RC23998 (W0607-023) July 12, 2006
Computer Science

IBM Research Report

Software/Hardware Co-managed Cache Optimizations

Rajiv Ravindran, Krishnan Kailas, Zehra Sura
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Software/Hardware Co-managed Cache Optimizations

Rajiv Ravindran, Krishnan Kailas, Zehra Sura

Abstract

Caches are traditionally managed by relatively simple hardware based replacement and placement schemes. Prior work has shown that such history based schemes are not optimal across a range of applications. Ideally, we would like to tailor the caching strategy for each individual application, but doing this in hardware alone is difficult because of the cost of added hardware complexity and the overhead of dynamic discovery of application characteristics. Instead, software can be used to determine the caching strategy by analyzing the application code and to provide hints to the hardware. In this paper, we introduce two such hardware/software assisted cache management techniques. These techniques improve upon prior work in that they better balance the flexibility in using available cache space with the hardware complexity. In our *priority-based replacement* scheme, software assigns a relative priority for selected memory reference instructions, and hardware uses the priority information to make better replacement decisions. In our *virtual cache partitioning* technique, software assigns selected memory reference instructions to certain virtual cache partitions to reduce conflict misses due to address based fixed mapping of data into cache arrays. For both these cache management mechanisms, we describe heuristics that the software can use to determine what cache strategy will be most effective for caching during the execution of a given application. Our preliminary experimental results show that the priority-based replacement can reduce the miss rate up to 7.1% (with software assigned priorities for approximately 1% of load/store instructions), and the virtual partitioning scheme can reduce the miss rate up to 5.6% (with software assigned partitions for less than 10 instructions in the application).

1 Introduction

There has been an ever increasing performance gap between the processor and the memory system. This gap, caused by the inability of the memory system to supply data at the rate at which it is consumed by the processor core in spite of using multiple levels of on-chip caches, has been the major bottleneck for single-thread performance in modern day processors. Although these processors employ a suite of sophisticated microarchitectural techniques for hiding cache misses to

improve the performance, much of the benefits have been incremental, thus clearly indicating the need for more efficient management of the cache hierarchy.

Caches have traditionally been managed by the hardware. The advantage of using a hardware managed cache is that the architecture and functionality of the cache remains transparent to the programmer and compiler. This helps the architect in varying the cache organization over different processor generations. Another major advantage of a hardware managed cache is that the management and access policy is general enough to handle any arbitrary data access patterns, be it strided, random, pointer, etc. But this layer of abstraction comes at a price. Hardware managed caches have been designed to support a very wide spectrum of workloads; hence their design tends to be very generic. In particular, hardware employs simple but greedy replacement algorithms like pseudo-LRU (Least Recently Used) for cache placement. In addition, set placement decisions are made based on selected bits from the physical address to index into a given set of a set-associative cache. Although such greedy/random replacement decisions provide reasonably high hit-rates at low cost, they have been proven to be ineffective for high memory bound workloads. For selected workloads, recent research has shown that other replacement schemes such as the Most Frequently Used (MFU) [1, 2] or generational replacement [3] have promising benefits. But, different applications tend to have differing temporal and spatial reuse properties and hence it is impractical or too expensive to incorporate any specific policies in the hardware. Moreover, even within an application, there tends to be different phases exhibiting different data reuse requirements. In general, hardware schemes are found to be incapable of exploiting such application specific behaviors to close the processor-memory performance gap.

Recently, researchers have suggested software managed caches or scratch-pads [4, 5, 6] to allow better data management. Here, the burden is on the programmer or the compiler to orchestrate data movement into these scratch-pads. Hardware tags are avoided while providing DMA support for efficient data transfer. But, software-only techniques tend to make conservative decisions to ensure correctness. Moreover, they are limited to analyzing programs with highly constrained code and memory access behavior. They limit the analysis to array-only code that is indexed through affine functions [4]. Pointer-based applications are particularly hard to analyze and may require dynamic memory disambiguation. The same is true for recursive function calls where there are multiple instances of local arrays and variables. None of the existing solutions handle these cases well [5]. To simplify the problem, they restrict their analysis to inner-most loops with no function calls or control-flow [6].

Software-managed techniques have to orchestrate the movement of data into and out of the on-chip scratch-pads so as to better utilize the limited available area. But in the process, they have

to conservatively insert data transfer instructions to dynamically copy data into the scratch-pads. Some of these transfers may be unnecessary in certain paths of execution in the program. For example, the control flow during execution sometimes causes the data to be already present in the scratch-pad. But for correctness and to maintain coherency, the code needs to ensure that the data for each access does exist in the scratch-pad cache, and copy it if it does not. Moreover, when data is copied to the scratch-pad, any references to the stale data in memory needs to be updated. This has to be done during eviction too. Hardware tags achieve this with minimal performance overhead.

In this paper, we propose two synergistic software controlled hardware assisted cache management techniques to efficiently use data caches. Our schemes try to combine the advantages of both the hardware and software based cache management schemes by relying on the software to convey additional information to the cache controller for making better cache management decisions. Software based management can help reduce hardware inefficiencies using global knowledge of the program behavior. Hardware assistance can help reduce software overheads while capturing dynamic program behavior and thus aids in making software techniques more aggressive and effective. The proposed *priority based replacement* scheme uses the knowledge of the memory reference patterns in the programs to specify and change the relative priority of the cache lines using software to make better cache line replacement decisions than the traditional hardware based schemes. The proposed *virtual cache partitioning* scheme tries to reduce cache misses via confining the conflicting address references into separate logical cache partitions instead of traditional address based fixed mapping of data into cache arrays.

The rest of the paper is organized as follows. Section 2 describes the basic idea behind a hardware/software co-managed cache. Section 3 describes two basic software driven cache hierarchy management schemes - priority based replacement and partitioning. Initial evaluation of these schemes is presented in Section 4. Section 5 briefly surveys the related work. Finally, we conclude in Section 6.

2 Hardware/Software Co-Managed Cache

In this work, we propose a new cache mechanism where the cache is managed by both hardware and software. Caches perform two basic activities.

- Checking if the referenced data is present in the cache,
- On a miss, decide if the requested data needs to be allocated and where in the cache it is to be allocated.

Traditionally, both these decisions are performed by the hardware. Hardware provides fast and efficient checking for the presence of the referenced data. The primary advantage of hardware is that it allows support for data accesses of any type. In particular, pointer accesses are hard to disambiguate statically. Hence, run-time schemes provide a fast and generic scheme to locate the data. Hardware performs the checking using tags which are maintained in the tag-directory structure. Emulating this in software is time-consuming. Since hits are in the critical path, it is best to do this as fast as possible. In our scheme, we allow the hardware to do the checking.

But, hardware usually employs a single very conservative algorithm to make allocation and replacement decisions. Implementing a more flexible replacement policy entirely in hardware is usually expensive. Here, software can do a better job as it can employ more intelligent heuristics to make replacement decisions at considerably lower costs. Also, software has the added advantage of analyzing the application's future behavior by profiling the application on a representative input set. The profile based memory access information collected can be later used to make replacement decisions. Since misses involve access to the slower next level memories (mostly off-chip) in the cache hierarchy, employing a more expensive (in terms of run-time) replacement algorithm in software will have less impact on the performance. More importantly, making good replacement decisions can help in reducing the miss rate and thus, improve the overall performance.

Within the software managed replacement schemes, there are two possible alternatives. One is that software has complete control and decides what blocks are to be replaced, where to allocate a block, and so on. This can be implemented using a cache miss-handler routine that gets invoked on a cache miss. This routine can maintain internal states to implement its own replacement policy. The tag-directory can be exposed as a memory-mapped region which can then be accessed by this handler routine. By relegating the responsibility to software, more state and complexity can be incorporated into the replacement decision making, which may be too expensive to be implemented in hardware. Also, this replacement handler can be customized for different applications. The hardware will only be responsible for maintaining the tag directory for the tags so as to check if a data item is present and if not, invoke the miss handler.

An alternate approach is to let the hardware make the replacement decisions as is done currently, but additionally allow the software to provide hints to the hardware to make better replacement decisions. Examples of such hints include marking what blocks need to be evicted, what blocks are to be fetched in or bypassed (i.e., accessing a datum without caching it), etc. This has much lower overhead as compared to the all-software approach. What the hardware needs to do in this case is to maintain some extra state efficiently which can be managed intelligently by the software. This state has to be general enough so that different software management policies can be applied

without getting tied to a particular policy.

3 Hardware/Software Managed Replacement Schemes

In the following sections, we propose two novel hardware/software co-managed caching schemes.

3.1 Software-specified priority based replacement

Replacement decisions are crucial in maintaining high hit-rates. Recent research [7] has shown that there is a significant gap between the currently employed LRU and an optimal replacement policy. LRU is based on the assumption that a block, if referenced, will very likely be used soon and hence the need to retain it, while if a block is not referenced for a while, it is highly likely that it will not be used in the future and so is replaced. This decision is based solely on the history of the program and incorporates no future information. This scheme can fail for several reasons. For example, a block that is currently accessed may not have any temporal behavior. But this block, anticipating a future reuse, is still brought into the cache which then replaces another block that may have a reuse in the near future. Another possibility is that a block can have frequent reuses, but the reuses are separated far apart in time. But the LRU policy may replace a block that has a nearer reuse than one with a farther reuse. Yet another scenario is when the block is not being referenced now, but has a high reuse in the near future. But, the hardware will prematurely evict the block anticipating no future reuse.

In this work, instead of always evicting the block at the LRU position, we assign a certain priority level to each block. A block with a higher priority level always replaces a block with a lower priority in the same set. If all blocks in a given set are of higher or equal priority than the new block that is brought in, the default LRU scheme is used to replace the block at the LRU-position irrespective of its priority. As a special case, the hardware can ignore blocks with the least priority, i.e., instead of fetching the block into the cache, it is bypassed, thus preventing cache pollution.

Figure 1 shows an example code sequence illustrating the benefits of the priority-based scheme. The array C is used in the loops with loop indices j and k. If the cache size is such that it can hold only two of the arrays (assume all arrays are of equal size), then during the second inner loop, hardware can potentially replace C with B, although C is getting reused. In the priority-based scheme, the compiler analyzes the code and discovers the temporal reuse of the array C. The blocks constituting this array are then marked as high priority. This prevents C from getting displaced in the second loop, thus reducing the miss rate.

Ideally, priorities need to be applied to the blocks as replacement decisions are made for each

```
for (i = 0; i < 1000; i++) {
  ...
  for (j = 0; j < 1000; j++) {
    sum1 += A[j] * C[j];
  }
  for (k = 0; k < 1000; k++) {
    sum1 += B[k] * C[k];
  }
}
```

Figure 1: Example code showing benefits of priority scheme

individual block. But, since the memory blocks that a program accesses can vary for every run of the program, we instead assign priorities to the load or store instructions that access these blocks. Thus, the priority of a block in turn is the priority of the load or store instruction that fetches the block. When a block is fetched into the cache or accessed during a cache hit, its priority is recorded within a separate field in the tag-directory. For a block that is fetched on a miss, if no free entries are found, its priority (which is again the same as the priority of the load or store instruction that accesses the block) is compared against the priority of all the blocks in the set. If the lowest block priority is less than the priority of the load or store instruction, the corresponding block is selected as the candidate for replacement. If no such block is found, the default block at the LRU position is replaced.

The responsibility of the hardware in this case is to maintain the priorities in the tag directory and compare the priorities while making replacement decisions. The software (compiler), on the other hand, decides the priorities of the load/store instructions based on prior analysis of the program. A possible heuristic the compiler can employ to assign priorities to load/store instructions is to incorporate the reuse characteristics of the program. References with high number of short reuses (reuses that are closer to each other in time) are assigned a higher priority than those with longer or fewer reuses. This procedure attempts to approximate the optimal replacement policy where the blocks with the farthest reuse are selected for replacement. Irrespective of what the heuristics are, the output of the compiler is just an annotation of the priorities of each load and store instruction. It should be noted that the compiler does not need to annotate all load and store instructions. Instead, it annotates only those that are frequently missing or are likely to be in the critical path of the program. Also, if none of the load/store instructions are annotated, then the scheme reduces to the default LRU policy implemented by the hardware, as all blocks now have the same priority (null priority).

The annotations can be modeled as extended load and store opcodes. For example, four different variations of load and store instructions can be used where each such load/store corresponds to a distinct priority level. Alternately, a special load or store instruction that takes an additional immediate or register operand to contain the priority of the instruction can be added to the ISA. For register operands, the compiler then inserts instructions to assign priority values to these registers prior to their use in the following load or store instruction.

The priorities need not be purely static values. The compiler, based on the phase/context behavior of the program can adjust the priority values of the memory reference instructions. In addition, after the last use of certain data items, the compiler can insert additional instructions that lowers the priorities of the corresponding blocks in the cache. This can help prevent higher priority, not-in-use data elements from occupying the cache space.

input : $\forall Block_i,\ NumShortRefs(Block_i)$ & $NumLongRefs(Block_i)$,
$Blocks \rightarrow \{Set\ of\ all\ Blocks\}$,
$MemRefs \rightarrow \{Set\ of\ All\ Loads/Stores\}$,
$\forall MemRef_i,\ BlocksInMemRef(MemRef_i)$

output: $\forall MemRef_i$, Priority($MemRef_i$)

begin
  foreach $Block_i \in Blocks$ **do**
    Priority($Block_i$) $\leftarrow$ Weight(ShortRefs) * NumShortRefs($Block_i$) +
    Weight(LongRefs) * NumLongRefs($Block_i$);
  foreach $MemRef_i \in MemRefs$ **do**
    Priority($MemRef_i$) $\leftarrow \sum Priority(Block_j), Block_j \in BlocksInMemRef(MemRef_i)$
end

Algorithm 1: Heuristic to assign priorities to load/store instructions

A simple heuristic to assign priorities to load/store instructions is shown in Algorithm 1. Initially, the application is profiled using a cache simulator (performance monitoring counters may also be used) to identify load/store instructions that cause the highest number of cache misses. In addition, all the cache blocks that are accessed during the profile run are recorded. For each such block, two reuse metrics are computed - $NumShortRefs$ and $NumLongRefs$. $NumShortRefs$ is the frequency of short references to the block, where a reference is defined as 'short' when two consecutive references to the block occurs within $10,000$ execution cycles. Similarly, $NumLongRefs$ is the frequency of long references to the block, where a 'long' reference is when two consecutive references to the block are separated by a million cycles. This gives an approximation of the tem-

poral behavior of the data accessed within the application. For each memory reference instruction, $MemRef_i$, all blocks accessed by that instruction are recorded in $BlocksInMemRef$. The heuristic initially computes a priority value for each block. The number of short and long references to each block is multiplied by a weighing factor to normalize the reference count. To capture temporal locality, short references are usually given higher weight than long references. Once the priority of each block is computed, then for each memory reference instruction, its priority is calculated by summing up the priorities of each block that is accessed by that instruction. It should be noted that this is just a heuristic, where the final output is a priority number assigned to each load/store instruction. The priorities are later normalized to a smaller value that can be encoded in an instruction. Note that priorities need only be computed for instructions that will have a significant impact on cache performance, as determined using profiling information.

3.2 Virtually partitioned cache

In traditional caches, a block placement decision is performed based on two factors - address of the block and the replacement policy (eg. pseudo LRU). The address of the block decides which set in the cache to place the block in, while the replacement policy decides the way within the set. One of the primary disadvantages of a set-based replacement policy is that the hardware decides the set purely based on the physical or virtual address of the block. Although this does simplify the hardware and the checking process, this can seriously affect performance. Data items present at different locations in the memory, but with the same set address gets mapped to the same set resulting in conflict misses. Increasing the associativity is one possible way of reducing such conflict misses. But if two load or store instructions have a large footprint, then mapping them onto the same set can still cause high conflicts even with higher associativity, as the number of ways are not large enough to hold the working set of both the instructions. Full associativity may be the only option here.

In large benchmarks, one of the primary causes of misses is due to the presence of certain load or store instructions with large data footprint and very long data reuse. Some of these loads tend to reuse their data set over a long period of time, possibly exceeding millions of cycles. Thus, the primary contributor towards these misses is capacity misses since a very large cache is needed to retain the working set so as to reuse this data set. Even with large sized caches (around 1 or 2 MB), these misses still tend to be significant. The only solution seems to be is to use extremely large caches to capture the large working set. This clearly is an inefficient use of cache memory. It should be noted that since the reuse distance for such data items are large, the number of unique references to other data items between two occurrences of such a data item is very large too. Thus,

unless the cache is large enough to retain all such unique references, for most moderately sized caches, this data item is guaranteed to create a miss. The hardware based cache management schemes tend to cache data items assuming that they will be used in the near future. Since it has no notion of future reference behavior, it is unable to predict such long reuses. Thus, not only does this data item miss due to its long reuse, it can potentially evict more important blocks that have a shorter reuse and hence should have been retained in the cache.

In order to solve both the above problems, we propose a new virtually partitioned cache. Conceptually, the basic idea is to maintain multiple (4 to 8) smaller caches instead of a unified cache. This partitioning can be done logically without actually maintaining multiple distinct physical caches. The data items can then be assigned separately to each of these partitions so as to reduce conflicts. In addition, for data sets with long reuse, by restricting them to a certain partition, it is less likely that they will replace other more useful data items which are assigned to other partitions. The long reuse data set can then potentially replace itself out more than replacing others. This behavior may be acceptable, as the data set has a long reuse and will not benefit from the cache in any case, unless an extremely large cache or some form of oracle prefetching is used.

As is the case with the assignment of priorities, since the data set of a program can vary for each run, it is impossible to tag which data item should be placed where. Instead, we tag the load or store instruction that accesses these data elements with the desired partition information. With every such interesting load or store instruction, a partition identifier or a bit-vector of partition identifiers is maintained. When such a load/store instruction is issued, every partition is first checked for the desired data item. On a hit, the data is fetched from that partition and forwarded to the processor. On a miss, the data is loaded only to those partitions that the referring load/store has been assigned to. This ensures that it only replaces data items from its own partition and not from the partition of other load or store instructions. It should be noted that to check for the presence of a memory block, all partitions needs to be probed. This is because, two load instructions assigned to two different partitions can possibly be sharing the same data. If both these loads are assigned to two different partitions, at run-time, they have to check in each others partition for the referenced block.

3.2.1 Partitioning Example

Figure 2 shows an example code sequence from the benchmark *adpcm* from the Mediabench [8] benchmark suite. The execution is dominated by a single function *adpcm_encode*, whose control-flow graph is shown in Figure 2(a). Out of the 11 load/store instructions, 6 of the important ones that were identified through profiling is shown. Four different cache configurations, each of size

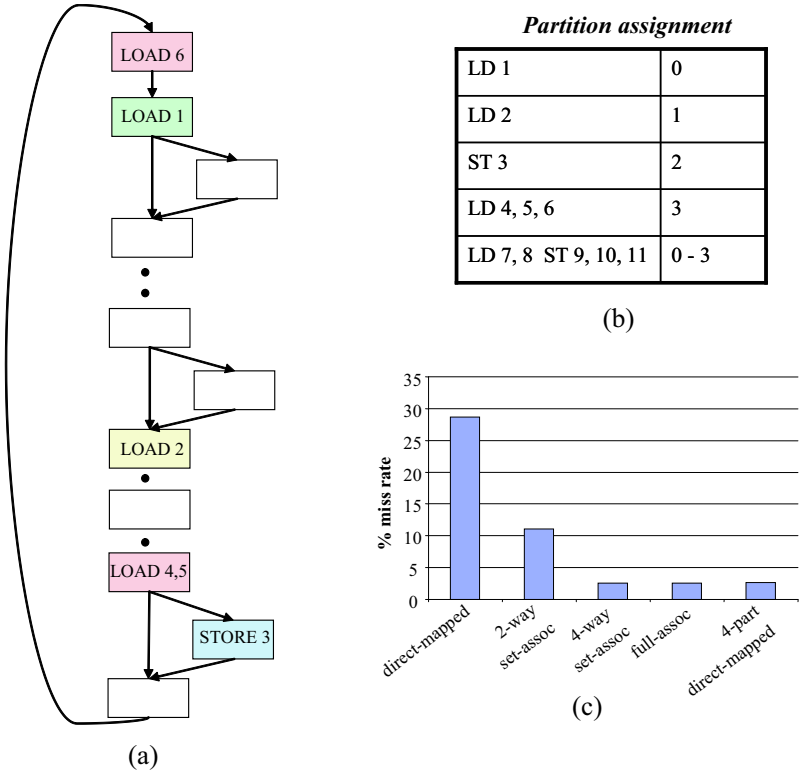

Figure 2: Partitioning example (a) Example code (b) Assignment of load/store instructions to partitions (c) Miss rates for different cache configurations.

256-bytes and line size of 32-bytes were selected. These configurations are (i) direct-mapped, (ii) 2-way set-associative, (iii) 4-way set-associative, and (iv) fully-associative. The miss-rates of each of these four configurations are shown in Figure 2(c). The direct-mapped configuration performed the worst, while the fully-associative configuration was the best. To compare against a partitioned architecture, a 4-way partitioned cache, where each cache is a 64-byte direct-mapped cache with a 32-byte line size was selected.

Table 1 shows the conflict information between the most frequent load/store instructions in the 256-byte direct-mapped cache. Each entry $memref_{ij}$, indicates how often did the i^{th} memory reference instruction miss due to its cache line being replaced by the j^{th} memory reference instruction. This information is gathered during the profiling phase. Since these load/store instructions conflicted each other often, they were partitioned as shown in Figure 2 so as to prevent them from replacing each other and thus, reduce the conflict misses. The unselected load/store instructions (7 to 11), were assigned to all partitions. Note that in a partitioned cache, each memory reference instruction will check in all the partitions for the referenced data, but on a miss, will only replace a block in the assigned partition. As seen in Figure 2(c), the partitioned cache almost equaled in performance to a fully-associate cache in spite of each partition being a smaller direct-mapped

% Conflicts	Load 1	Load 2	Store 3	Load 4
Load 1	5.4%	40%	12.8%	31%
Load 2	55.6%	0%	44%	0.3%
Store 3	27.9%	39.2%	0%	22%
Load 4	59.7%	0%	39.8%	0%

Table 1: Percentage conflicts between different load/store instructions in a 256-byte direct mapped cache for *adpcm*.

cache. The benefits are due to reductions in conflict misses observed in the unified direct-mapped cache.

3.2.2 Partitioning Heuristic

A simple compiler heuristic to partition load/store instructions is now described. Initially, the benchmark is profiled to identify the most frequent load/store instructions. In addition, all blocks accessed by these load/store instructions are recorded. For each pair of frequent load/store instructions, two metrics are identified: affinity and conflict information. The affinity between two memory reference instructions is defined as the number of common blocks that are accessed by these instructions within a specific time frame. If two instructions are accessing the same data set, assigning them to the same partition is useful as they can aid one another, each by prefetching data for the other instruction. The conflict information is used to identify pairs of load/store instructions that potentially displace each other and hence are likely candidates to be placed in different partitions.

Affinity computation: In order to compute the degree of affinity between pairs of load/store instructions, their temporal behavior needs to be identified. Two memory reference instructions display high affinity if they access a common set of blocks within the same program phase. For instance, consider an array of structs where each struct occupies a single cache line. Two loads accessing two different fields of the struct will refer to the same cache block. Assigning these two loads to the same cache partition can thus help in mutual prefetching and also reduce cache pollution in other partitions. The affinity metric can be computed during profiling by recording for every fixed time interval of say 1 million cycles, the number of common blocks accessed by every pair of load/store instructions and then summing them over all intervals. The affinity metric is indirectly affected by the data layout generated by the compiler or run-time system.

Conflict computation: The degree of conflict is computed during profiling by identifying for a given load/store instruction, how many times it gets replaced by another load/store instruction.

To do this, whenever a cache block is brought in, the load/store instruction that referenced the cache block is recorded along with the cache block. If the cache block that it replaced was valid, then the load/store instruction that fetched the cache block is defined to conflict with the load/store instruction that fetched the replaced cache block. The total number of such replacements for every load/store instruction pair measures the degree of conflict. This information is similar to that showed for the benchmark *adpcm* in Table 1.

Certain load/store instructions may have a large data footprint and a single cache partition may be insufficient to hold the data they access. In addition to the affinity and conflict metrics, each frequent load/store instruction's cache footprint is also recorded. The footprint is defined as the number of unique reused blocks accessed by that load/store instruction. Instructions with a large footprint may have to be assigned to multiple partitions. Each frequent memory reference instruction is virtually represented by multiple *memref_let*s. During partitioning, each *memref_let* is assigned a single cache partition. The number of such *memref_let*s for a memory reference instruction is given by the equation
$Footprint(mem\_ref\_instr)/sizeof(single\_cache\_partition)$, where $Footprint(mem\_ref\_instr)$, is the number of unique reused blocks accessed by that memory reference instruction. In the trivial case where the data footprint fits within a single cache partition, the corresponding memory reference instruction gets associated with a single *memref_let*. In order to assign a given memory reference instruction to multiple cache partitions, each of the constituent *memref_let*s have to be assigned to different cache partitions. The conflict and affinity information computed for a memory reference instruction is inherited by each of its constituent *memref_let*s. The *memref_let*s of a given memory reference instruction will have no affinity between themselves, but will have the highest degree of conflict, signifying that they should be assigned to different partitions.

The actual partitioning is formulated as a graph partitioning problem, where the nodes correspond to the *memref_let*s, while the edge weights measure the desire to be assigned to same/different partitions. The edge weight of the graph is computed based on the above mentioned affinity and conflict metrics such that instructions exhibiting a high degree of conflict are assigned to different partitions, while those with high affinity are assigned to the same partition. The edge weights between *memref_let*s of the same memory reference instruction are such that they always get assigned to different partitions. This allows a memory reference instruction with a large footprint to be assigned to different partitions. The instructions are then annotated with the respective partition identifiers. Again, it should be noted that only a few selected load/store instructions may need to be partitioned.

The conflict and affinity between a pair of load/store instructions can vary over the life-time of

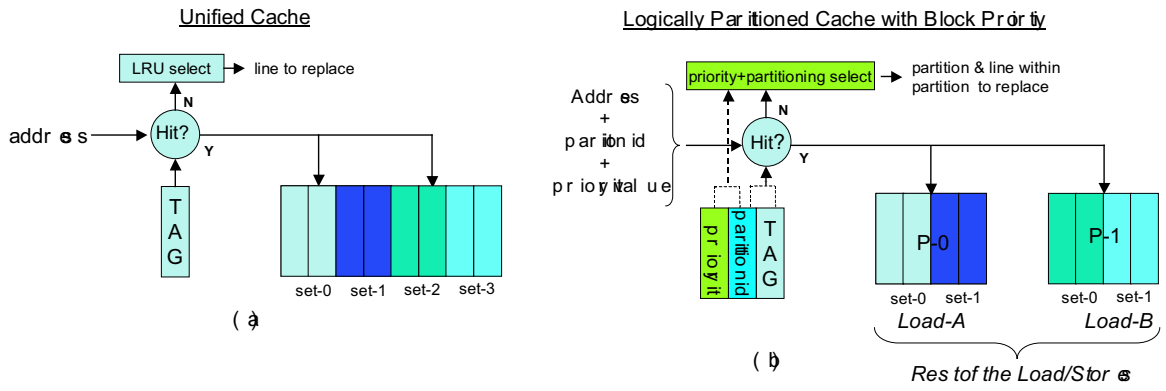

Figure 3: Partitioned Cache Architecture. (a) Traditional unified cache (b) Partitioned cache with priority figure

the instructions. In such cases, it is desirable to "split" an instruction into a number of instructions such that different *temporal* access patterns corresponding to the original instruction can be separated by spreading them among the multiple "split" instructions. To achieve this, we need to clone parts of the data flow graph such that a unique temporal access pattern of a load/store instruction is restricted to one of the clones. The cloning will result in multple copies of the original load/store instruction, and will enable us to statically assign different partitions to these instruction copies.

3.3 Hardware/Software Implementation

The data arrays of a traditional unified cache can be virtually partitioned by maintaining the partition information in the tag directory. The modified cache controller compares the tag as well as the partition identifier to determine whether an access is a hit or miss. The same fields from the physical/virtual address (set/tag) can be used to check for the presence of a data item in the partitions.

Each load/store instruction contains the partition identifier specifying which partition it has been assigned to. This can be implemented as either a special load/store instruction with a few (2-3) extra bits to identify the partition, or a separate prefix instruction specifying the partition of the next sequential load/store instruction. Alternately, special registers can be used as operands to load/store instructions to specify the partition information. The register based solution allows assigning the load or store to multiple partitions as each register will be a bit-vector specifying the assigned partitions. The registers can be initialized with a literal corresponding to the partition bit-vector using register move instructions. These move instructions are inserted prior to the load/store instruction that uses them. For small number of partitions, such a bit-vector can also be specified as an immediate operand of the load/store instruction.

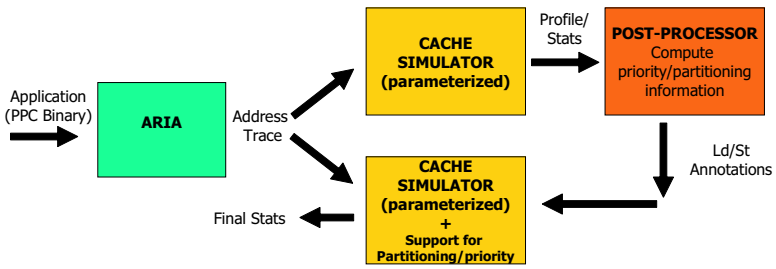

Figure 4: Experimental flow diagram.

Figure 3 shows the architecture of a partitioned cache. Here a unified set associative cache with 4-sets of 2-ways each (Figure 3(a)) is partitioned into two partitions of 2-sets each with each set consisting of 2-ways (Figure 3(b)). In a traditional unified cache, on a miss, the LRU replacement policy is used to select the line within the selected set. For a partition based cache, the tag directory is extended with the partition identifier as shown in Figure 3(b). To check for the presence of the referenced data, both the partitions are probed using the tag and set fields of the data address. On a miss, the partition identifier, which is specified as part of the corresponding memory reference instruction, is used to select the desired partition whose cache line/block is to be replaced. For a priority based replacement scheme (see Section 3.1), the priority value, which is again embedded within the tag directory, is used to select the line/block to be replaced. As shown, load instructions A and B are assigned to separate partitions, while the rest of the load/store instructions have been assigned to both the partitions.

The compiler need assign the partition information only to those load/store instructions that it believes will be useful, based on prior analysis of the application. The unassigned load/store instructions treat the partitioned cache as a traditional unified cache. So on a miss, the address of the block can be used to decide the placement. But to check for the presence of a data item, as mentioned previously, it still needs to check all partitions for the referenced block.

4 Experimental Results

We conducted some preliminary experiments on partitioning and priority based cache replacement policies on a few of the SPECint2000 benchmark suite [9]. We used a trace-driven, parameterized, cache simulator for the experimental evaluation. Memory reference traces of the benchmark programs were dynamically generated using Aria [10], an execution-driven trace generator for PowerPC processors. The replacement schemes were evaluated on the L2 cache. A 32Kb 4-way set associative L1 cache with 128-byte line size was selected. The L2 cache modeled had a total size of 1Mb with 128-byte line size and 8-way associativity. Only the data references were simulated.

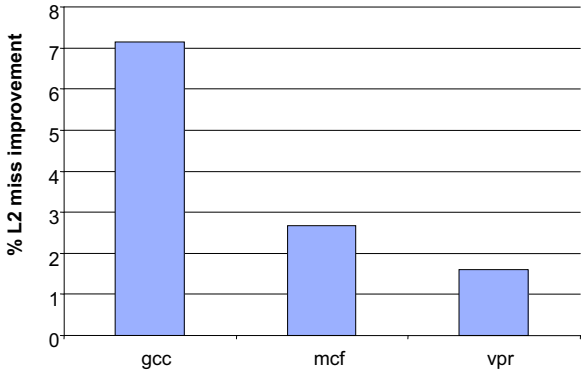

Figure 5: Priority based replacement results

No timing or bus related issues were modeled. This helped us study the first order effects due to these replacement algorithms without their effects getting masked by microarchitectural features that we don't optimize upon.

A top level flow diagram of the experimental setup is shown in Figure 4. Initially, the benchmarks were run on the 1Mb cache to collect some base statistics. The statistics collected include hit/miss ratios, most frequent executed load/stores, top missing load/store instructions, block access patterns etc. These statistics were used to evaluate both the partitioning and priority based replacement heuristics. For our experiments, the priorities for each load/store instruction was computed using the algorithm illustrated in Section 3.1. For the partitioned scheme, the partitioning was performed manually. The top missing/conflicting load/store instructions were identified during the profiling phase. Based on the conflict/affinity profile, they were then assigned to different partitions. Depending on the size of the data accessed, the load/store instruction could be assigned to multiple partitions. In the future, we plan to use the compiler directed partitioning algorithm, as illustrated in Section 3.2.2, for automatic partitioning. After the relevant load/store instructions were assigned partitions/priorities, the benchmark was re-run through the cache simulator with modifications added to support both partitioned caches and priority based replacement.

The L2 miss rate improvement for priority based replacement policy is shown in Figure 5. The base line is the unprioritized application with the default L2 cache implementing perfect LRU. For the priority based scheme, the annotated priorities were used while making replacement decisions. For each of the benchmarks, less than 1% of the load/store instructions were assigned priorities. The benchmark *gcc* performed the best with a 7% improvement in L2 miss rate.

The L2 miss rate improvement for partitioning based replacement policy is shown in Figure 6. The base line again is the default L2 cache with perfect LRU. For the partitioned caches, four 256-Kb partitioned caches with 128-byte line size and 8-way associativity each were used. The

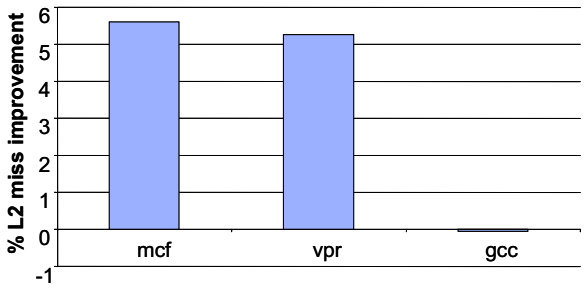

Figure 6: Partitioning based replacement results

partitioning was performed only for the top few missing load /store instructions (< 10). This is less than .1% of all static memory instructions. The benchmark *mcf* performed the best with a 5.8% improvement in L2 miss rate.

For both priority and partitioned based schemes, we only show results where we observed a non-trivial improvement in miss-rates. All the other benchmarks displayed only minor performance improvements. This, we believe, is due to the limitations of our heuristics and also due to the low L2 miss-rates of these applications and therefore, is not fundamental to our proposed strategies. It is important to note that the compiler can choose to not annotate the instructions with either partition or priority values, in which case, it defaults to the LRU replacement, thus, not affecting performance.

5 Related Work

A variety of hardware cache organizations [11, 12, 13, 14, 15] consisting of multiple/split caches aimed at storing data based on spatial, temporal, or a combination of access pattern behavior has been explored by researchers in the past. All of these schemes employ hardware techniques to dynamically classify memory blocks into each of the special caches (partitions). The cache controller is modified to detect the access pattern and route the data to the appropriate cache partition. Assist Cache [16] and Victim Cache [17] schemes also fall in this category in which a small fully associative cache next to a large direct-mapped cache is used to reduce conflict misses. The hardware moves the conflicting lines into the smaller cache.

The use of compile time classification of memory reference instructions into spatial, temporal, and spatial-temporal has been explored in [18]. The classified data references are then cached into three separate organizations. At run-time, the cache controller places data in a given cache depending on the instruction. Spatial and temporal caches are very small and fully associative, while the spatial-temporal cache is larger. Different block sizes are also used for each of the caches.

Hardware/compiler scheme is used in [19] to classify an instruction with the poorest miss rate as cacheable or non-cacheable based on the miss rate. In this scheme the cache is not split as in the other schemes.

The IBM PowerPC 440x5 processor [20] allows the software to configure the ways of the cache sets into 4 types of lines – normal, normal/transient, transient and locked. Each one of these line types may be considered to be a partition. Similarly, in [21], the ways (columns) of a cache are partitioned such that the replacement decisions are restricted to certain ways. A bit-vector is used to specify the allowable ways. Way-based partitioning is used to implement dedicated scratch-pad memory and also for reserving partitions in a multi-process system. One disadvantage is that the way-partitioning can reduce the associativity which could affect performance. In our case, we reduce the number of sets but retain the associativity.

A modified cache replacement scheme using the software to specify when to "kill" (evict) and "keep" a cache element for improving the LRU based replacement policy has been proposed in [22]. The scheme was further enhanced to combine prefetching with the software-assisted cache replacement in [23]. Kill Load/Store instructions are used by the software to set kill-bit associated with each cache line. The Kill Range instruction is used to kill parts of arrays or other data structures residing in cache. When an access falls within the range the associated cache lines get killed. A similar cache organization with an additional tag-bit called 'evict-me' is used in [24] to select the line to be evicted. Special load/store instructions are used to set the 'evict-me' bit in this scheme as well. An enhanced scheme is proposed in [25] in which compiler analysis is used to find array accesses with temporal and spatial reuse that the cache has sufficient capacity to retain and marks the first memory instruction with keep-me. The hardware evicts cache lines not marked with keep-me in LRU order. If all lines are keep-me it defaults to LRU. A keep-me counter is used to decay the keep-me bit to prevent it from monopolizing the cache.

Although our partitioned cache approach is similar in spirit to earlier work on split caches, our scheme is more flexible in that we allow variable number of partitions. In addition, instead of a dedicated hardware controller deciding on what data needs to reside in which partition, we use the compiler, which has a whole program knowledge, to make partitioning decisions.

Our priority based partitioning scheme is more general than the keep me/evict me schemes. By allowing different levels of priorities, we allow much broader replacement policies. Instead of removing everything that is marked 'kill' or retaining everything marked 'keep', we can selectively retain or remove cache lines using a range of priorities. In addition, priorities can be used to mark certain data elements with low reuse as uncachable.

A number of partitioned cache schemes [26, 27] were proposed in the past. The partition

cache presented in [26] differs from the scheme we proposed in several aspects. Their hardware scheme does not handle coherency issues, where as we use parallel compares in all cache partitions to detect and resolve coherency issues. Our scheme also has the ability to specify multiple non-contiguous partitions (through the bit-vectors) with possibly a global replacement (among the different partitions). In addition, in our scheme we need to restrict the partitioning to only select load/store instructions.

A partitioned instruction cache architecture is proposed in [27]. A partition descriptor table is used to record the start address and the size of the partition. A new instruction is used to set a special purpose partition identifier register which is used to specify the current partition identifier. Similar to TLB based address translation, all load/store instructions goes through an additional address translation process using the partition descriptor table entry specified by the partition identifier register.

The PC of the load/store instruction in a loop is used to index into a Partition Mapping Identification Table (PMIT) in the scheme proposed in [28]. The PMIT points to the Partition Identification Table (PIT). The PIT contains the partition size and offset of the partition in the original cache - this information is used later to index into the correct set of the partition. The load/store instructions that are not partitioned are mapped to a dedicated partition. This scheme does not handle coherency issues as they limit their analysis to loops with affine accesses. The table lookups could be in the critical path and thus could delay the hit. Unlike ours, they do not handle multiple partitions for a single load/store.

6 Conclusion

We presented two software/hardware co-managed cache optimization schemes to address the limitations of traditional hardware based cache management schemes. We have shown that software specified priority based replacement scheme can reduce the cache misses than LRU based schemes. We have shown that it is feasible to achieve better block placement via virtual partitioning of cache arrays instead of making placement decisions based on memory address and replacement heuristics. We described algorithms and heuristics for selectively assigning priorities and virtual partitions to the data referenced by load/store instructions. Our preliminary experimental results show that the proposed schemes can be effectively used for tailoring the caching strategy for each individual application.

References

[1] K. So and R. Rechtshaffen, "Cache operations by MRU change," *IEEE Transactions on Computers*, vol. 37, pp. 685–690, June 1998.

[2] J. Robinson and M. Deverakonda, "Data cache management using frequency-based replacement," in *Proc. of the 1990 ACM Sigmetrics Conf. on Measurement and Modelling of Computer Systems*, 1990.

[3] E. Hallnor and S. Reinhardt, "A Fully Associative Software-Managed Cache Design," in *Proc. of the 27th Annual Int'l Symposium on Computer Architecture*, June 2000.

[4] M. Kandemir *et al.*, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Transactions on Computer-Aided Design*, vol. 23, pp. 243–260, Feb. 2004.

[5] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems," in *Proc. of Intl. Conf. on Compilers, Architecters and Synthesis of Embedded Systems*, Sept. 2003.

[6] M. Verma *et al.*, "Dynamic Overlay of Scratchpad Memory for Energy Minimization," in *Proc. of the Intl. Symposium on System Synthesis*, Sept. 2004.

[7] M. Quershi, D. Thompson, and Y. Patt, "The V-Way Cache: Demand Based Associativity via Global Replacement," in *Proc. of the Intl. Symposium on Computer Architecture*, June 2005.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of the 4th IEEE Workshop on Workload Characterization*, Dec. 2001.

[9] L. Henning, "SPEC CPU2000: Measure CPU Performance in the new New Millennium," *IEEE Computer*, vol. 33, pp. 28–35, July 2000.

[10] M. Moudgill, J.-D. Wellman, and J. H. Moreno, "Environment for PowerPC microarchitecture exploration," *IEEE-MICRO*, vol. 19, pp. 15–25, May/June 1999.

[11] J. Rivers and E. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," in *Proc. of the Int'l Conf. on Parallel Processing*, Aug. 1996.

[12] A. Gonzalez, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proc. of the Int'l Conf. on Supercomputing*, July 1995.

[13] M. Prvulovic *et al.*, "The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation," in *IEEE TCCA Newsletter*, July 1999.

[14] E. S. Tam, "Improving Cache Performance via Active Management," Tech. Rep. Doctoral dissertation, Univ. of Michigan, June 1999.

[15] V. Milutinovic *et al.*, "The Split Temporal/Spatial Cache: Initial Performance Analysis," in *Proc. of the SCIzzL-5*, Mar. 1996.

[16] G. Kurpanek *et al.*, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," in *COMPCON Digest of PApers*, Feb. 1994.

[17] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. of the 30th Annual Intl. Symposium on Microarchitecture*, Dec. 1997.

[18] J. Sanchez and A. Gonzalez, "A Locality Sensitive Multi-Module Cache with Explicit Management," in *Proc. of the Intl. Conf. on Supercomputing*, June 1999.

[19] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," in *Proc. of the 28th Annual Intl. Symposium on Microarchitecture*, Dec. 1995.

[20] IBM Corporation, *PPC440x5 CPU Core User's Manual*, July 2003. (SA14-2613-03).

[21] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application Specific Memory Management in Embedded Systems Using Software-Controlled Caches," in *Proc. of the 37th Design Automation Conf.*, June 2000.

[22] P. Jain and S. Devdas, "Software-assisted Cache Replacement Mechanisms for Embedded Systems," in *Proc. of the Int'l Conf. on Computer-Aided Design*, Nov. 2001.

[23] P. Jain, S. Devadas, and L. Rudolph, "Controlling Cache Pollution in Prefetching With Software-assisted Cache Replacement," Tech. Rep. CSG Memo 462, MIT, July 2001.

[24] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems, "Using the Compiler to Improve Cache Replacement Decisions," in *Proc. of the 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[25] J. Sartor, S. Venkiteswaran, K. McKinley, and Z. Wang, "Cooperative Caching with Keep-Me and Evict-Me," in *Proc. of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9)*, Feb. 2005.

[26] T. Juan, D. Royo, and J. J. Navarro, "Dynamic Cache Spliting," in *Proc. of the XV Int'l Conf. of Chilean Computational Society*, Nov. 1995.

[27] J. Irwin, M. May, H. Muller, and D.Page, "Predictable Instruction Caching for Media Processors," in *Proc. of the 13th Intl. Conf. on Application-specific Systems, Architectures and Processors*, July 2002.

[28] P. Petrov and A. Orailoglu, "Performance and Power Effectiveness in Embedded Processors: Customizable Partitioned Caches," *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 1309–1318, Nov. 2001.