.

Thus, we can see that, for this set of configurations and benchmarks, either the Victim or NTS cache should be used to provide high performance with low cost.

6.2 The LE cache model vs. other behavioral cache timing models

For these experiments, we evaluated the outputs of the four different cache timing models when used in the *RCM_brisc+mlcache* simulator. To assess these four timing models, we chose to simulate two of the more memory-intensive codes, the integer-based EQNTOTT and the floating-point based SPHOT. The simulator was fed with the same traces used in the experiments in Section 6.1. The only difference here is that we used the perfect memory, adjusted, and LE-nominal cache timing models in addition to our Full-LE model.

Figure 6 shows the output of the simulator using each of the timing models for each cache configuration. The results are consistent for both of the benchmarks. The perfect memory output is constant across all configurations, as it should be -- regardless of the cache configuration, memory accesses require a constant amount of time. Since latencies are not incorporated in the perfect cache model, the cycle count output by the simulator is the time required by the processor to execute the instructions without memory effects.

We also see that the adjusted timing model runs report the highest number of cycles executed for all but the 2-level cache configuration on SPHOT. The exaggerated cycle counts result from the model ignoring the masking effects of processor execution while the cache is active. As a result, the sum of the nominal latencies is directly added to the program's overall simulated execution time. Given the nature of the adjusted timing model, we would expect that its cycle count output would consistently be the highest; the reason for the apparent anomaly for the 2-level configuration is addressed below, when we discuss the Full-LE model results.

On the other hand, the LE-nominal timing model consistently reports the lowest number of cycles executed of the three latency-incorporating models. Despite incorporating processor masking and leading-edge penalties for misses, additional latency-adding effects are not incorporated into the LE-nominal simulation. If there is a lot of processor masking, i.e. the processor is able to do useful work while waiting for memory accesses to return, the output of the LE-nominal model can be quite reasonable. However, when the processor is stalled or slowed due to memory dependencies, the penalty cycles will be underestimated and the LE-nominal model will report optimistic execution times. This inaccuracy is noticeable, but possibly acceptable in these simulations.

The Full-LE model cycle counts are consistently between the adjusted and LE-nominal cycle counts. This is clearly due to its ability to account for processor masking effects (which the adjusted model does not) while also taking into account latency-adding effects (which the LE-nominal model does not). While the LE-nominal output can match the output of the Full-LE model in some instances, as stated earlier, this only occurs when there is a lot of processor masking involved in the program's execution and hence accurate cache characterization, and indeed cache performance itself, is not as critical.

It is interesting to see that, for the 2-level and Victim cache configurations, the cycle counts reported when using the Full-LE model are nearly as large (and for the 2-level configuration running SPHOT are even greater than) those reported by the adjusted model simulation. This is due to the extended LE cache model's incorporation of latencies between caches in addition to any latency-adding effects over and above the nominal hit/miss latencies. For the 2-level cache, the latency between the first and second levels is 5 cycles. In SPHOT, there are a total of 1.2 million promotions from the second level cache to the first level. All of these promotions require no time in the adjusted cache model, as these accesses still "hit" in the cache structure. For SPHOT, it is apparent that the incorporation of these latencies results in a higher, though more realistic, cycle count reported by the simulator. In EQNTOTT, there are only 0.5 million promotions from the second level cache; this is not large enough to cause the cycle count reported by the Full-LE model to exceed that of the adjusted model. A similar situation can be seen with the Victim cache, although the adjusted simulation cycle count output is still higher than that reported by the Full-LE model -- it is much closer for SPHOT than for EQNTOTT and the latency of swaps between caches is only two cycles, so their impact in the Victim cache configuration should be less than that seen in the 2-level configuration.

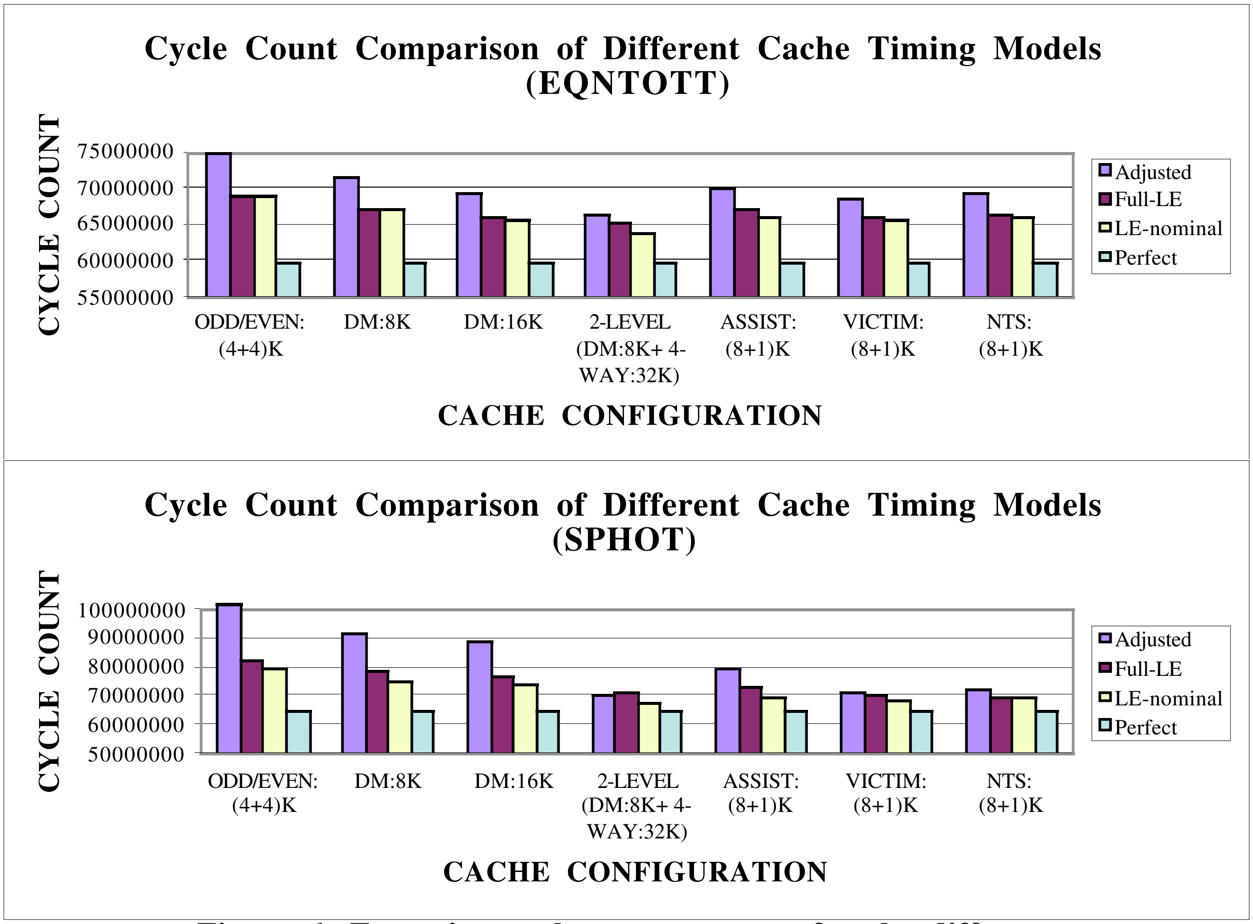


Figure 6: Execution cycle count outputs for the different timing models while simulating different cache configurations

Thus, we see that of the four different cache timing models, the Full-LE cache model gives the most consistent, explainable performance reports. The adjusted model outputs range from 29.7% greater (SPHOT odd/even) to 2% lower reported cycle counts (SPHOT 2-level), while the LE-nominal model reports 0.07% (EQNTOTT DM:8K) to 5.4% (SPHOT 2-level) lower cycle counts when compared to the Full-LE model output. The perfect cache model outputs are constant and do not account for any memory effects for any cache configurations, as expected. The performance of the other cache timing models confirms the benefits of incorporating latency-adding memory and processor-masking effects into processor/cache simulation in providing more useful, realistic performance estimates.

7 Conclusion

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. The performance of a particular cache configuration in a system is dependent not only on the number of misses that the configuration incurs during the execution of a particular workload (the miss ratio), but also where in the program's execution the misses occur and the latency of each miss. However, useful timing simulation of caches is typically unavailable until late in the design stage. In this paper we have presented the basic Latency-Effects (LE) cache model, which accounts for the nominal leading-edge penalties experienced for each cache miss as well as additional latencies due to trailing-edge effects, bus width considerations, port limitations, and the number of outstanding accesses that the cache allows before it blocks.

We also extended the LE cache model to handle multiple caches and the latencies between them. An easily configurable, extensible multiple-cache simulator, *mlcache*, was developed based on the extended LE cache model. We used *mlcache*, combined with the RCM_brisc instruction-level processor simulator [Wellman95], to determine the performance of six different cache configurations: the baseline 8K direct-mapped cache; a larger, 16K direct-mapped cache; an odd/even interleaved cache; a 2-level cache; and several novel, cache-and-buffer configurations: Assist cache, Victim cache, and NTS cache. Thus, the use of *mlcache* allows designers to easily configure and evaluate various multiple-cache configurations and their timing effects early in the system design cycle.

Finally, we showed the benefits of using the full Latency-Effects cache model over the perfect, adjusted, and LE-nominal cache timing models. The latter two models are other (simpler) timing models that can be used to incorporate latencies into a behavioral cache simulation. The adjusted cache model's simulated cycle count output was usually the highest (up to 29.7% greater than the Full-LE model's output) while the LE-nominal reported cycle counts were lower than those output by the Full-LE model (by up to 5.4%). The cycle count reported by the Full-LE model was only higher than that of the adjusted model on cases where the adjusted model did not account for latencies included in the Full-LE model (such as latencies between caches). While the LE-nominal timing model functions well for codes which include many processor masking effects, those codes that are memory bound can be modeled much more realistically by the Full-LE model, due to the additional latencies and effects that it incorporates into the simulation.

8 References

[Baer88] J-L. Baer and W-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of ISCA-15*, May 1988.

[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.

[Chatterjee93] A. Chatterjee and J. Volakis, "Application of Edge-based Finite Elements and ABCs to 3-D Scattering," *IEEE Transactions on Antennas and Propagation*, 1993.

[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.

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

[Hill93] M. D. Hill et al., "Wisconsin Architectural Research Tool Set," *Computer Architecture News*, vol. 21 No. 4, September, 1993, pp. 8-10.

[Jouppi90] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 17th Annual ISCA Conference*, Los Alamitos, CA, May 1990, pp. 364-373.

[Kurpanchek94] G. Kurpanchek et al, "PA-7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface." *COMPCON Digest of Papers*, February 1994.

[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.

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

[Rashid94] E. Rashid et al, "A CMOS RISC CPU with On-Chip Parallel Cache," *ISSCC Digest of Papers*, February 1994.

[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.

[Rivers97] J. A. Rivers, E. S. Tam, and E. S. Davidson, "On Effective Data Supply for Multi-Issue Processors," to appear in the *Proceedings of the 1997 ICCD*.

[Tam96] E. S. Tam and E. S. Davidson, "Early Design Cycle Timing Simulation of Caches," Technical Report CSE-TR-317-96, University of Michigan, November 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.