Adaptive Block Placement Policy for Cache Hierarchies

Mohamed Zahran
Dept. of Electrical Engineering
City University of New York
mzahran@ccny.cuny.edu

Sally A. McKee
Dept. of Computer Science and Engineering
Chalmers University of Technology
sally.mckee@chalmers.se

Abstract

Cache memories currently treat all block as if they were equally important, but this is not the case. For instance, not all blocks deserve to be in L1 cache. In this paper we propose to perform globalized *block placement. We present a global replacement algorithm for managing blocks in a cache hierarchy by deciding where in the hierarchy a new incoming block should be placed. Our technique adapts to the access patterns of the different blocks, making its decisions based on these.*

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 our design. The proposed scheme enhances overall system performance (IPC) by an average of 9% while reducing traffic from the L1 cache to L2 by an average of 15%, and the traffic from L2 to main memory by an average of 40%. All of this is achieved with a table as small as 1 KBytes.

1 Introduction

As the gap between processor and memory speeds increases, the role of the cache hierarchy becomes more and more crucial. Having several levels of caches is currently the most common method for addressing the memory wall problem. Unfortunately, designing an efficient cache hierarchy is anything but trivial, and requires choosing among myriad parameters at each level. One pivotal design decision is the replacement policy. Replacement policies affect overall cache performance, not only in terms of hits and misses, but also in terms of bandwidth utilization and response time: a poor policy can increase the number of misses, can trigger much traffic out of the cache, and increase the miss penalty. Given these problems, much research and development effort has been devoted to finding effective cache replacement policies. Almost all designs resulting from these studies deal with the replacement policy *within a single cache*. Although such local replacement policies can be efficient within one cache, they do not take into account the interactions among different caches in the (ever deeper) hierarchy. Therefore, we need a *holistic view of the cache hierarchy*.

Cache hierarchy assumes that all blocks are of the same importance and hence deserves a place in L1, L2, up to the last level cache. This is not true. A block that is references only once, or very few times over a long period, does not need to be in cache, or at least not in L1 cache. Overall performance depends not only on how much data the hierarchy can retain, but also on *which* data the hierarchy retains. The working sets of current applications are much larger than all caches in most hierarchies (the exceptions being very large, off-chip L3 caches, for instance), which makes deciding which blocks to retain and where to retain them in the hierarchy of crucial importance. We address precisely this problem.

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

2 Background and Related Work

Inclusion in the cache hierarchy has attracted attention from researchers since the early days of cache memories. For almost two decades, cache hierarchies have largely been inclusive; 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 main design choice. Uniprocessor cycle times were often large enough to hide the latency of accesses within the cache hierarchy, and execution was not that aggressive.

With the advent of multiple CPU cores on a chip [1, 2, 3], on-chip caches are increasing in number, size, and design sophistication. For instance, IBM's POWER4 architecture [4] has a 1.5MByte L2 cache organized as three slices shared among its two processor cores, the IBM's POWER5 has a 1.875MByte L2 cache with a 36MByte off-chip L3 [5], and the Intel Itanium [6] has a three-level, on-chip cache with

combined capacity of 3MBytes. As the size and complexity of on-chip caches increase, the need to decrease miss rates gains additional significance, as does access time (even for L1 caches, single-cycle access times are no longer possible).

We've traditionally maintained inclusion in our hierarchies for several reasons: for instance, in multiprocessor systems, inclusion simplifies memory controller and processor design by limiting the effects of cache coherence messages to higher levels in the memory hierarchy. Unfortunately, cache designs that enforce inclusion are inherently wasteful of space and bandwidth: every cache line in a lower level is duplicated in the 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 in the cache hierarchy in two ways. The first is to have a non-inclusive cache, and the second is to have a mutually exclusive cache. For the former design, 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. In Qureshi et al. [7] the authors propose a replacement algorithm in which an incoming block is inserted in the LRU instead of MRU position without enforcing inclusion. In other words, blocks brought into cache have been observed to move from MRU to LRU without being referenced again. By bringing a new block into LRU and only making it MRU when referenced again improves efficiency. This approach improves efficiency, but only at the levels of individual caches: each cache in the hierarchy 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 latter method for violating inclusion is to have mutually exclusive caches [8]. For instance, in a two-level hierarchy, the caches can hold the number of unique blocks that can fit into both L1 and L2. This approach obviously makes the best use of the on-chip cache real-estate. In an exclusive hierarchy, the L2 acts as a victim cache [9] for L1. When both L1 and L2 miss, the new block comes into L1, and when evicted it moves to L2. A block is promoted from L2 to L1 when an access hits in L2.

Our proposed scheme works as a "middle way" between inclusive cache hierarchies and mutually exclusive cache hierarchies. It can be viewed as a non-inclusive scheme, but in spite of its being non-inclusive, our scheme is 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.

With respect to replacement policies, in general, most prior work is also local in nature. Some proposed policies adapt to application behavior, but within a single cache. For instance, Qureshi et al. [7] propose retaining some fraction of the working set in cache so that fraction can contribute to cache hits. Subramanian et al [10] present another adaptive replacement policy: the cache switches between two different replacement policies based on access behavior. Wong and Baer [11] 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) [12, 13] also affects application performance, and thus Qureshi et al. [14] propose an MLP-aware cache replacement policy.

Studying different replacement policies in depth usually requires a series of long simulation experiments, although there are some proposals for using analytic models for cache replacement [15, 16]. Analytic models are useful and efficient, but only provide information about the cache hierarchy, without providing a whole picture of the chip itself (as simulation studies can). Ultimately, the approaches are complementary.

3 Blocks Behavior: A Case Study

The performance of a cache hierarchy and its effect on the overall system performance depends on the cache blocks behavior. For example, a block that is very rarely accessed may evict a block that is heavily accessed, resulting in higher miss rate and sometimes, if the evicted block is dirty, a hirgher bandwidth requirement.

The behavior of a cache block can be summarized by two main aspects: the number of times the block is accessed and the number of times that block has been evicted and brought into the cache. The first aspect is an indication of the importance of that block, and the second aspect shows how these block accesses are distributed in time. As an example, Figure 1 shows two benchmarks from the Spec2000: `twolf`

from SpecINT and `art` from SpecFP [17]. These two benchmarks are known to be memory bound applications [18]. The figure shows four histograms. The ones on the left show the distribution of the total number of accesses to the different blocks accessed. For `twolf` the majority of the blocks are accessed between 1,000 and 10,000 times, while for `art` the majority of the blocks are accessed between 100 and 1,000 times. There are some blocks that are accessed very few times. For instance more than 8,000 blocks are accessed less than 100 times.

The histograms on the right show the number of sessions, that is, the number of times a block has been evicted and then brought again into the cache. We can see that more than 15,000 unique blocks in `twolf` and more than 25,000 unique blocks from `art` are brought into the cache more than 1000 times.

Based on the above two aspects, we can see that a block can be brought into the cache very few times and accessed very lightly in each session. Some other blocks can be brought many times and accessed very heavily in each, and there are blocks whose behavior falls in between these two extremes.

The success of any cache hierarchy placement policy depends on its success in categorizing the access behavior of each block, and the correct placement of that block in the hierarchy based on this behavior. This placement policy must be global, that is, it must be able to place a block at any level of the cache hierarchy based on the block's behavior. In this paper we assume two-level cache hierarchy.

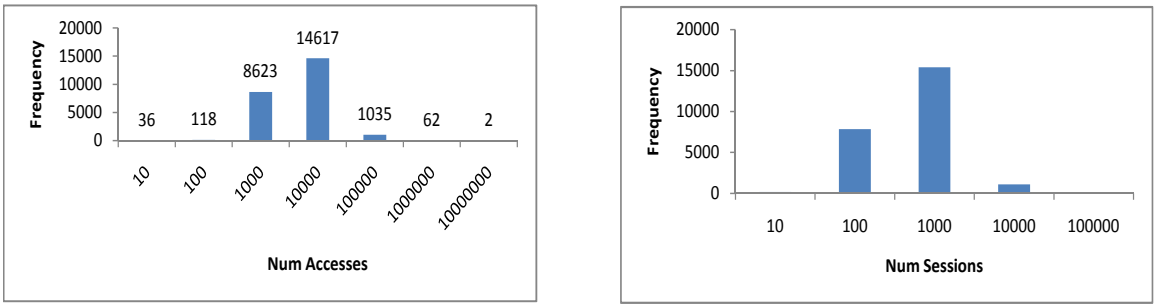
4 Adaptive Block Placement (ABP)

The success of ABP depends on capturing the behavior of cache blocks. With current state-of-the-art processors [4, 5, 19, 20, 21], block size is fixed across the cache hierarchy. Observing block requests from the processor to the cache hierarchy, as we have seen in Section 3, allows us to classify each block into one of four categories.

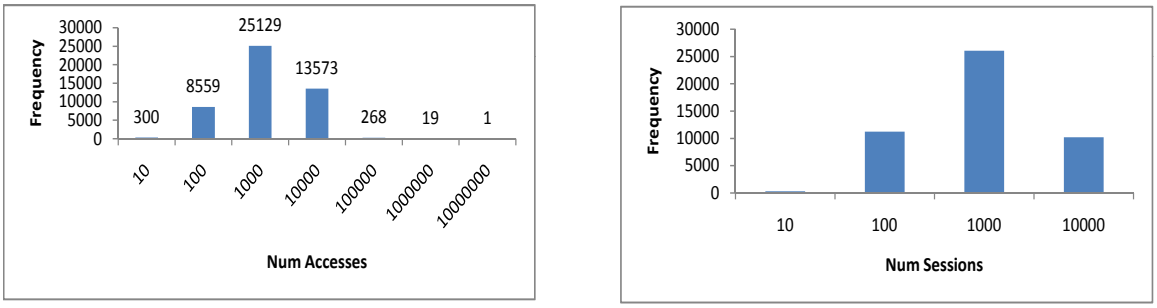
- a block is accessed frequently and the time between consecutive accesses is small (high temporal locality);
- A block is accessed frequently in short duration, but then is not accessed for some time, and then is accessed again frequently within short periods (repetitive bursty behavior);
- a block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

Figure 2 shows the four 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 a block from L1 due to interference should allow it to reside in L2, since it is still needed. A block from the second category should be placed in L1 but not in L2. This block will be heavily referenced for a while, so it should reside in L1, but once evicted it will not be needed again soon, and so it need not reside in L2. A block in the third category will be placed in L2 but not L1. L1 will thus be reserved for blocks that are heavily referenced, and blocks not heavily referenced will not severely affect performance, since they will only reside in L2 while being accessed. Finally, a block from the last category will bypass the cache hierarchy [22], and will be stored in neither L1 nor L2. It is to be noted that if the consecutive accesses to the same block in the third category are very far away, this category can be reduced to the fourth category.

The advantage of having an adaptive block placement scheme becomes more obvious with an example. Assume we have the memory accesses shown in Figure 3(a). These instructions are accessing four different cache blocks
tt X, Y, Z, and W. Assume, for the sake of the example, that we have a direct-mapped L1 cache and a 2-way set associative L2 cache; and that all blocks accessed in that example map to the same set at L1 and at L2. Figure 3(b) shows the timeline for these accesses for each block. Every tick represents an access. If we try to map the four categories we have discussed above to these four blocks then block X must be put in L1 only because it has bursty access, then period of no access, then is accessed again. Blocks Y and Z are placed in L2 only because they are consistently accessed but the time between successive accesses is not very short. Finally, block W will not be cached at all because it is rarely accessed. Figure 3(c) shows the hits and misses for both L1 and L2 for a traditional LRU scheme. In that part we do not enforce inclusion, that is, a block victimized at L2 does not victimize corresponding block at L1, if any. A quick look at hits and misses from a traditional LRU reveals a couple of things. First, hit rate at L1 is 1/11 and for L2 is 2/10 (a hit at L1 does not need an L2 access). The second thing is that L2 has been accessed ten times out of eleven references. If we now look at Figure refexample(d) where the adaptive scheme is used, and both caches are assumed empty at the beginning, we find a better situation. When block X



(a) twolf



(b) art

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

is brought into the cache hierarchy after a compulsory miss, it is put into L1 but not L2. Blocks Y and Z are brought into L2 but not L1. Block W is not brought into any of the caches. The hit rate at L1 is now 4/11 and at L2 is 3/7, both of them higher than the traditional LRU. Moreover, L2 was accessed only seven times, which reduces the energy consumption.

Our goal is to design a system that captures the behavior of different cache blocks and categorizes each into one of the four categories we identify. The ultimate goal, of course, is to satisfy most of the references from L1. Keeping L1 size fixed, and because L1 cannot, until now at least, be fully associative, we are trying to decrease conflict misses by decreasing contentions on L1 sets. We do so by keeping some blocks away from L1.

Figure 4 shows one possible implementation of ABP. The main component is the *Behavior Catcher* (BC). BC keeps track of all the address requests generated by the processor. Thus, BC keeps snooping on the address bus coming out of the processor and toward 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 decides the category of the incoming block, if possible. If the access misses in the L2 and a memory reference is required, the incoming block is placed according to the decision of the BC. That is, it will go to L1, L2, both, or neither depending on the BC's decision. If no decision can be made, the hierarchy will follow a traditional inclusive 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 will not go to L1, and the required word will be delivered 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 will be delivered to L1. The BC is updated each time the processor generates a memory request. This update operation is not on the critical path, and will not affect access latency. Note that ABP hierarchies do not differentiate between loads or stores: they are all simply treated as memory accesses. Because the BC is triggered[1] only after L1 misses, a block that has been designated before as "L2 only" can now become "L1 and L2" if the BC decision has changed based on the access pattern.

4.1 ABP Design

Figure 4 shows the design of the BC. It consists of a small direct-mapped cache. The address decoder

[1]BC is triggered to make a decision after an L1 miss. However, BC keeps track of all address requests generated by the processor all the time.

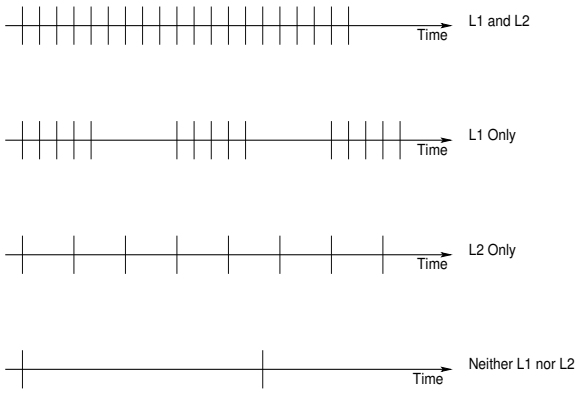


Figure 2: Different Patterns for Block Accesses (A vertical line presents a block access.)

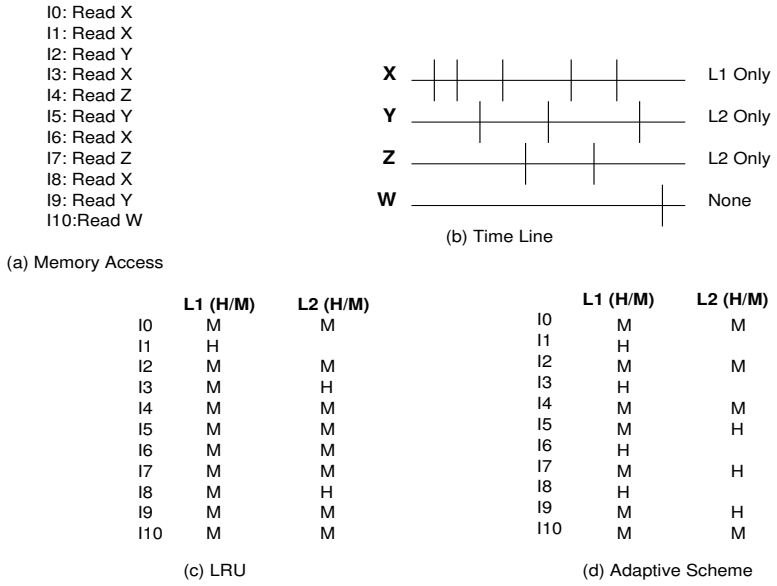


Figure 3: Example of LRU vs Adaptive Scheme

consists only of tag and set, because no offset is needed. Each entry of data array consists of a *pattern*, or string of bits representing the history of the corresponding cache block. A 1 indicates that a block is accessed, and a 0 means the block is not accessed. When an address is delivered to the BC as part of the update operation, if the block is present in the table, a 1 is entered to the right of the pattern entry, and the pattern is shifted 1 bit to the left. To approximate temporal reference activity, every X cycles we insert a 0 to the right of each pattern, and the entry is shifted one bit left. Therefore each address has a pattern that represents its *access history*. This presentation can be thought of as an approximation the patterns presented in Figure 2 where each vertical line is represented with a 1.

When the BC is presented with an address about which to make a decision, the address decoder splits that address into TAG and SET. If the tag of the entry specified by the SET matches the TAG, the corresponding pattern is copied 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 the block was heavily accessed. A majority of 0s means it was rarely accessed. A tie in a half means the block is frequently accessed, but not as heavily as if it had a majority of 1s. The figure shows a table that indicates the decision based on the majority rule of each half. Obviously, there are some cases where the system cannot make a decision, in which case the caches behave as a traditional inclusive hierarchy.

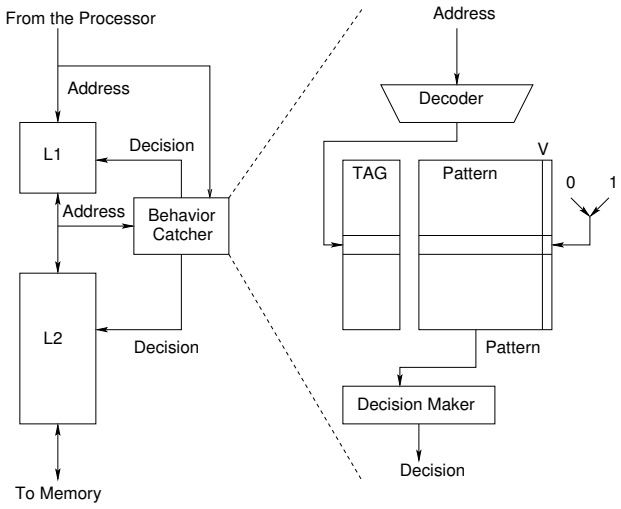


Figure 4: One Possible Implementation of ABP

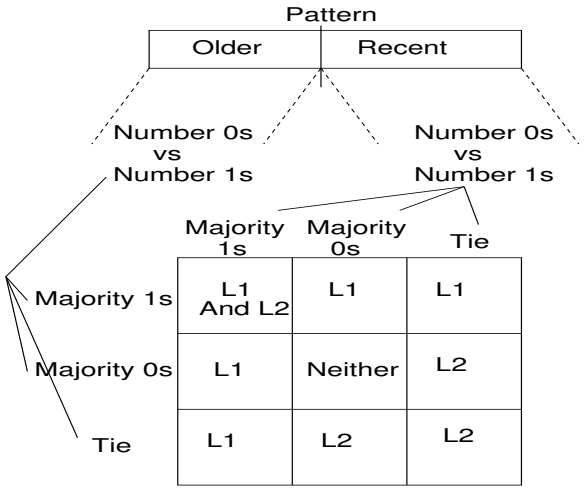


Figure 5: The Decision Making Process at ABP

4.2 Hardware Complexity

The proposed design requires very little hardware. We use CACTI 4.2 [23] to calculate the cost of our ABP hardware. We find that implementing ABP takes around 1% of unified L2 area, and consumes less than 4% of the power per access. The above numbers are for a table of 1024 entries, with 8-bit patterns per entry, and 8-bit tags. Our experiments show that ABP greatly reduces the number of write-backs to multiple levels of the hierarchy, and also reduces the scenario of a block staying in the cache for a long time without being accessed and starts leaking. Therefore ABP saves power. These savings more than offset the power consumed by the ABP hardware. Finally, each cache slot must be augmented by two bits to indicate whether the status of the block's last BC decision. A no-decision state is considered "L1 and L2".

ABP is not in the critical path, therefore it does not affect the access time of the cache hierarchy.

The design we present is but one way of building an ABP. More efficient designs are certainly possible, but here we focus on demonstrating the efficiency of the concept of using an adaptive global replacement policy for non-inclusive caches. Refining our design is part of currently ongoing work.

5 Experimental Setup

To conduct our experiments, we use a heavily modified simplescalar [24] with the ALPHA instruction set, modifying it to generate the statistics we need and implementing our set of proposed cache management modifications. We choose a fixed block size among all the caches in the hierarchy, similar to IBM POWER series [5] and many other state-of-the-art processors. Table 1 shows the fixed simulator parameters. Parameters not in the table use the default simulator values. We choose these values because they are typical of a current high-performance cores.

The table for the ABP has 1024 entries, keeps a pattern of 8 bits per entry, uses 1 byte of the block address as the tag (the least significant byte), with a refresh (0 inserted) at every L2 cache miss. Therefore, all what ABP requires is 2 Kbytes of data storage. We have chosen these parameters after running many experiments to explore the design space and find the best design in terms of price/performance. However, the best parameters are application dependent. We are currently enhancing the system to make it adjust depending on the application at hand, hence getting the best for each program.

We compare ABP with a conventional inclusive hierarchy with LRU replacement policy. The size of the caches at all levels is the same for all schemes.

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 access 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 SPEC CPU

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

2000 suite, both integers and floating point. We used simpoint [25] to skip the startup part of each application, and we simulated a representative 250M instructions. We use reference inputs with all the benchmarks. Table 2 shows the total number of committed references (loads and stores).

6 Results and Discussion

In this section we will discuss the results of several experiments done to assess the performance of ABP. It is important note that these results combine the performance of two things. First they present a proof-of-concept of the main idea of global block placement based on the four categories we have discussed in Section 4. Second, these results heavily depend on our implementation. Therefore, with different implementations results may differ.

6.1 ABP Behavior

The best way to assess the performance of ABP is to first see the decisions it makes and then see how these decisions are translated into a gain. Figure 6 shows the decisions made by ABP for each application in the benchmark suite. For some benchmarks, `applu, art, equake, lucas, mcf, and swim`, ABP decides to bypass *both* caches about 50% of the time. This happens because of one of two reasons, depending on each benchmark. The first reason is that the blocks are accessed very few times and not referenced again. The second reason is that the blocks are accessed frequently but the time between successive access is large enough to not be captured by the behavior pattern in ABP, such as `art` discussed in Section 3.

In this second case, blocks accessed frequently but across a large time frame, the overall performance is lightly affected, if at all. This is because the cache real-estate is reserved for more urgent blocks. Moreover, memory level parallelism [14] in the current state of the art memories is another way of masking the penalty of bypassing the cache for these blocks. However, we have not explored this possibility in this paper. Furthermore, from a power-consumption point of view, these blocks that are accessed across a long time frame will be staying, if brought into the cache, in the cache data-array for extended period of time without any access, increasing the leakage power dissipation of the cache, which is becoming a major factor in the current sub-micron era [26].

Another factor that encourages the fact of not bringing these blocks into the cache is the replacement policy. If a cache set is full, a replacement policy must be applied to choose a block to be evicted in case another block is fetched. This increases the power consumption as well as the hit latency. If the set has one of more empty slots, no replacement policy will be executed, or at least the LRU stack (in case Least Recently Used policy is used) will not be full. By decreasing the number of blocks coming into the cache, fewer sets will be full, and hence a new block can be fetched into the set without the need of any replacement policy to be conducted. Also we may save in bandwidth in case the evicted block is dirty.

In order to show that the effect of ABP on performance, Figure 7 shows the instruction per cycle for the different techniques. ABP, called *dynamic* in the figure, is better than the traditional inclusive hierarchy for most of the benchmarks. On average, ABP is about 9% better than traditional LRU. This percentage is expected to be higher if MLP is used, or in SMT scenarios because the effective cache size has increased.

6.2 Traffic Reduction

Another important factor in the efficiency of a cache hierarchy is its on-chip and off-chip traffic. In this section we look at the traffic between L1 and L2 caches and between L2 and off-chip memory. 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 sharing an L2 cache. If the traffic between each core and the shared L2 is high, this will put a lot of pressure on the on-chip interconnection network and on the number of ports required for the L2 cache to achieve adequate response time. Therefore, reducing the L1-L2 bandwidth requirement is of crucial importance for future systems. Also a high traffic from L2 (or last-level-cache) to off-chip puts a lot of pressure on memory ports, channels, and chip pins (which are not very scalable).

Figure 8 shows the writeback traffic between L1

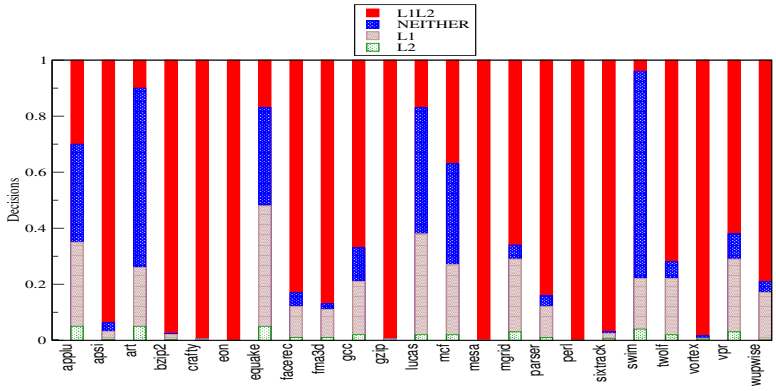


Figure 6: Decision Statistics of ABP

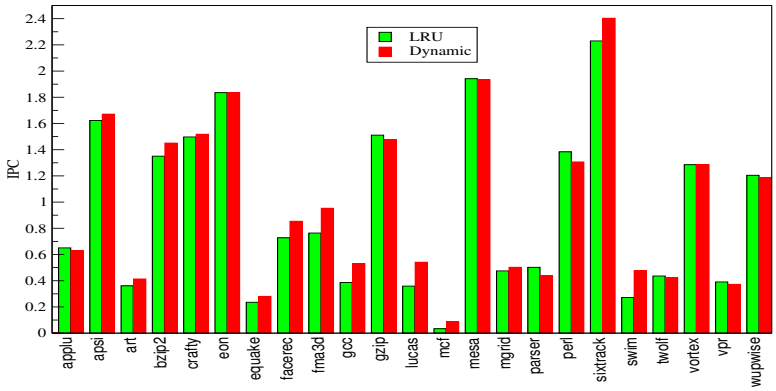


Figure 7: Instruction Per Cycle

and L2 caches, and between L2 and off-chip memory. We decided to look at the traffic due to writebacks, which is the traffic going toward the memory, in order to isolate the effect of ABP only. Because total traffic involves cache misses from instruction-cache, which we are not interested in, in our current study.

As we can see from these two figures, we have reduced the number of writebacks by about 15% from L1 to L2, and with more than 40% from L2 to main memory. This reduction is due to less replacements in caches, and hence less probability of victimizing a dirty block.

7 Conclusions and Future Work

In this paper we present a method for categorizing the access pattern of each block into one of four categories, and we present a possible design for doing so.

We found out that by using a table as small as 1 Kbytes, we enhance the overall performance by an average of 9% while reducing traffic from L1 cache to L2 cache by an average of 15%, and the traffic from L2 to main memory by an average of 40%. This is very useful when the number of cores per-chip increase. Because ABP makes its decision based on address streams so is independent of the number of cores.

We are currently working on enhancing our scheme by following several paths.

- Fine tuning ABP using several sensitivity studies
- Making ABP self-tuned by adding another level of *learning*
- Extending our concepts to both SMT processors and to CMP designs

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 Sympo-*

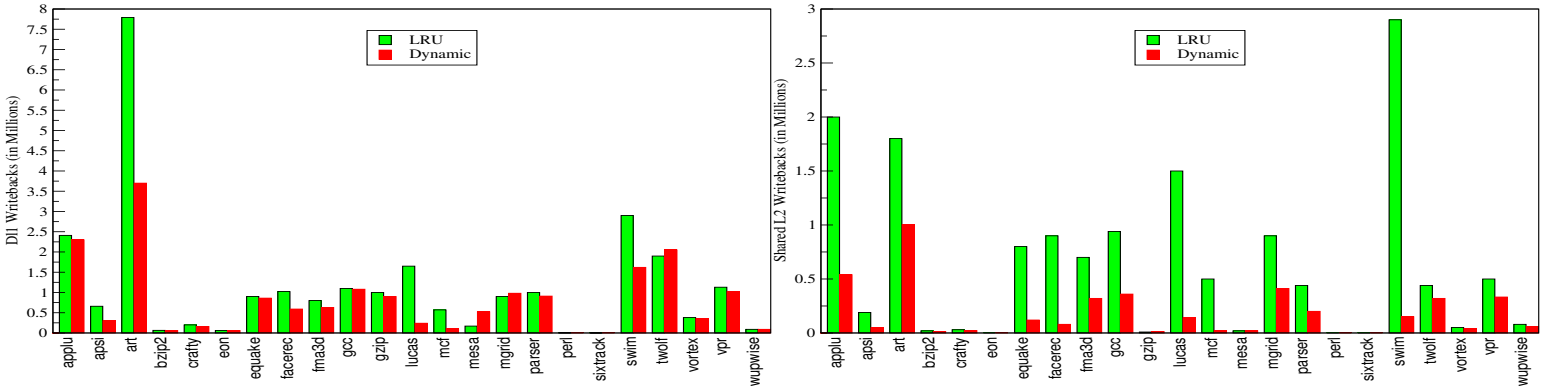


Figure 8: Total Writebacks Between L1 and L1, and Between L2 and Memory

sium 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," *, VOLUME = 46, NUMBER = 1, YEAR = 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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," in *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pp. 228–239, Jun. 2006.

[16] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Predictability of cache replacement policies,"

Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS, Sep. 2006. ISSN: 1860-9821.

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

[18] 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.

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

[20] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson, "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.

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

[22] 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.

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

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

[25] 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.

[26] 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.