# *mlcache:* A Flexible Multi-Lateral Cache Simulator

Edward S. Tam, Jude A. Rivers, Gary S. Tyson, and Edward S. Davidson
Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, MI. 48105
{estam,jrivers,tyson,davidson}@eecs.umich.edu

**Abstract**

*As the gap between processor and memory speeds increases, cache performance becomes more critical to overall system performance. Multi-lateral[1] cache designs such as the Assist, Victim, and NTS cache have been shown to perform as well as or better than larger, single structure caches. Unlike current cache simulators, mlcache (an event-driven, timing-sensitive simulator based on the Latency Effects cache timing model) can evaluate a variety of multi-lateral cache configurations. It was developed to help designers in the middle of the design cycle decide which cache configuration would best meet the performance needs of the target processor. It can easily model various cache configurations by using its library of cache state and data movement routines.We use the SPEC95 benchmarks to illustrate how mlcache can be used to compare the performance of several different data cache configurations.*

***Keywords****: multi-lateral cache, timing simulation, performance evaluation*

## 1. Introduction

As the gap between processor and memory speeds increases, cache performance becomes more critical to overall system performance. To address this, processor designers typically design for the largest possible caches that can still remain on the ever growing processor die. However, multi-lateral cache designs such as the Assist Cache [1], Victim Cache [2], and NTS Cache [3] have been shown to perform as well as or better than larger, single structure caches while requiring less die area, thereby allowing more space for enhancements of other resources [4][5].

The *mlcache* multi-lateral cache simulator was developed to help designers in the middle of the design cycle decide which cache configuration would best meet the performance needs of the target processor. Early in the design cycle, metrics such as miss ratio and bandwidth requirements can be used to narrow the spectrum of caches considered for a processor design. Before detailed cycle-by-cycle simulators are constructed for final candidate designs, it is often very helpful to use higher-level, mid-cycle timing simulators to obtain a more detailed evaluation of cache and processor performance. *mlcache* is an event-driven, timing-sensitive cache simulator based on the Latency Effects cache timing model [6]. It can be easily configured to model various multi-lateral cache configurations by using its library of cache state and data movement routines. For interactions not modeled in the library routines, users can write their own management routines and call them from this simulator, which is structured for easy extensibility. The tool can be easily joined to a wide range of event-driven processor simulators such as *RCM_brisc* [7], *SimICS* [8], and *SimpleScalar* [9]. Together, a combined processor-and-cache simulator such as *SimpleScalar+mlcache* can provide detailed evaluations of multiple cache designs running target workloads on proposed processor/cache configurations.

In this paper we present the *mlcache* tool and show how it can be used to compare the performance of several different L1 data cache configurations when running the SPEC95 benchmarks.The multi-lateral cache configurations considered include the Assist cache, Victim cache, NTS cache, and PCS cache [10]. We show how *mlcache* easily realizes each of these and exposes its performance for various design parameters. *mlcache* can also easily model single structure caches.

Section 2 below discusses the modeling and simulation of multiple caches. Section 3 introduces the *mlcache* tool. Section 4 discusses the cache configurations we evaluate. Section 5 presents the implementation and testing of *mlcache*. Section 6 discusses our simulation environment, benchmarks, and experiments. Section 7 presents experimental results, and we conclude in Section 8.

## 2. Modeling and simulating multi-lateral caches

In order to model and compare multi-lateral cache con-

[1] Multi-lateral cache refers to a level of cache that contains two or more data stores that have disjoint contents and operate in parallel.

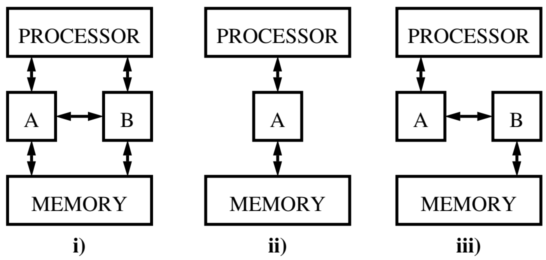

Figure 1: Schematic views of processor-memory systems: i) a fully-connected multi-lateral cache, ii) a single structure cache, and iii) a two-level cache.

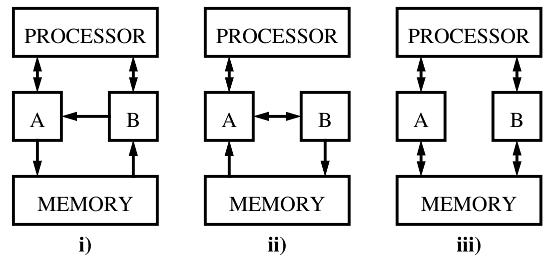

Figure 2: Representations of some multi-lateral cache configurations: i) Assist cache, ii) Victim cache, and iii) NTS and PCS caches.

figurations, it is helpful to have a figure from which each of the configurations can be derived. Figure 1i shows a "fully-connected" processor-memory system with two cache structures in L1 backed by a main memory. Depending upon the specific configuration being evaluated, some of the links will be deleted. No direct link between memory and processor is included in the figure, as cache-bypass data can simply be routed through the cache unit. The effects of a direct memory to processor transfer can be obtained by assigning it an appropriate delay for traversing the corresponding memory-to-cache and cache-to-processor links.

This figure can thus represent, at a high level, practically any system consisting of two caches, a processor, and memory. (Note that the "memory" in this figure can actually represent a second level cache.) By removing particular arcs and elements from Figure 1i, different cache configurations can be represented, e.g. a traditional single structure cache (Figure 1ii) or a 2-level cache (Figure 1iii). The A and B caches can have different sizes, associativities, replacement policies, etc., which are specified separately by assigning parameter values.

Figure 2 shows representations of several multi-lateral cache configurations. Figure 2i shows the Assist cache configuration, used in the HP PA-7200 [1][11]. All blocks that enter the cache from memory must enter through the Assist buffer (B). Note that there is no direct memory-to-A cache transfer link in Figure 2i. B uses FIFO replacement, and evicted blocks are moved ("promoted") into A unless they are tagged with a compiler-supplied spatial-only hint. Thus, B serves as a "staging area" for data, potentially reducing the number of conflict misses that a program would experience if it were run on a system with a single, direct-mapped L1 cache. Once a block has been promoted to A, it resides there until replaced under its LRU policy. As originally proposed, dirty blocks that are replaced in A are written back to memory; thus, there is no direct link from A to B.

In the Victim cache (Figure 2ii), blocks replaced in A move to B, and move back to A if hit while in B. In the NTS or PCS cache (Figure 2iii), blocks deemed to be temporal are allocated to A, nontemporal blocks to B[2].

In the middle of the design cycle, it is helpful to incorporate latency effects when considering memory accesses, as miss ratios alone are not a sufficiently accurate performance metric. When cycle-level simulation is performed, two cache designs with similar miss ratios can have drastically different program execution times depending upon the actual latency of each access. The Latency Effects (LE) cache timing model [6] does account for the latency seen by each memory access; it is a paramaterizeable model that determines the latency for each access by considering leading and trailing edge effects, bus width and contention effects, port conflicts, and the number of outstanding accesses (NOA) allowed without blocking. LE also determines the latency of hits, misses, and delayed hits[3] to single structure caches. For multiple-cache configurations, more latencies must be considered.

A latency is assigned to each of the relevant links in the figure for each type of operation to be performed. For instance, if an access is made to a block during its promotion from B to A, the promotion time must be included in the access latency. The latency of an access can be determined by summing the time to traverse each link of the critical path of the access. For the Assist cache configuration, the nominal miss latency, trailing-edge effects, and bus width and contention considerations are incorporated in the memory-to-cache link, while the latency between caches and trailing edge effects are included in the cache-to-cache link. Regardless of the cache configuration, each access is subject to the added latencies, if any, due to port conflicts and NOA.

2. A block is considered temporal if any word in it is accessed more than once during a tour. A block is considered spatial if more than one word is accessed during a tour. A block tour refers to one of the time intervals that the block spends in the cache (between an allocation and its subsequent eviction). A given block can have many tours through the cache.

3. Delayed hits are accesses to data currently returning to the cache on behalf of earlier misses to the same block.

| Support Routine | Description |
|---|---|
| *check_for_cache_hit()* | check to see if an 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 another |
| *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 cache1 to cache2 |
| *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 |

Table 1: The basic support routines provided by *mlcache*. Users control the cache state and interactions by calling these routines from a configuration file.

# 3. *mlcache* — an easily configurable tool

## 3.1. High-level parameterization

To make *mlcache* easily retargetable, we chose to provide a library of routines that users can 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 modifies `config.c` to describe all of the desired interactions between the caches, processor, and memory. The user also controls when the actions occur via the delayed update mechanism built into the cache simulator[4]. Table 1 gives brief descriptions of the provided routines. If other interactions are needed, they can then be coded into the simulator by hand. However, the routines that are provided are already sufficient to model many multi-lateral cache designs.

While *mlcache* models many of the effects seen by a memory access in a multi-lateral configuration, some key effects are still not accounted for. Prefetching, e.g. with a streaming buffer [2], 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 multi-lateral) cache can potentially violate the multi-level inclusion principle [13]; the potential for this violation is common in multi-lateral caches and has not been addressed in our current studies.

4. Delayed update is used to allow a behavioral cache simulator such as DineroIII [12] to account for latency-adding effects. The use of delayed update causes the effects of an access, i.e. an access' placement into the cache, the removal of the replaced block, etc. to occur only after the calculated latency of the access has passed.

| | Single | Assist | | Victim | | NTS | | PCS | |
|---|---|---|---|---|---|---|---|---|---|
| **Cache** | A | A | B | A | B | A | B | A | B |
| **Size** | 8/16K | 8K | 1K | 8K | 1K | 8K | 1K | 8K | 1K |
| **Associativity** | 1/1 | 1 | full | 1 | full | 1 | full | 1 | full |
| **Replacement Policy** | –/– | – | FIFO | – | LRU | – | LRU | – | LRU |
| **move time** | – | 1 | | 2 | | – | | – | |
| **latency to next level** | 18 | – | 18 | 18 | – | 18 | 18 | 18 | 18 |

Table 2: Characteristics of the five configurations studied. Times and latencies are in cycles.

## 3.2. 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 adjust for any latency-adding effects. For this paper, we assume that dedicated busses between the caches are as wide as the smaller cache's blocksize so that we may ignore bus width considerations for moves between A and B. We also assume that there are dedicated ports for accesses traveling between the caches (e.g. this permits a processor read from the A cache while a different element is being moved to A from B). Implementations of these caches in a well-designed machine would likely satisfy these assumptions.

A link can have different latencies depending upon what operation is being performed. The latency assigned for the `move_time` can differ among the cache configurations, as shown in Table 2. For an Assist cache (Figure 2i), moves between the caches are always in the direction from B to A. In our experiments, an Assist cache move requires a single cycle, meaning that an access that hits a block being promoted from the B cache to the A cache is satisfied with a two cycle latency (one cycle for the move and one cycle to return the data to the processor). Accesses that hit in the A cache are returned in the next cycle, as are accesses that hit in the B cache.

For a Victim cache (Figure 2ii), 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 it evicts from the A cache is placed in the B cache. In our Victim configuration, this operation cannot complete in a single cycle, as there is only a single, albeit dedicated, 1 block wide bus between the caches, and two elements need to be moved using the common bus. Thus, we assign a latency of two cycles for this move (swap).

Each of the multi-lateral configurations is discussed in more detail in Section 4. From these brief examples, however, it is easy to see that this modular, library-based approach to defining a cache configuration does allow a significant range of configurations to be examined.

## 4. Cache configurations

In each of the following multi-lateral cache configurations, both the A and B caches are checked in parallel on each memory access.

### 4.1. Assist cache

The Assist cache [1] (Figure 2i) is a multi-lateral design where the B cache is used as a "staging area" for data entering the A (main) cache. On a hit, the data is returned to the processor the next cycle, but remains in the cache in which it is found. On a miss, the block entering the L1 cache structure is placed into the B cache. Blocks in the B cache are managed in a FIFO fashion; blocks evicted from the B cache are placed in the A cache. A dirty block evicted from the A cache due to a promotion returns to the next level of memory.

In the Assist implementation in the HP PA-7200 [10], a compiler hint is used to aid in keeping spatial-only[5] data from polluting the A cache. Blocks that have the spatial-only hint bit set are returned to the next level of memory upon eviction from the B cache. However, we do not model these compiler hints in this paper. As a result, the placement of data within the Assist cache is determined only by the program's reference pattern. This omission will be addressed when we discuss the resulting performance of the Assist cache implementation.

### 4.2. Victim cache

The Victim cache (Figure 2ii), based on the scheme proposed in [2], is a multi-lateral design where B is managed in a fashion akin to the victim buffer. On a memory access, desired data found in A is returned to the processor the next cycle. If the desired data is found in B, the desired block must first be "swapped" into A; the block from B is placed in A and the resulting evicted block is placed in B. Depending upon the amount of hardware dedicated to handling these swaps, a hit in B may require varying amounts of time. In this study, we assume an additional 2 cycle latency for B cache hits.

On a miss, the block entering L1 from the next level of memory is placed in A. A block evicted from A as the result of the new block's arrival is placed in B; dirty blocks evicted from B return to the next level of memory. Like the Assist cache, the Victim cache manages the cache state passively; recently evicted blocks are always saved in the auxiliary cache for faster access; no active placement decision is made based on reuse information.

The Victim cache can be implemented using the library routines provided. On a cache miss, a *do_save_evicted* is performed: the new block is placed in A and the evicted block is placed in B (after the specified access latencies). On a hit in B, a *do_swap* is done to place the referenced data in A, and the block it evicts from A is moved into B.

### 4.3. NTS cache

The NTS cache, using the model in [10], which was adapted from the scheme proposed in [3], actively places data within L1 based on each block's usage characteristics. In particular, blocks that have been found to exhibit temporal reuse are placed in A, while nontemporal data are sent to B. This is done in the hope of allowing temporal data to remain in the larger A cache for longer periods of time, while shorter lifetime nontemporal data can for a short while be quickly accessed from the small, but fully associative B cache.

On a memory access, if the desired data is found in either A or B, the data is returned to the processor with a 0 added latency, but the block remains in the cache in which it is found. On a miss, the block entering L1 is checked to see if it has an entry in a Detection Unit (DU). The DU contains temporality information about blocks recently evicted from L1. On eviction, a block is checked to see if it exhibited temporal reuse during its most recent tour in the L1 cache structure and is marked accordingly in the DU. If the new block address matches an entry in the DU, the tag of that entry is checked and the block is placed in A if it indicates temporal, and B if not. The reuse information is kept via a single bit, the T/NT bit. If no DU match is found, a new entry is created in the DU, the block is assumed to be temporal, and it is placed in the larger A cache.

*mlcache* does not provide routines for managing a DU. The user can add routines that manage the DU and perform placement decisions based on the results of those routines. We added routines that maintain the DU as a separate, finite-size fully-associative cache indexed by an address. Each entry in the DU contains an effective address and a T/NT bit (set to 1 if the block at that address exhibited temporal reuse during its most recent cache tour, and 0 if not). This T/NT bit is used to predict the temporality of the next tour of this block. These routines are called within the `config.c` file when a cache miss is handled and used to decide whether to place the data in the A or B cache. Since this is a user-added routine, we can specify separate parameters for this module. For our experiments, we use a DU size of 32 entries.

### 4.4. PCS cache

The PCS cache [10] decides on data placement based on the program counter value of the memory instruction causing the current miss, rather than on the effective address of the block as in the NTS cache. Thus, the performance of individual memory accessing instructions, rather than individual data blocks, is used to determine placement of data in the PCS scheme.

---

[5] Data is tagged spatial-only if it is predicted to be use-once or too large to be effectively cached.

| | |
|---|---|
| **Fetch Mechanism** | fetches up to 16 instructions in program order per cycle |
| **Instruction Cache** | perfect cache, 1 cycle latency |
| **Branch Predictor** | perfect branch prediction |
| **Issue Mechanism** | out-or-order issue of up to 16 operations per cycle, 256 entry re-order buffer (RUU), 128 entry load/store queue (LSQ); loads may execute when all prior store addresses are known |
| **Functional Units** | 16 integer ALUs, 16 FP ALUs, 8 integer MULT/DIV, 8 FP MULT/DIV, 4 L/S units |
| **Functional Unit Latency (total/issue)** | integer ALU:1/1, integer MULT:3/1, integer DIV:12/12, FP adder:2/1, FP MULT:4/1, FP DIV:12/12. load/store:1/1 |
| **Data Cache** | write-back, write-allocate, 32B lines, 4 read/write ports, non-blocking |

Table 3: Processor and memory characteristics.

The PCS cache structure modeled is as similar as possible to the NTS cache (Figure 2iii). The DU is indexed by the memory accessing instruction's program counter, but is updated in a similar manner to the NTS scheme. When a block is replaced, the temporality bit of its entry is set according to the block's reuse characteristics during its last tour of the cache. Thus, if that instruction subsequently misses, the loaded block is placed in B if the instruction's PC hits in the DU and the prediction bit indicates NT; otherwise the block is placed in A. If the instruction also misses in the DU, a new DU entry is created for this instruction and the data is placed in A.

Given the similarity of the NTS and PCS cache structures, we implemented the PCS cache using the routines that we created for the NTS cache, but simply indexed the DU by the PC instead of by the effective address of its data. As with the NTS cache, a 32-entry DU is used.

## 5. Implementations in *mlcache*

We implemented five different cache configurations using the *mlcache* simulator: a direct-mapped single structure cache, an Assist cache, a Victim cache, an NTS cache, and a PCS cache. The latencies used for the timing simulation of these caches are shown in Table 2.

## 6. Simulation environment

A timing simulation of caches is of limited use without considering the latency-masking effects of processor execution. Thus, we integrated *mlcache* into the *SimpleScalar* [9] *sim-outorder* simulator by replacing *SimpleScalar's* data cache module with *mlcache*. Note that *mlcache* can be combined with any currently available instruction-level simulator, including *SimICS* [8], *RCM_brisc* [7], and others. This flexibility 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 two caches plus a perfect memory. We chose the *sim-outorder* processor model because it performs out-of-order issue, execution, and completion on a derivative of the MIPS instruction set architecture, and is a well regarded processor simulator in general.

| **Program** | **Instruction Count (millions)** | **Memory References (millions)** | | **Perfect Cache Performance** | |
|---|---|---|---|---|---|
| | | **Loads** | **Stores** | **Cycle Count (millions)** | **IPC** |
| **SPEC95 Integer Benchmarks** | | | | | |
| **compress** | 35.68 | 7.37 | 5.99 | 5.35 | 6.6644 |
| **gcc** | 263.85 | 61.15 | 36.24 | 43.50 | 6.0648 |
| **go** | 548.13 | 115.79 | 41.40 | 91.33 | 6.0049 |
| **li** | 956.49 | 286.38 | 168.79 | 151.32 | 6.3210 |
| **perl** | 1,500.00 | 396.82 | 269.83 | 232.89 | 6.4408 |
| **SPEC95 Floating Point Benchmarks** | | | | | |
| **hydro2d** | 974.50 | 196.11 | 60.90 | 127.63 | 7.6353 |
| **su2cor** | 1,054.09 | 262.20 | 84.74 | 152.34 | 6.9192 |
| **swim** | 849.92 | 205.18 | 58.44 | 113.02 | 7.5201 |
| **wave5** | 1,500.00 | 321.87 | 133.26 | 318.69 | 4.7067 |

Table 4: The nine benchmarks and their memory characteristics. IPC is Instructions Completed Per Cycle.

Since our study focuses on the effectiveness of a given data cache configuration in reducing memory access time, we designed a near perfect instruction supply mechanism for our processor simulator. In addition, we provided ample resources for the instruction processing phase in order to maximize the effect of data cache performance. Table 3 details our chosen parameters and architectural assumptions. Since 16-way superscalar processors are likely in the near future, this processor configuration will shed light on the benefits of using multi-lateral caches in processor designs of coming generations.

### 6.1. Benchmarks

Table 4 shows the 9 programs (5 integer and 4 floating point) selected from the SPEC95 benchmark suite for this study. Each program, using the training data sets, was run to completion or through the first 1.5 billion instructions.

### 6.2. Experiments

*mlcache* can evaluate a large range of cache designs simply by changing cache parameters; Table 5 lists the available parameters. In order to present a cache configuration evaluation using *mlcache*, we chose to keep some of these parameters constant. The multi-lateral caches are all of the same size and configuration: A is 8K and direct-mapped and B is 1K and fully associative. While these caches may be small by today's standards, they are conve-

| | |
|---|---|
| **Sizes** | cache size; block size; word size |
| **Widths** | CPU-to-cache bus width; cache-to-memory bus width |
| **Cache Management** | associativity; replacement policy; return policy (requested word first/first subblock first); NOA |
| **Ports** | number of read ports, write ports, read/write ports |
| **Latencies** | read miss latency; write miss latency; hit latency |

Table 5: A listing of the parameters in *mlcache*. The parameters can be changed for each simulation run without recompiling the simulator.

nient for illustrative purposes. Previous proposed/evaluated multi-lateral cache designs were small, as their performance was found to rival the performance of direct-mapped caches of twice the size. Further, these cache sizes are driven by our use of the SPEC benchmarks; larger caches can hold the working set of these benchmarks, reducing the need for multi-lateral designs [14].

# 7. Results

## 7.1. Miss ratio

Miss ratio is a first-order performance metric typically used to describe cache performance. However, incorporation of latencies introduces three categories of memory accesses: 1) hits to the cache, 2) misses to the cache, and 3) delayed hits. Hits (misses) to the cache are defined simply as accesses to data that are resident (not resident) in the cache at the time of access. The third category, delayed hits, is a refinement of category 2. Delayed hits, typically categorized as hits in behavioral cache simulators, do not cause any additional memory traffic. However, due to latency-adding effects, delayed hits typically experience a greater latency than the nominal hit latency.

*mlcache* distinguishes between these three access categories and reports the ratio of the number of accesses in each category. Figure 3 shows the miss and delayed hit ratios for three benchmarks that show the range of impact of delayed hits: delayed hits are most prominent in **go**, misses in **hydro2d**, and **compress** is in between these two.

The average delayed read and write hit latencies in these experiments were 9.89 cycles for read and 11.17 cycles for write (vs. 1 for hit, 18 for miss)[6]. Behavioral cache simulators typically count delayed hits as cache hits, and thus report the same performance for configurations exhibiting the same miss ratio, despite the number of delayed hits seen. In actuality, of two configurations with the same miss ratio, the one with the lower delayed hit ratio would perform better.

[6] In *mlcache*, read latency ends when the data is at the processor; write latency when the data is written in cache, freeing that block for later accesses.

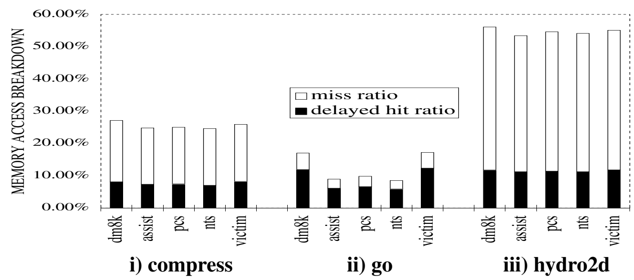


Figure 3: Memory access breakdown for the different cache configurations for three benchmarks. i) **compress** (delayed hits contribute moderately to the number of accesses that miss in the cache) ii) **go** (delayed hits are prominent) and iii) **hydro2d** (where misses are dominant).

## 7.2. Relative Cache Effect Ratio (RCR)

Hit/miss ratios, or even delayed hits, are not the best metric by which to evaluate a cache configuration's performance when latencies can be accounted for in simulation. A better metric for highlighting performance gains is the Relative Cache Effect Ratio, as defined in [4], which is:

$$\mathrm{RCR}_X = \frac{\text{Cycle Count}_X - \text{Cycle Count}_{\textit{Perfect Cache}}}{\text{Cycle Count}_{\textit{base}} - \text{Cycle Count}_{\textit{Perfect Cache}}}$$

This ratio provides the finite cache penalty of a given cache configuration *X* relative to the penalty of a specified base cache. The base cache has an RCR of 1, caches that perform better than the base have RCR between zero and one, and those that perform worse have RCR > 1. The RCR thus more accurately reflects a cache configuration's actual contribution to total system performance when running a target application. Table 4 shows the number of cycles required to execute the benchmark codes on our SimpleScalar processor configuration with a perfect cache.

The RCR for each of the cache configurations, using the direct-mapped 8K single cache as the base, is shown in Figure 4. While performance varies among the various configurations, several trends emerge. First, multi-lateral designs can significantly reduce the cache penalty. For **go** and **perl**, multi-lateral designs can eliminate over half the total finite cache penalty of the base 8K direct-mapped cache. Further, the multi-lateral designs often approach, or even exceed, the performance of a larger, 16K direct-mapped cache; the 16K cache performs best only in **compress**, **gcc**, **go**, **hydro2d**, and **li**, where the added capacity of the larger cache is more beneficial than the improved cache management provided by the multi-lateral schemes.

The Assist cache used in the HP PA-7200 made use of a compiler-supplied hint. Though such hints are not implemented here, the Assist, NTS, and PCS caches could all benefit from compiler hints regarding proper management for particular cache blocks, e.g. NTS and PCS could use compiler hints to place data for which no DU entry exists

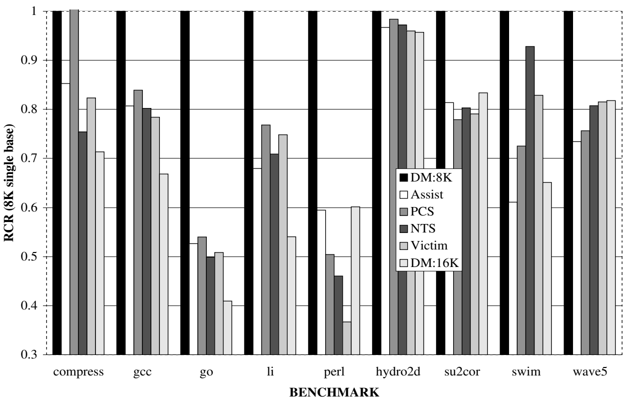


Figure 4: Relative Cache Effect Ratio comparison for the two direct-mapped and four multi-lateral configurations.

into B, rather than A by default; if the compiler knew that the accessed data was nontemporal, this hint could reduce the pollution of A with nontemporal data. Thus, it is likely that with compiler assistance, the performance of the Assist, NTS, and PCS caches would each improve over what these experiments have shown.

## 7.3. Block tour information

In general, reducing the number of tours through the cache for a given benchmark results in improved overall performance; fewer tours indicate that data is used more during a tour, resulting in fewer fetches to the next level of memory. *mlcache* provides statistics for block tours to each cache. For configurations that do not transfer data directly between A and B (i.e. single, NTS, and PCS caches) the number of block tours corresponds to the miss ratio. However, for Assist and Victim, which allow data movement between the cache, the number of tours also counts the number of times that a block is moved between A and B.

To more accurately explain the performance of the Assist and Victim caches with regard to block tours, we must also account for the number of tours caused by data movement between the caches. These tours are not as costly as accesses to the next level of memory. We can subtract the number of tours due to swaps and migrations (saves) from the total tours to obtain L1 tours (i.e. the number of fetches from the next level of memory).

For three cache configurations, an 8K single structure cache, an NTS cache, and a Victim cache, Table 6 shows the tour information for two benchmarks, **go** and **hydro2d**, with different performance when using multi-lateral caches. For **go**, the multi-lateral designs result in large performance improvements over the base 8K cache, with RCR and L1 tours at approximately half those of the base cache

Note that though the Victim cache incurs the fewest L1 tours of the three configurations running **go**, its RCR is

| | RCR | Total Tours | Swaps | Saves | L1 Tours | Total % of References to each Tour Group | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | NTNS | NTS | TNS | TS |
| **D** | 1.00 | 16.1M | – | – | 16.1M | 3.91 | 1.95 | 24.0 | 70.1 |
| **N** | 0.49 | 8.1M | – | – | 8.1M | 1.41 | 1.05 | 17.8 | 79.8 |
| **V** | 0.51 | 23.9M | 9.9M | 7.0M | 7.0M | 9.14 | 1.84 | 22.7 | 66.3 |

| | RCR | Total Tours | Swaps | Saves | L1 Tours | Total % of References to each Tour Group | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | NTNS | NTS | TNS | TS |
| **D** | 1.00 | 28.7M | – | – | 28.7M | 0.13 | 29.4 | 6.8 | 63.7 |
| **N** | 0.97 | 27.4M | – | – | 27.4M | 0.03 | 28.8 | 4.9 | 66.2 |
| **V** | 0.96 | 56.3M | 1.4M | 27.4M | 27.5M | 10.2 | 26.3 | 6.1 | 57.3 |

Table 6: Block tour information for three cache configurations running **go** (top) and **hydro2d** (bottom). **D** is the 8K direct-mapped single structure cache, **N** is the NTS cache, and **V** is the Victim cache.

higher than that of NTS. This is due to the added two cycle latency for each swap — though the Victim cache has 1.1M fewer L1 block tours than the NTS cache, each of the 9.9M swaps adds two cycles to the access latency, and potentially to the cycle count. If the latency of the swap is reduced, the Victim cache's performance will improve.

For **hydro2d**, the performance of the three configurations is much closer, with the RCRs differing by less than 5%. This performance similarity is also reflected in the number of L1 tours experienced by each configuration, approximately 28M each. Though the NTS cache has slightly fewer L1 tours, the RCR of the Victim cache is slightly lower.

## 7.4. Block reuse information

*mlcache* also provides reuse information for each block tour. Block usage can be broken into four categories: 1) nontemporal nonspatial (NTNS), 2) nontemporal spatial (NTS), 3) temporal nonspatial (TNS), and 4) temporal spatial (TS). Good cache configurations should result in fewer tours and a higher percentage of data references to blocks making TS tours. NTNS and NTS tours are problematic; they may cause excessive cache pollution, and should be minimized if possible.

We see in Table 6 that the NTS cache does a very good job of managing the cache state, resulting in a high percentage of TS tours in **go**. Furthermore, the NTS cache reduces the percentage of references to NTNS and NTS data relative to the base cache, which also contributes to its performance. The Victim cache's reuse information is less indicative of its performance. Blocks in a Victim cache may experience many different tours while still remaining in the L1 structure. Thus, though Victim has a lower percentage of TS tours and a higher percentage of NTNS tours compared to the base cache, its performance is still better than the base cache. However, the Victim cache manages its state passively, so block reuse information is more of a statistic than an aid to cache management, as in the NTS scheme.

The lack of performance improvement of the multi-lateral designs running **hydro2d** can also be explained by the high percentage of nontemporal (NTNS + NTS) accesses that each configuration experiences. Data in these categories often pollute the cache, as they are not reused with high frequency and often evict more useful, temporal data. This information can be used to design cache configurations and management schemes than can help attain higher performance when running benchmarks like **hydro2d**.

Thus, block tour and reuse information can be used to evaluate and explain the performance of proposed caches running target applications. Furthermore, the statistics provided by *mlcache* can help in deriving new cache designs that can better handle the reference patterns of target workloads.

## 8. Conclusion

*mlcache* is a flexible, multi-lateral cache simulator developed to help designers in the middle of the design cycle make cache configuration decisions that would best aid in attaining the desired performance goals of the target processor. *mlcache* is an event-driven, timing-sensitive simulator based on the Latency Effects cache timing model. It can be easily configured to model various multi-lateral cache configurations by using its library of cache state and data movement routines. The simulator can be easily joined to a wide range of event-driven processor simulators. For this study, we integrated *mlcache* into the *SimpleScalar sim-outorder* processor simulator.

We showed implementations of five different cache configurations and their resulting performance when running nine of the SPEC95 benchmarks. These configurations included a direct-mapped single structure cache and four multi-lateral caches: an Assist cache, a Victim cache, an NTS cache, and a PCS cache. Each was easily modeled in *mlcache* by using the library routines provided, plus a few user-added status routines.

*mlcache* provides many statistics that can help explain the performance of proposed cache configurations when running target workloads. Hit, miss, and delayed hit ratios reveal each program's memory access characteristics for these caches, while block tour and reuse information say more about the actual data usage within each program. These statistics can be used to explain the performance of each cache configuration as well as help to drive the development of future cache designs that better handle the reference streams presented by future processors when running the target workloads.

## 9. Acknowledgments

This research was supported in part by a gift from IBM. The simulation facility was provided through an Intel Technology for Education 2000 grant.

## Availability

The *mlcache* tool is available on the web at http://www.eecs.umich.edu/~estam/mlcache.

## References

[1] E. Rashid et al, "A CMOS RISC CPU with On-Chip Parallel Cache," *ISSCC Digest of Papers*, February 1994, pp. 210-211.

[2] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of ISCA-17*, Los Alamitos, CA, May 1990, pp. 364-373.

[3] 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. 154 - 163.

[4] J. A. Rivers, E. S. Tam, and E. S. Davidson, "On Effective Data Supply for Multi-Issue Processors," *Proceedings of the 1997 ICCD*, October 1997, pp. 519-528.

[5] E. S. Tam, J.A. Rivers, and E. S. Davidson, "Flexible Timing Simulation of Multiple Cache Configurations," *Technical Report CSE-TR-348-97,* University of Michigan, November 1997.

[6] 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.

[7] 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.

[8] P. Magnusson and B. Werner, "Efficient Memory Simulation in SimICS," *Proceedings of the 28th Annual Simulation Symposium*, April, 1995, pp. 62-73.

[9] D. Burger and T. M. Austin, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," *Technical Report #1342*, University of Wisconsin, June 1997.

[10] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens, "Utilizing Reuse Information in Data Cache Management," *Proceedings of the 1998 ICS,* July, 1998.

[11] G. Kurpanchek et al, "PA-7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface." *COMPCON Digest of Papers*, February 1994, pp. 375-382.

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

[13] J-L. Baer and W-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of ISCA-15*, May 1988, pp. 73-80.

[14] M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark suite," *IBM J. R&D*, Vol. 41 Number 3, May 1997, pp. 265-286