Global Management of Cache Hierarchies

Mohamed Zahran
City University of New York
New York, NY
USA
mzahran@acm.org

Sally A. McKee
Chalmers University of Technology
Gothenburg
Sweden
mckee@chalmers.se

ABSTRACT

Cache memories currently treat all blocks as if they were equally important, but this assumption of equally importance is not always valid. For instance, not all blocks deserve to be in L1 cache. We therefore propose globalized *block placement, and we present a global placement algorithm for managing blocks in a cache hierarchy by deciding where in the hierarchy an incoming block should be placed. Our technique makes decisions by adapting to the access patterns of different blocks.*

The contributions of this paper are fourfold. First, we motivate our solution by demonstrating the importance of a globalized placement scheme. Second, we present a method to categorize cache block behavior into one of four categories. Third, we present one potential design exploiting this categorization. Finally, we demonstrate the performance of the proposed design. For the SPEC CPU benchmark suite, the scheme enhances overall system performance (IPC) by an average of 12% over a traditional LRU scheme, reducing traffic between the L1 and L2 caches by an average of 20% while using a table as small as 3KB.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: Processor Architectures — *General*

General Terms

Design, Performance

Keywords

cache memory, memory hierarchy

1. INTRODUCTION

As the gap between processor and memory speeds increases, effective and efficient cache hierarchies become more and more crucial. The currently common method for addressing this memory wall problem exploits several levels of cache (and usually some form of hardware prefetching). Unfortunately, designing an efficient cache hierarchy is anything but trivial, and requires choosing among myriad parameters at each level. One pivotal design decision controls block placement — where to put an incoming block. Placement policies affect overall cache performance, not only in terms of hits and misses, but also in terms of bandwidth utilization and response times. A poor policy can increase the number of misses, trigger higher traffic lower levels of the hierarchy, and increase miss penalties. Given these problems, much research and development effort has been devoted to finding effective cache placement and replacement policies. Almost all designs resulting from these studies deal with the policies *within a single cache*. Although such local policies can be efficient within a cache, they cannot take into account interactions among several caches in the (ever deeper) hierarchy. Given this, we advocate a *holistic* view of the cache hierarchy.

Cache policies usually assume that all blocks are of the same importance, deserving a place in all caches, since *inclusive* policies are usually enforced. In our observation, this is not true. A block that is referenced only once does not need to be in cache, and the same holds for a block referenced very few times over a long period (especially for L1 cache). Overall performance depends not only on how much data the hierarchy holds, but also on *which* data it retains. The working sets of modern applications are much larger than all caches in most hierarchies (exceptions being very large, off-chip L3 caches, for instance), which makes deciding which blocks to keep where in the hierarchy of crucial importance. We address precisely this problem.

In this paper we segregate block behaviors into four categories. We show how each category must be treated in terms of cache hierarchy placement. Finally, we propose an architecture implementation that dynamically categorizes blocks and inserts them in the hierarchy based on their categories.

2. BACKGROUND AND RELATED WORK

Inclusion in the cache hierarchy has attracted attention from researchers since caches were introduced. Cache hierarchies have largely been inclusive for almost two decades; that is, L1 is a subset of L2, which is subset of L3, and so on. This organization worked well before the sub-micron era, especially when single-core chips were the primary design choice. Uniprocessor cycle times were often large enough to hide cache access latencies.

With the advent of multiple cores on a chip [1, 2, 3], on-chip caches are increasing in number, size, and design so-

CF'10, May 17–19, 2010, Bertinoro, Italy.

phistication, and private caches are decreasing in size. For instance, the IBM POWER4 architecture [4] has a 1.5MB L2 cache organized as three slices shared among its two processor cores; the IBM POWER5 has a 1.875MB L2 cache with a 36MB off-chip L3 [5]; the Intel Itanium [6] has a three-level, on-chip cache with combined capacity of 3MB; and the Intel Core i7 (Nehalem) has a shared L3 inclusive cache of 8MB [7]. As the complexity of on-chip caches increases, the need to reduce miss rates grows in importance, as does access time (even for L1 caches, single-cycle access times are no longer possible).

Designers have traditionally maintained inclusion in the memory hierarchy for several reasons: for instance, in multiprocessor systems, inclusion simplifies memory controller and processor design by limiting effects of cache coherence messages to higher levels in the memory hierarchy. Unfortunately, cache designs that enforce inclusion are inherently wasteful with respect to both space and bandwidth: every line in a lower level is duplicated in higher levels, and updates in lower levels trigger many more updates in other levels, wasting bandwidth. As the relative bandwidth onto a multiple-core chip decreases with the number of on-chip CPUs and relatively smaller cache real estate per CPU, this problem has sparked a wave of proposals for non-inclusive cache hierarchies.

We can violate inclusion two ways. The first is to have a non-inclusive cache, and the second is to have a mutually exclusive cache. For the former, we simply do not enforce inclusion. Most of the proposals in this category apply a replacement algorithm that is local to individual caches. For instance, when a block is evicted from L2, its corresponding block is *not* evicted from L1. However, the motivation for such schemes is to develop innovative *local* replacement policies. Qureshi et al. [8] propose a replacement algorithm in which an incoming block is inserted in the LRU instead of MRU position without enforcing inclusion, since blocks brought into cache have been observed to move from MRU to LRU without being referenced again. The authors decouple block placement in the LRU stack and victim selection. Xie and Loh propose to also decouple block promotion on a reference [9]. All these techniques improve efficiency, but only at the levels of individual caches: each cache acts individually, with no *global* view of the hierarchy. We instead propose schemes that are complementary to and can be combined with such local schemes, but our approach has a globalized view of the whole hierarchy.

The second method for violating inclusion exploits mutually exclusive caches [10]. In a two-level hierarchy, the caches can hold the number of unique blocks that fit into both L1 and L2. This approach obviously makes the best use of on-chip cache real estate. In an exclusive hierarchy, the L2 acts as a victim cache [11] for the L1. When both miss, the new block comes into L1, and when evicted, it moves to L2. A block is promoted to L1 when an access hits in L2.

Our proposed scheme works as a "middle way" between inclusive cache hierarchies and mutually exclusive hierarchies. It can be viewed as non-inclusive, but the scheme enforces a *global placement* policy. It manages the entire cache hierarchy, instead of individually managing each separate cache. Most related work concentrates on managing individual caches.

Some proposed policies adapt to application behavior, but usually within a single cache. For instance, Qureshi et al. [8] propose retaining some fraction of the working set in cache so that fraction can contribute to cache hits. Subramanian et al [12] present another adaptive replacement policy: the cache switches between different replacement policies based on access behavior. Wong and Baer [13] propose techniques to close the gap between LRU and OPT replacement.

All cache misses are not of equal importance (e.g., some data are required more quickly by the instructions that consume them, whereas others are required by instructions that are more latency tolerant). The amount of exploitable memory level parallelism (MLP) [14, 15, 16] also affects application performance, and thus Qureshi et al. [17] propose an MLP-aware cache replacement policy. In this paper we propose a scheme that is *globalized*, *adaptive*, and *complementary* to most of the aforementioned techniques.

3. BLOCK BEHAVIOR: A CASE STUDY

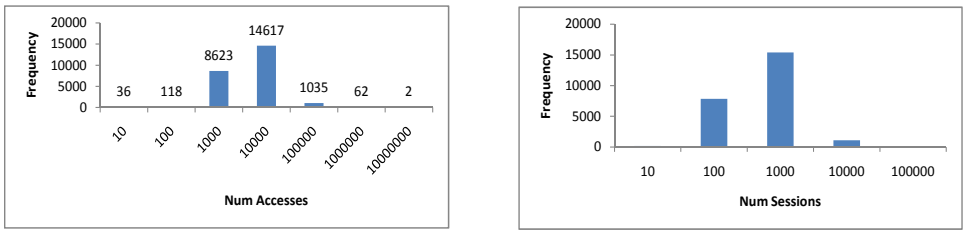
The performance of a cache hierarchy and its effects on overall system performance inherently depend on cache block behavior. For example, a block rarely accessed may evict a block very heavily accessed, causing in higher miss rates. Sometimes, if the evicted block is dirty, higher bandwidth requirements result.

The behavior of a cache block can be summarized by two main characteristics: the number of times it is accessed, and the number of times it has been evicted and re-fetched. The first is an indication of the importance of the block, and the second shows how block accesses are distributed in time. As an example, Figure 1 shows two benchmarks from SPEC2000: `twolf` from SPECINT and `art` from SPECFP [18]. These two benchmarks are known to be memory bound [19]. The figure shows four histograms. Those on the left show the distribution of total numbers of accesses to different blocks. For `twolf` the majority of blocks are accessed between 1,000 and 10,000 times, but for `art` the majority are accessed between 100 and 1,000 times. Some blocks are accessed very few times: more than 8,000 blocks are accessed fewer than 100 times. The histograms on the right show numbers of block reuses, or the number of times a block is evicted and reloaded. Over 15,000 unique blocks in `twolf` and over 25,000 in `art` are loaded more than 1000 times.

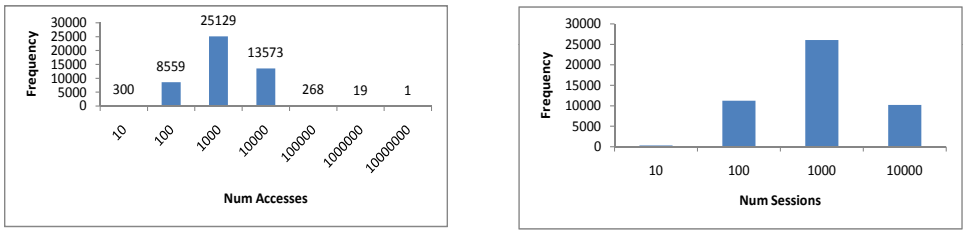
Based on these observations, a block may be loaded very few times, and may be accessed very lightly in each epoch (time between evictions). Other blocks can be loaded many times and accessed very heavily in epoch. Many fall between these extremes. Success of any cache placement policy depends on its ability to categorize block access behavior to determine correct block placement based on this behavior. This placement policy must be global: it must be able to place a block at any hierarchy level based on the block's behavior. In this paper we assume a two-level cache hierarchy for demonstration purposes.

4. ADAPTIVE BLOCK PLACEMENT (ABP)

The success of *Adaptive Block Placement* (ABP) depends on the ability to capture the behavior of cache blocks. With current state-of-the-art processors [4, 5, 20, 21, 22], block size is fixed across the cache hierarchy. Unfortunately, this often means that misses in a higher level will also miss in a lower level cache. Observing block requests from the processor to the cache hierarchy, as in Section 3, allows us to classify each block into one of four categories:



(a) twolf



(b) art

Figure 1: `twolf` and `art` Block Behavior at the L1 Data Cache

- blocks that are accessed frequently, and for which time between consecutive accesses is small (high temporal locality);
- blocks that are accessed frequently for a short period, are not accessed again soon, but then are accessed again frequently within short periods (repetitive, bursty behavior);
- blocks that are accessed in a consistent manner, but for which the time between consecutive accesses is larger than for the first category; and
- blocks that are rarely accessed.

Figure 2 shows the four access types. The best hierarchy should behave differently for each category. Blocks from the first category should be placed in both L1 and L2, since these are accessed frequently. Placing them in L1 allows them to be delivered to the processor as fast as possible. Evicting such blocks from L1 due to interference should allow them to reside in L2, since they are still needed. Blocks from the second category do not benefit from the L2. These blocks will be heavily referenced for a while, so they should be kept in L1, but they will not be needed again soon once evicted. Blocks in the third category should be placed in L2 but not in L1. L1 can thus be reserved for blocks that are heavily referenced, and for blocks not heavily referenced but that do not severely affect performance. Finally, blocks from the last category will bypass the cache hierarchy [23], and will be stored in no cache. Note that if consecutive accesses to blocks in the third category are very far apart, these blocks can be downgraded to the fourth category.

The advantages of adaptive block placement schemes can be illustrated by the following example. Assume the memory is accessed as shown in Figure 3(a). These instructions access four different cache blocks, `X, Y, Z, and W`. For the sake of the example, assume L1 is direct-mapped and L2 is two-way set associative, and that the four blocks map to the same set in both caches. Figure 3(b) shows the timeline for accesses to each block. Every tick represents an access. If we map our four categories to these blocks, then block `X` is placed in L1, because it has bursty access patterns, then periods of no accesses before it is touched again. Blocks `Y` and `Z` are placed in L2, because they are accessed consistently, but time between successive accesses is long. Finally, block `W` is not cached, because it is rarely accessed. Figure 3(c) shows hits and misses for both L1 and L2 for a traditional LRU scheme. We do not enforce inclusion: that is, a block victimized at L2 does not victimize any corresponding block at L1. A quick look at hits and misses from a traditional LRU policy reveals two things. First, the hit rate at L1 is 1/11 and at L2 is 2/10 (since a hit at L1 needs no L2 access). Second, L2 has been accessed ten times out of eleven references. Figure 3(d) illustrates ABP, which yields better performance. Both caches start empty. When block `X` is loaded after a compulsory miss, it is put into L1 but not L2. Blocks `Y` and `Z` are loaded into L2 but not L1. Block `W` is not loaded into any cache. The hit rate at L1 becomes 4/11, and at L2, 3/7, both of which are higher than for the traditional LRU hierarchy. Moreover, L2 is accessed only seven times, which reduces energy consumption.

Our goal is to design a system that captures the behavior of each cache block and categorizes it into one of the four behaviors we identify. The ultimate goal, of course, is to satisfy most of the references from L1. Keeping the L1 size fixed and assuming it is not fully associative (although there exist some examples of fully associative L1s in real machines [24]), we try to decrease conflict misses by reducing contention among L1 sets. We do so by keeping some blocks out of the L1.

Figure 4 shows one possible ABP implementation. The main component is the *Behavior Catcher* (BC). The BC keeps track of all address requests the processor generates. Thus, the BC snoops the address bus from the processor to the cache hierarchy. After an L1 miss, the BC is triggered to make a decision about the incoming block's placement. While L2 is accessed, the BC chooses the category of the incoming block, if possible. If the access misses in the L2 and

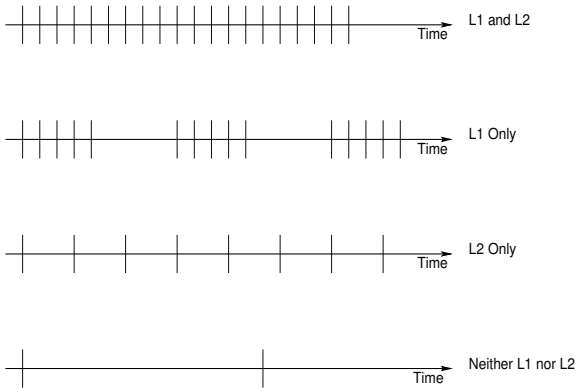


Figure 2: Access Patterns for Different Categories (vertical lines represent block accesses)

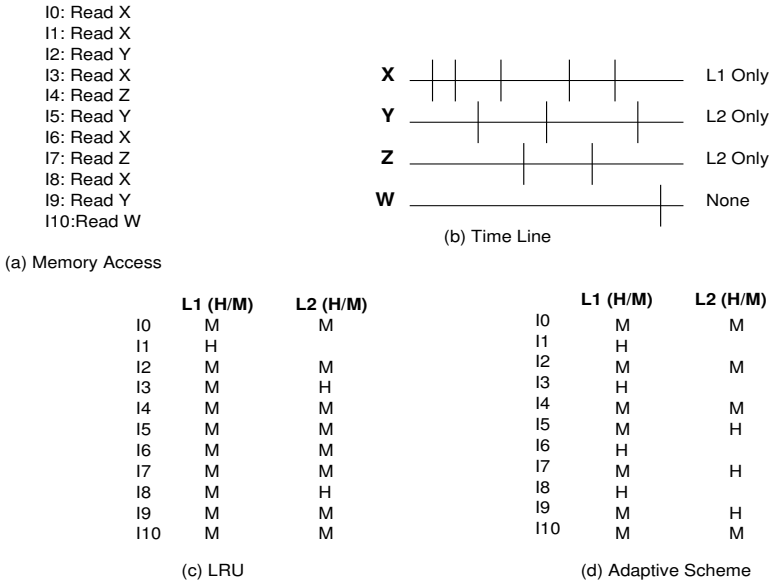


Figure 3: Example of LRU vs an Adaptive Scheme

memory is accessed, the incoming block is placed according to the BC's decision. That is, it will go to L1, L2, both, or neither, depending on categorization. If no decision can be made, the hierarchy follows the behavior of a traditional hierarchy by bringing the block into both L1 and L2. If the access hits in L2 and the BC's decision is "L2 only" or "neither L1 nor L2", then the block does not go to L1, and the required word goes directly to the processor. On the other hand, if, after an L2 miss, the BC's decision is "L1 only", or "L2 and L1", the block goes to L1. The BC is updated each time the processor generates a memory request. This operation is not on the critical path, and does not affect access latency. ABP hierarchies do not differentiate between loads and stores: all are treated simply as memory accesses. Because the BC is triggered only after L1 misses, a block that was previously designated as "L2 only" now becomes "L1 and L2" if the BC decision changed based on the access pattern.

4.1 ABP Design

Figure 4 shows one potential BC design, a small direct-mapped cache, the address decoder for which consists only of tag and set: no offset is required. Each data array entry consists of a *pattern*, or string of bits representing the history of the corresponding cache block. A 1 indicates that the block is accessed, and a 0 means the block is not accessed. When an address goes to the BC as part of an update operation, if the block resides in the table, a 1 is entered to the right of the pattern entry, and the pattern is shifted left one bit. To approximate temporal reference activity, we insert a 0 to the right of each pattern at specific events, and the entry is shifted one bit left, as per usual updates. Each address thus has a pattern representing its *access history*. This history is an approximation of the patterns from Figure 2, where each vertical line is represented with a 1. A major design parameter here is when to insert the 0s. We can insert them every X cycles. However, this does not capture the behavior of the program at hand. For instance, if a program is processor-bound and the memory system is rarely accessed, many 0s will be inserted, resulting in most blocks being kept out the cache system, which can negatively affect performance. Inserting 0s at events such as L1 misses or L2 misses is another option. Preliminary experiment results were not very good for these policies, since a long sequence

of misses due to working set change, for example, can disturb the prediction scheme. We need a scheme that can represent a block's behavior as faithfully as possible. For a program that accesses the memory M times, spans B distinct cache blocks, and takes a total of C cycles, M is not equally distributed among B and C. In order to capture a block's behavior, we need somehow to *re-map* the time as if there were a memory access at every cycle. So after the table is updated with U 1s, where U in the number of entries in the table, we insert 0s. The main idea is that we are giving each entry of the table a chance to be updated. Some entries will be updated more than others, though.

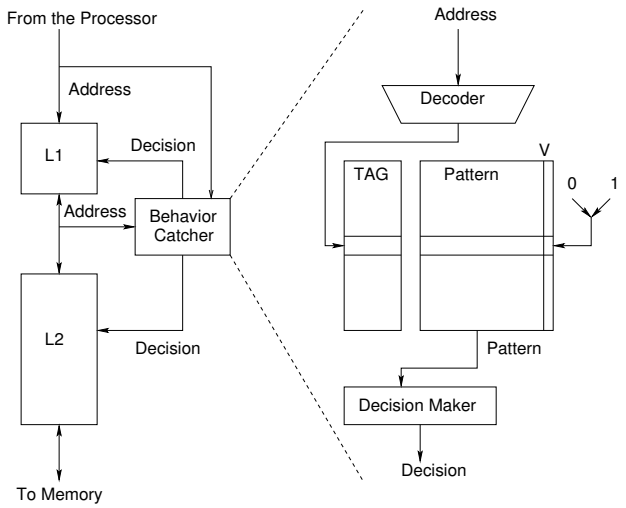


Figure 4: One Possible Implementation of ABP

When the BC is given an address about which to make a decision, the address decoder splits that address into TAG and SET values. If the stored tag of the entry specified by the SET value matches the TAG value, the BC copies the corresponding pattern to the decision maker. Figure 5 presents a pictorial view of the decision-making process. The pattern is split into left and right halves. The right part indicates the most recent access pattern, due to the shift operation, and the left represents older behavior. Each half is checked to see whether it has a majority of 1s or 0s, or an equal number of each. A majority of 1s in a half means that the block is heavily accessed. A majority of 0s means it is rarely accessed. A tie means the block is frequently accessed, but not as heavily as if it had a majority of 1s. Figure 5 indicates the decision based on the majority rule of each half.

4.2 Hardware Complexity

The proposed design requires very little hardware. We use CACTI 4.2 [25] to calculate the cost of our ABP hardware. We find that for a table of 1024 entries with 8-bit patterns per entry and 8-bit tags, implementing ABP takes around 1% of a unified L2's area and consumes less than 4% of the power per access. Our experiments show that ABP greatly reduces the number of writebacks to multiple levels of the hierarchy. It also reduces the number of blocks residing in cache for a long time without being accessed (which reduces leakage). ABP can thus save power, and these savings more than offset the power consumed by the ABP hardware. Finally, each cache slot must be augmented by two bits to indicate the state its last BC decision. A no-decision state is considered "L1 and L2". Like the BC, the ABP is not on the critical path, and does not affect the access times of the cache hierarchy.

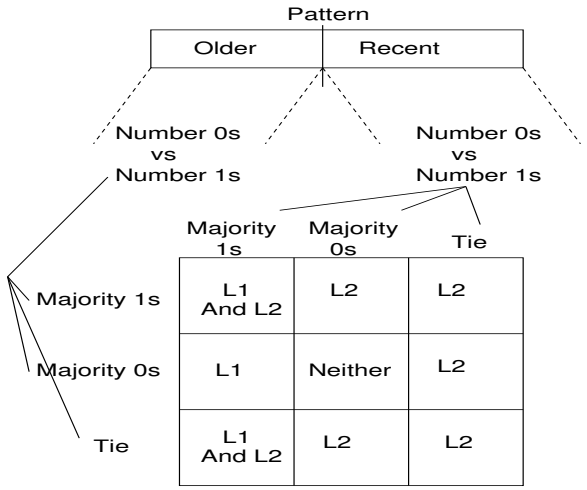


Figure 5: The Decision Process for ABP

The design we present is but one way of implementing ABP. More efficient designs are certainly possible, but here we focus on demonstrating the potential effectiveness of adaptive global placement policies. Refining our design is part of ongoing work.

5. EXPERIMENTAL SETUP

For our experiments, we use a heavily modified version of simplescalar [26] with the Alpha instruction set, modifying it further to generate the statistics we need and to implement our proposed cache management modifications. We choose a fixed block size for all caches in the hierarchy, similar to IBM POWER series [5] and many other state-of-the-art processors. Table 1 shows fixed simulator parameters. Parameters not in the table use default simulator values that are typical of current high-performance cores.

Our table for the ABP has 1024 entries, keeps a pattern of 16 bits per entry, and uses 1.5 bytes of the block address as the tag (from the least significant bits). This ABP requires only 3KB of storage. We choose these parameters empirically from running many design-space exploration experiments to find the best price/performance. As always, the best parameter values are application dependent. We are enhancing the system to make it more application-aware in its adjustments for the application at hand. We tried several schemes with respect to when entries are shifted (at L1 miss, at L2 miss, after X cycles). The best strategy is also application specific. We choose shifting entries after 10000 cycles, which has adequate performance for most applications. We compare ABP with a conventional inclusive hierarchy with LRU replacement, an exclusive hierarchy[10], and a hierarchy with cache bypassing[23]. Cache sizes capacity is the same for all schemes. Cache bypassing uses two tables, one for L1 data cache and the other for the shared L2. Each table consists of 128 entries, with five-bit saturating counters per entry, and macroblocks of 1KB. We choose these sizes to be as close as possible to our ABP scheme, although

bypassing uses a larger table due to the need for saturating counters and the fact that two tables require at least two ports.

Parameter	Setting
Instruction Fetch Queue *Decode Width*	32 8
Issue width	8
Instruction Window	128
Load/Store Queue	64
Branch Predictor	combination of bimodal, 2048 table size and 2-level predictor
L1 Icache/Dcache	32KB, 4-way associative, LRU 64B line size, 2-cycle latency
L2 Unified	1MB, 8-way associative, LRU 64B line size, 10-cycle latency
Memory Latency	300 cycles for the first chunk
Execution Units	2 int and 1 fp ALUs 2 int mult/div and 1 fp mult/div

Table 1: Simulator Parameters

For the benchmarks, we use 23 of the SPEC2000 CPU suite, both integer and floating point. We use SimPoint [27] to skip the startup portion of each application, and we simulate a representative 250M instructions on reference inputs. Table 2 shows total numbers of committed references (loads and stores).

6. RESULTS AND DISCUSSION

These results present a proof-of-concept for global block placement based on the four categories discussed in Section 4, but they are heavily implementation-dependent. Obviously, different implementations are likely to deliver different results.

6.1 ABP Behavior

The best way to assess the performance of ABP is to first observe the decisions it makes, and then see how these decisions affect performance. Figure 6 shows the decisions made by ABP for each application in our benchmark suite. For several benchmarks, ABP decides to bypass *both* caches about 50% of the time. This happens for one of two reasons, depending on the benchmark. The first is that the blocks are accessed very few times and not referenced again. The second is that the blocks are accessed frequently, but the time between successive accesses is large enough not to be captured by the behavior patterns in ABP (as with `art`, discussed in Section 3). Note that, on average, ABP provides a prediction 98% of the time.

Benchmark	Total References	Benchmark	Total References
applu	94621773	lucas	54192511
apsi	93965724	mcf	83005160
art	87593640	mesa	94429723
bzip2	92398416	mgrid	91704475
crafty	94450697	parser	94957683
eon	117813076	perlbmk	100245381
equake	112536995	sixtrack	61725677
facerec	82164501	swim	82790690
fma3d	2951034	twolf	83256750
gcc	121146712	vortex	110628341
gcc	121146712	vpr	110195396
gzip	79803491	wupwise	79449025

Table 2: Simulated Applications

For blocks accessed frequently across a large time-frame, overall performance is practically unaffected. The cache real estate is reserved for more urgently needed blocks. Moreover, memory level parallelism (MLP) [17] in current state-of-the-art memories helps mask the penalty of bypassing the cache for these blocks, although we do not explore ways to better exploit this MLP here. Furthermore, from a power-consumption point of view, blocks that are accessed across a long time frame will stay in cache once loaded, especially for caches with high associativity. This is true even if they are not accessed for an extended period. Unused cache lines not only may degrade performance, but they also cause leakage power dissipation, which has become a major factor in the current sub-micron era [28]. The replacement policy may be another factor that supports making such blocks bypass the cache hierarchy. If a cache set is full, a replacement policy must choose a block to evict when another is fetched. This increases power consumption as well as hit latency. If the set has one or more empty slots, no replacement is required, or at least the LRU stack (when a Least Recently Used policy is employed) will not be full. By decreasing the number of blocks coming into the cache, fewer sets will be full, and hence more new blocks can be fetched into a set without replacement (and possible write-backs of dirty blocks to memory, which consume power and bandwidth).

In order to show the effects of ABP on performance, Figure 7 shows the instructions per cycle for the different techniques. ABP shows higher performance than the other techniques for most benchmarks. ABP is better than the traditional inclusive hierarchy for most benchmarks by an average of about 12%. This performance is due to several factors that differ in importance based on the application. The first is the decrease in cache pollution. The second is the decrease in the amount of bandwidth required, especially the between L1 and L2 caches, as will be shown in the next section. The third is the saving of replacement overhead. With fewer blocks coming into the cache, fewer sets will be full, and hence fewer replacements will be needed. Advantages should be even larger if MLP techniques are used, or in SMT scenarios, since the effective cache size increases.

Another factor that affects performance is the off-chip bandwidth requirement, which is mainly due to the L2 miss rate. Figure 8 shows the miss rates for the different schemes. ABP has lower miss rates than the other schemes. This means less off-chip bandwidth is required, and hence less pressure is placed on the socket pins (which are not very scalable), memory bus, memory channels, and off-chip memory ports.

6.2 Traffic Reduction

Another important factor in the efficiency of a cache hierarchy is the on-chip and off-chip traffic. In the previous section we showed that ABP has lower L2 miss rates, and this translates to lower off-chip bandwidth requirements. In this section we look at the traffic between L1 and L2 caches, which is expected to skyrocket with the increase in the number of on-chip cores. This traffic has a pivotal effect on the scalability of the system. The traditional multicore design consists of several cores with private L1 caches and a shared L2 cache. If the traffic between each core and the shared L2 is high, this puts pressure on the on-chip interconnection network and on the number of ports required for the L2 cache to deliver adequate response times. Reducing the

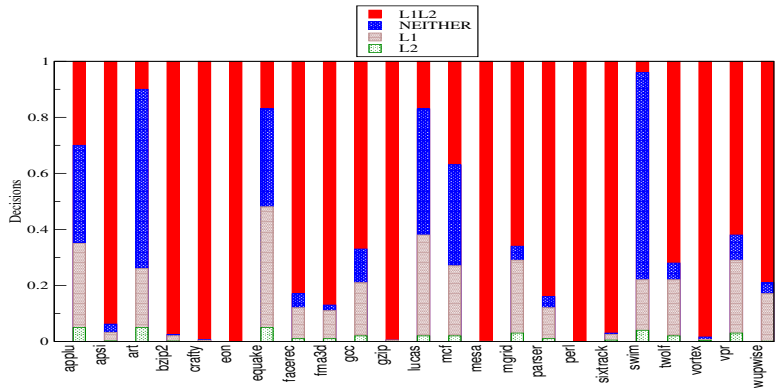


Figure 6: Decision Statistics of ABP

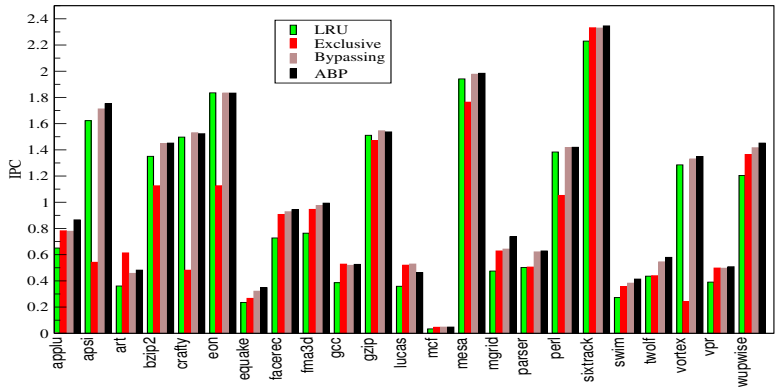


Figure 7: Instructions per Cycle

L1-L2 bandwidth requirement is thus of crucial importance for future systems.

Figure 9 shows the total traffic between the L1 and L2 caches. We omit the exclusive hierarchy because L2 acts as a large victim cache, and hence traffic is expected to be an order of magnitude higher than in traditional schemes. ABP has much lower traffic than bypassing and lower traffic than LRU. This is due to two main factors. The first is the reduction in L1 pollution and hence lower miss rates, so Figure 9 by itself is an indication of lower L1 miss rates. The second is the decrease in the number of full sets in L1, which reduces traffic due to writebacks.

These data show that some blocks cannot be easily categorized. They exhibit seemingly random behaviors. For these blocks, ABP responds in one of two ways, depending on the program. Either ABP makes no predictions, and the system behaves as a traditional inclusive hierarchy (with no affect on performance), or ABP makes predictions that may be wrong, usually due to aliasing. Fortunately, incorrect predictions minimally degrade performance.

7. ABP IN DIFFERENT ENVIRONMENTS

We have shown how ABP works in a two-level cache hierarchy. In this section we will give a quick glimpse on the usage of ABP in different organizations.

ABP in multi-level caches:.

Changing the decision matrix (Figure 5) allows ABP to scale to any number of cache levels. The more important a block (according to its access pattern), the better that it be placed in more cache levels.

ABP in multicore processors:.

Using ABP is like using a non-inclusive hierarchy when it comes to coherence. L1 tags must be duplicated in the L2, assuming that the shared cache is L3. One ABP table keeps track of all block accesses.

ABP with an inclusive L2:.

To impose inclusiveness on some blocks (i.e., some blocks MUST be both in L1 and L2), these blocks can just be marked as inclusive (like blocks marked as cachable or non-cachable), and then ABP will not predict placement for them.

8. CONCLUSIONS AND FUTURE WORK

In this paper we present a method for placing the access pattern of each block into one of four categories, and we present a possible design for doing so. We find that with a table as small as 3 KB, we enhance the overall performance of 23 SPEC2000 applications by an average of 12% while reducing traffic from the L1 cache to the L2 cache. This

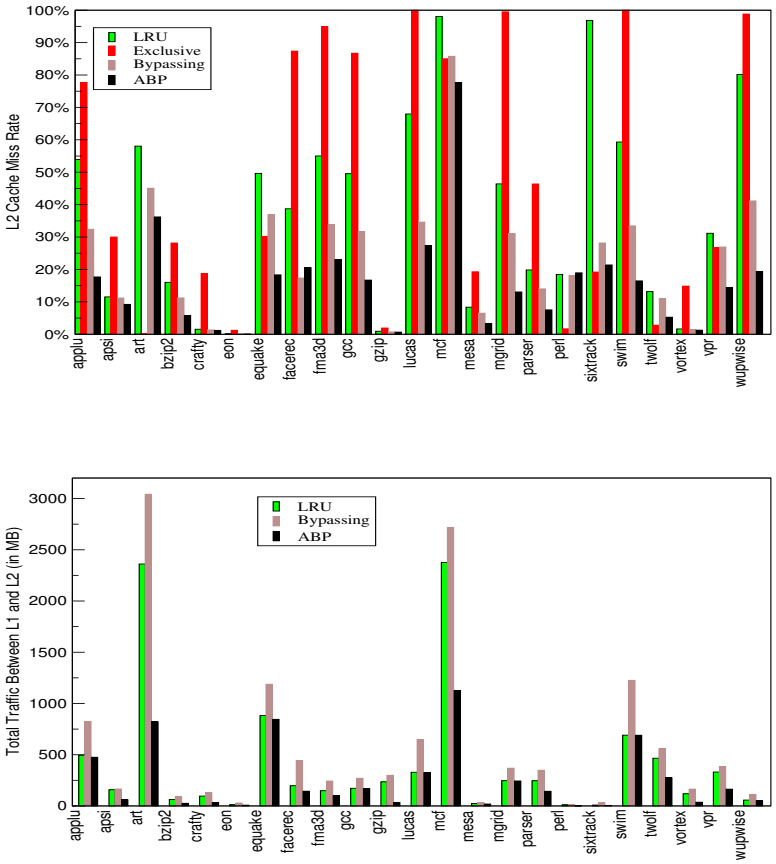


Figure 9: Total Traffic between L1 and L2 Caches (MB)

approach should become increasingly useful as the number of cores per chip scales. ABP makes its decisions based on address streams, and thus it is independent of the number of cores. We are currently enhancing our scheme via several paths, including conducting sensitivity studies to fine-tune ABP, making ABP self-tuned by adding another level of *learning*, and extending our concepts to both SMT processors and to CMP designs.

9. REFERENCES

[1] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, "CMP design space exploration subject to physical constraints," in *Proc. 12th IEEE Symposium on High Performance Computer Architecture*, pp. 15–26, Feb. 2006.

[2] J. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP throughput with mediocre cores," in *Proc.IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 51–62, Oct. 2005.

[3] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *IEEE Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[4] I. Corporation, "POWER4 system microarchitecture," , vol. 46, no. 1, 2002.

[5] B. Sinharoy, R. N. Kalla, J. M. T. andR. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM Journal or Research and Development*, vol. 49, no. 4/5, 2005.

[6] D. Weiss, J. Wuu, and V. Chin, "The On-Chip 3-MB Subarray-Based Third-Level Cache on an Itanium Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, 2002.

[7] http://www.intel.com/technology/architecture-silicon/next-gen/.

[8] M. Qureshi, A. Jaleel, Y. Patt, S. S. Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th International Symposium on Computer Architecture (ISCA)*, pp. 381–391, Jun. 2007.

[9] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. 36th International Symposium on Computer Architecture (ISCA)*, Jun. 2009.

[10] Y. Zheng, B. T. Davis, and M. Jordan, "Performance evaluation of exclusive cache hierarchies," in *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 89–96, Mar. 2004.

[11] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffer," in *Proc. 17th International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[12] R. Subramanian, Y. Smaragdakis, and G. Loh,

"Adaptive caches: Effective shaping of cache behavior to workloads," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pp. 385–396, Dec. 2006.

[13] W. Wong and J.-L. Baer, "Modified lru policies for improving second level cache behavior," in *Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pp. 49–60, Jan. 2000.

[14] T. Puzak, A. Hartstein, P. Emma, and V. Srinivasan, "Measuring the cost of a cache miss," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, Jun. 2006.

[15] S. McKee, W. Wulf, J. Aylor, R. Klenke, M. Salinas, S. Hong, and D. Weikle, "Dynamic access ordering for streamed computations," *IEEE Transactions on Computers*, vol. 49, pp. 1255–1271, Nov. 2000.

[16] A. Glew, "MLP yes! ILP no!." , Oct. 1998. ASPLOS VIII Wild and Crazy Ideas Session.

[17] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt, "A case for mlp-aware cache replacement," in *Proc. 33rd International Symposium on Computer Architecture (ISCA)*, Jun. 2006.

[18] Standard Performance Evaluation Corporation, "SPEC CPU benchmark suite." `http://www.specbench.org/osg/cpu2000/`, 2000.

[19] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "Dynamically controlled resource allocation in smt processors," in *37th annual IEEE/ACM international symposium on Microarchitecture*, December 2004.

[20] Broadcom Corporation, "BCM1455: Quad-core 64-bit MIPS processor." http://www.broadcom.com/collateral/pb/1455-PB04-R.pdf, 2006.

[21] Y. Choi, A. Knies, G. Vedaraman, and J. Wiliamson, "Design and experience: Using the Intel Itanium 2 processor performance monitoring unit to implement feedback optimizations," tech. rep., Itanium Architecture and Performance Team, Intel Corporation, 2004.

[22] *PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual*, 2.0 ed., July 2003.

[23] T. L. Johnson, D. A. Connors, M. C. Merten, and W.-M. Hwu, "Run-time cache bypassing," *IEEE Trans. Comput.*, vol. 48, no. 12, pp. 1338–1354, 1999.

[24] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski, "Blue Gene/L compute chip: Memory and ethernet subsystem," *IBM Journal of Research and Development*, vol. 49, no. 2-3, pp. 255–264, 2005.

[25] D. Tarjan, S.Thoziyoor, and N. Jouppi, "CACTI 4.0," Tech. Rep. HPL-2006-86, HP Western Research Laboratory, 2006.

[26] D. Burger and T. Austin, "The simplescalar toolset, version 2.0," Tech. Rep. 1342, University of Wisconsin, June 1997.

[27] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[28] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. International Symp. on Low Power Electronics and Design*, pp. 90–95, Jul. 2000.