Probabilistic Directed Writebacks for Exclusive Caches

Lena E. Olson
University of Wisconsin-Madison
lena@cs.wisc.edu

Mark D. Hill
University of Wisconsin-Madison
markhill@cs.wisc.edu

Abstract

Energy is an increasingly important consideration in memory system design. Although caches can save energy in several ways, such as by decreasing execution time and reducing the number of main memory accesses, they also suffer from known inefficiencies: the last-level cache (LLC) tends to have a high miss ratio while simultaneously storing many blocks that are never referenced after being written back to LLC. These blocks contribute to dynamic energy while simultaneously causing cache pollution.

Because these blocks are not referenced before they are evicted, we can write them directly to memory rather than to the LLC. To do so, we must predict which blocks will not be referenced. Previous approaches rely on additional state at the LLC and/or extra communication.

We show that by predicting working set size per program counter (PC), we can decide which blocks have low probability of being referenced. Our approach makes the prediction based solely on the address stream as seen by the level-one data cache (L1D) and thus avoids storing or communicating PC values between levels of the cache hierarchy. We require no modifications to the LLC.

We adapt Flajolet and Martin's probabilistic counting to keep the state small: two additional bits per L1D block, with an additional 6KB prediction table. This approach yields a large reduction in number of LLC writebacks: 25% fewer for SPEC on average, 80% fewer for graph500, and 67% fewer for an in-memory hash table.

1. Introduction

"Half the money I spend on advertising is wasted; the trouble is I don't know which half."
— Attributed to John Wanamaker
"Half the energy I spend on caching is wasted; the trouble is I don't know which half."
— This paper's authors

Caches are an effective way to decrease execution time and reduce the number of main memory accesses, making them essential for both high performance and low energy. They are so successful that they have been recursively replicated: today's systems have not only first- and second-level caches, but third-level caches are becoming increasingly common as well.

Because caches are so essential, much effort has been expended in improving their performance and energy efficiency. However, there are several inefficiencies that last-level caches (LLCs) suffer from, chief among them a **high local miss ratio** and a **large percentage of blocks that are never referenced before eviction**. Previous studies have found and our experiments confirm that the miss ratio at the LLC is high for many common applications [2,15]. We find that for the SPEC 2006 benchmarks, on average 50% of LLC accesses result in misses, and for memory-intensive applications such as an in-memory hash table, the miss ratio is much higher – over 95%. At the same time, many of the blocks that are written to the LLC after eviction from the level-one data cache (L1D) are never re-referenced. We find that these **useless blocks** account for over 80% of the writebacks to the LLC for SPEC, and over 95% for memory-intensive workloads.

There have been a number of approaches proposed to remedy the problems caused by indiscriminately writing back all data evicted from L1 to L2 and L3. These approaches have included modifying the LLC replacement policy to preferentially evict blocks predicted to be dead [22], prefetching into predicted dead blocks [21], and doing cache *bypassing* – that is, choosing to evict some blocks from L1 straight to memory [11].

A commonality of much previous work is that it makes decisions about insertion, bypass, and replacement policies at the LLC itself, and therefore any information from the executing process that might aid in the decision (such as program counter (PC) or instruction sequence) must be transferred between levels of cache, adding overheads in hardware and complexity. Many of these approaches do not use bypassing, instead varying only the insertion and replacement policies, which can reduce cache pollution, but do not eliminate useless writebacks to the cache – blocks are still written to the cache, they are simply evicted earlier. This is especially problematic with emerging technologies such as Spin-Transfer Torque RAM (STT-RAM), where writes are far more expensive than reads both in latency and energy [1].

Even in approaches with bypassing, placing the decision at the LLC prevents reducing the traffic to the LLC. In hierarchies with multiple levels, logic must be replicated at each cache level where bypassing is possible.

An assumption behind much of the prior work is that the reuse behavior of blocks in the LLC can best be determined by observing behavior at the LLC itself. We offer the insight that *LLC block locality can be predicted by observing the stream of addresses generated by the CPU*. We dynamically determine the cardinality of the set of blocks referenced by each PC, because this gives an estimate of per-PC working set size. Predicted working set size is a good indication of temporal locality because if the working set does not fit into the cache, blocks will likely suffer from a high miss rate and a tendency to be evicted before being referenced.

We show that by adding a working set prediction table in parallel with L1D accesses, we can effectively *direct* blocks to the appropriate level of the memory hierarchy upon L1D eviction. We therefore call our approach *Directed Writebacks*.

We use Flajolet and Martin's theory result on *probabilistic counting* [9] to limit state in the prediction table. Probabilistic counting is well suited for our case, where we require only a coarse estimate of the number of blocks accessed per PC. We can tolerate several binary orders of magnitude of error because we are only trying to predict whether the working set is smaller or larger than the cache size.

Our prediction table entries are each 65 bits per PC plus a 32 bit tag. For a 32KB L1D, a 1MB L2, and a 4MB LLC, we show that a 6KB structure per L1D is sufficiently large for reasonable prediction. Because this structure is not on the critical path for L1 accesses, it does not add any latency overhead for L1 hits.

We evaluate our design with an exclusive cache hierarchy in this work. Exclusive caches have significant effective storage size advantages in deep cache hierarchies [11,31]. Directed writebacks are also conceptually simpler with exclusive caches because blocks are guaranteed to only be found in one cache at a time. This allows bypassing dirty blocks to memory without needing to worry about stale copies causing consistency problems. We qualitatively discuss how to extend this work to other cache hierarchies in Section 7.2.

The contributions of this paper are:

- demonstrating on real hardware that there are performance benefits to cache bypassing, even in the absence of hit rate improvement
- the insight that cache block reuse behavior can be predicted without communicating with the cache itself, but rather by simply observing the PC and address streams of memory references
- providing the first adaptation (to our knowledge) of theory's probabilistic counting to computer architecture (to coarsely estimate LLC working set size with small L1-like state)
- proposing *directed writebacks* and demonstrating that a simple predictor added in parallel with the L1D miss path to *direct* writebacks to L2, L3, or memory can reduce energy and improve performance in an exclusive cache hierarchy

The paper is organized as follows: we start by characterizing LLC block behavior for several classes of workloads and its energy impact. We also motivate this work by demonstrating potential energy and performance improvement on real hardware. We then discuss probabilistic counting and show how it can be used to store working set size with very small area overhead, and evaluate the results, including in comparison with SHiP [33], another strategy for improving LLC behavior. Finally, we place our work in the context of related work.

2. Characterization of LLC Behavior

Table 1. System configuration.

L1D	Private. 8-way 32KB. LRU.
L2	Private. 16-way 1MB. LRU.
L3 (LLC)	Shared. 32-way 4MB. LRU.

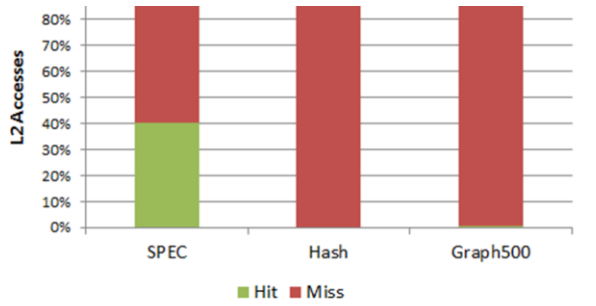

Figure 1. Breakdown of hits/misses at L2

Figure 2. Breakdown of hits/misses at L3

2.1 Workload Selection

We evaluate workloads from a variety of sources. The SPEC 2006 workloads have been previously characterized by Jaleel et al. [15] and Sandberg et al. [30], among others. Because we anticipate that directed writebacks will have greater benefits for applications with very large working sets, we also characterize graph500 [12] and a simple hashtable microbenchmark meant to approximate an in-memory key-value store. To allow execution of large numbers of instructions, we use a Pin-based simulator [24] and run for 10 billion instructions. We validated it against results gathered using gem5 [4] with Ruby and previous work and find that our characterizations agree. Unless otherwise noted, we run hashtable with a 1GB table and graph500 with scale size 25 (equivalent to a storage size of approximately 8GB). In all cases, we assume two cores and a three-level exclusive cache hierarchy with parameters shown in Table 1. We assume for this evaluation that all levels of the hierarchy use a true LRU replacement policy.

We examine pairs of workloads in a multiprogrammed environment. We run with all pairs of workloads.

We target our work to large workloads, such as graph500 and hashtable, but include SPEC 2006 to ensure that our

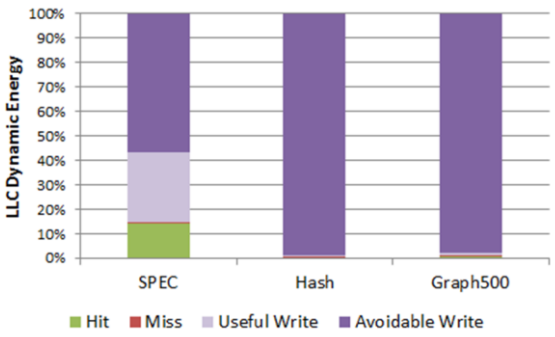

Figure 3. Dynamic energy breakdown at L3.

approach does not cause undue performance degradation even for smaller workloads.

2.2 LLC Local Miss Ratio

We investigate the L2 and L3 local miss ratio for a variety of applications and show our results in Figure 1 and Figure 2. We find that on average 60% of L2 accesses and over 65% of L3 accesses are misses for SPEC 2006. For graph500 and hashtable, more than 99.5% of accesses miss at L2, and more than 99% at L3. Our results are similar to those found by Jaleel et al. [15]. The high miss ratio indicates that there is potential for changes to the cache hierarchy to significantly improve hit rate and thus performance.

2.3 Avoidable Writebacks

In addition to a high local miss ratio, a large percentage of blocks written to the L2 and L3 are evicted before they are used (each miss once the cache is full causes another block to be evicted unused). In an exclusive cache hierarchy, these writebacks are *avoidable* – that is, if we avoided doing the writebacks and instead directed the writeback to a different level of cache or to memory, we would not later incur an extra miss.

2.4 Energy Impact of Avoidable Writebacks

Avoidable writebacks to the L2 and L3 contribute to execution energy in two major ways: the dynamic energy of doing needless cache writebacks, and the static energy when cache pollution results in increased run time. We discuss each in turn.

Dynamic energy: There is a cost in dynamic energy to doing cache writes. We model local L3 energy using Cacti [26]. As shown in Figure 3, we find that L3 local energy going to avoidable writes is approximately 57% on average for SPEC workloads. For graph500 and hashtable, over 98% of L3 dynamic energy goes to avoidable writebacks.

Static energy: Avoidable writes also contribute to static energy by potentially increasing the execution time for the program. By storing useless blocks in the cache, there is less room for useful blocks, which can increase the miss rate and cause execution overheads. In addition, the writeback traffic to the cache can interfere with read requests for blocks, increasing miss latency.

In addition to reducing the execution time of the program performing the bypassing, bypassing can also reduce cache pollution affecting other programs running concurrently, preventing a program which trashes the L3 (a "gobbler" in the terminology of Sandberg et al. [30]) from slowing down other programs. Sandberg et al. have previously demonstrated an improvement with multiprogrammed SPEC workloads when bypassing is employed.

Finally, by reducing the amount of data stored in the L2 and L3, bypassing can potentially pave the way toward using energy optimizations such as putting some cache ways to sleep to further reduce static energy.

3. Analysis of Avoidable Writebacks

To determine whether we can predict where blocks should be directed upon writeback, we did a characterization of avoidable writebacks to L3.

3.1 Program Counter Association

We examined the last PC used to access each block before eviction from L1D to see if there was a relationship between PC and L3 behavior. For SPEC, we find that on average, about 10% of blocks come from PCs that always generate avoidable writebacks to L3, and a further 35% come from blocks that do so at least 99% of the time. This suggests that by using a simple policy of always bypassing writebacks originating from these PCs, we could eliminate 45% of avoidable writebacks to L3 while incurring very few additional misses.

Similar results were found by Holloway et al. [14], who examined *problem stores* – that is, static instructions that are responsible for many later load misses. They found that a few static instructions were responsible for many of the misses. Intuitively, these results make sense because many L3 misses or no-reuse blocks are likely caused by the same type of accesses: loads and stores to data structures that are too large to fit in the L3. Other previous works that have shown a correlation between access PC and L3 behavior include SHiP [33].

The predictive value of the PC can be exploited to make effective decisions about the destination of directed writebacks from L1D.

3.2 Software Approach on Existing Hardware

To estimate the potential benefits of directed writebacks in a real system, we ran tests on an AMD A8-3850 processor with a 1MB private per-core L2 and no L3. This model has an exclusive cache hierarchy and limited support for software bypassing. When the processor encounters a non-temporal prefetch instruction (i.e. `prefetchnta`), it both prefetches the block to the L1D and also sets a sticky bit indicating that when the block is evicted, it should bypass the LLC. In this way, it implements directed writebacks to a limited extent.

We manually inserted `prefetchnta` instructions into two workloads: graph500 and hashtable. For graph500 we inserted four static instructions (using inline assembly) and

for hashtable we inserted two, one in the get function and one in set. To distinguish between the impact of the additional prefetch vs. the bypass hint, we also created a version of the workloads with `prefetch` instead of `prefetchnta`. In all cases, the additional prefetch instruction was inserted directly before the access. We then ran the workloads using `perf` to collect performance counter data. The base implementation of graph500 ran for 70 minutes, while hashtable ran for 8 minutes.

As shown in Figure 4, adding bypass hints drastically reduces the number of writebacks to the LLC, as expected, while just using prefetch does not reduce the number of LLC writebacks. In addition, Figure 5 shows that there is a substantial speedup – approximately 1.5X – for both graph500 and hashtable. We do not see a speedup for graph500 with just `prefetch` added, and for hashtable the speedup is less than with `prefetchnta`, indicating that it is indeed from bypassing and not the prefetching.

Interestingly, we do not see an increase in hit rate, indicating that the benefit is at least partially due to a decrease in writebacks to the LLC. When a writeback request is made to a full LLC (common case), a block must be evicted to memory, and when there is a high miss rate at L1D, this can result in saturating the bandwidth to memory, causing performance degradation.

These results suggest that directed writebacks can improve both performance and energy in a real system. However, because this approach requires annotating static instructions, we could not easily use it to evaluate our hardware directed writebacks mechanism. The remainder of the paper deals with the Pin-based cache model, but we include the results on real hardware to motivate this work.

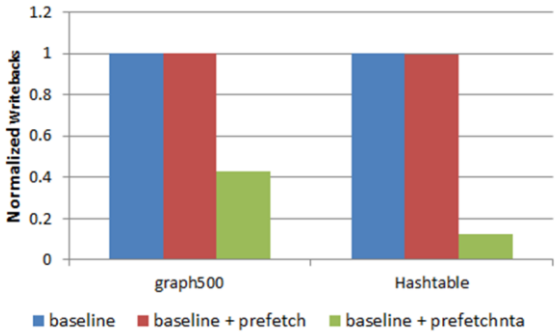

Figure 4. Normalized writebacks to the LLC with the original binary, with prefetch instructions inserted, and with non-temporal prefetch instructions inserted.

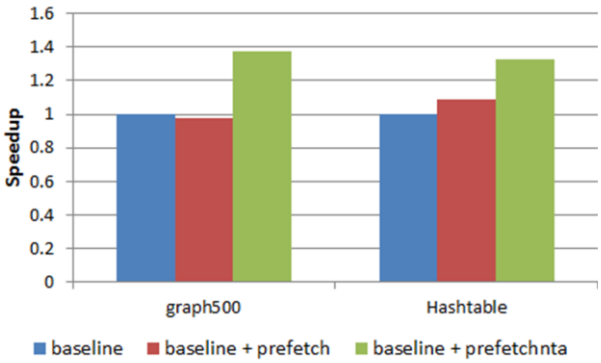

Figure 5. Speedup of graph500 and hashtable on hardware, using prefetch and non-temporal prefetch instructions

4. Directed Writebacks by Probabilistic Counting

As discussed in Section 3.1, PC is correlated with block reuse behavior. However, propagating and storing PC information at every level of the cache hierarchy adds overhead; therefore, it is desirable to make predictions upon L1D insertion, without any feedback from the other levels of cache. We hypothesize that if a single PC accesses more blocks than can fit in the entirety of a particular level of cache, it has poor locality and we will not benefit from storing blocks from this PC in that level. We thus use the PC and address stream to identify the working set size for each PC, and set a bit to direct the block to the appropriate cache or to memory upon eviction. In addition, we do not rely on any feedback from the cache hierarchy.

4.1 Probabilistic Counting

To determine working set size, we keep a probabilistic count of the number of distinct cache blocks accessed per PC. Probabilistic counting algorithms are ideal for this use case because we do not require an exact count and can tolerate significant error, but are constrained to a very limited amount of state. This algorithm requires $\log_2 N$ bits of state to store N distinct events.

Here we provide a brief overview of Flajolet and Martin's probabilistic counting [9,10]. It is a deep theoretical result with the original paper cited 800 times and the papers directly citing it being in turn cited over 10,000 times (per Google Scholar).

The probabilistic counting algorithm states that a sequence of events can be counted by generating a random number for each event, setting the bit in the bit vector that corresponds with the most significant 1 in the random number, and then counting the number of set bits in the bitmap from MSB to LSB until a cleared bit is encountered. The intuition behind this approach is that every random number will have a 50% chance of having the first 1 be in the MSB position, 25% of it being in the next position, 12.5% of it being in the position after, and so on. Thus, on average we would expect to see 2^{N+1} random numbers before bit N is set. We can therefore estimate that if the most significant 16 bits are set, then we expect to have seen 2^{16} = 64K distinct events.

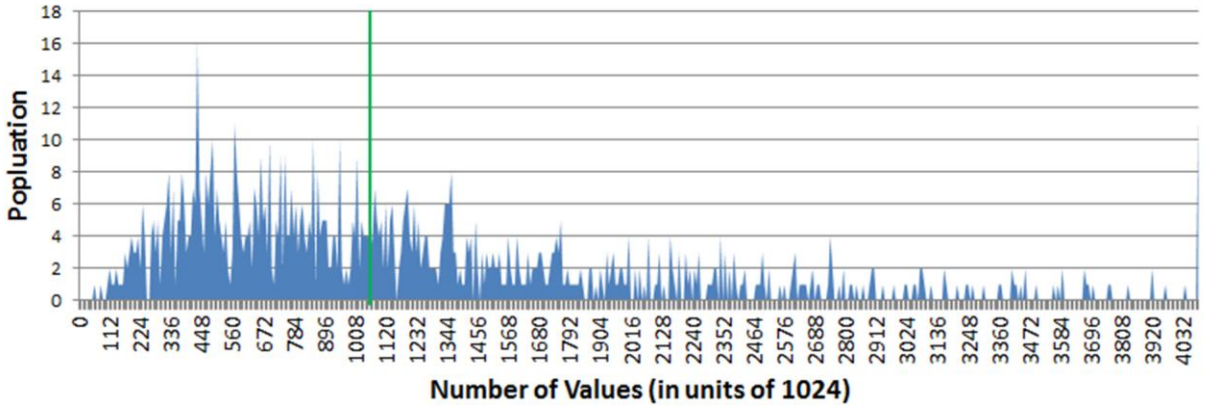

Figure 6. Distribution of number of values before map predicts 1 (binary) million. The vertical bar shows the expected value.

Table 2. An example address stream and working set map modification for a single PC.

Block Address	Hashed Address	Working Set Map	Working Set Size Prediction
Initial		0000000000	0
0x40fe	1011111110	10000000000	$2^1 = 2$
0x410b	0110110000	11000000000	$2^2 = 4$
0x0f0e	1100011010	11000000000	$2^2 = 4$
0x40fe	1011111110	11000000000	$2^2 = 4$
0xabcd	0001010101	11010000000	$2^2 = 4$

Flajolet and Martin discuss using this approach for counting distinct entities. In this case, each entity is hashed with a function that produces outputs that appear random but which are deterministic by input, preventing frequently occurring entities from inflating the count.

We use this approach to estimate working set size by hashing the address stream for each PC. We hash on the granularity of a cache block and perform the hash for every L1D insertion. The resulting bit vector is stored per PC, and when the working set is estimated to be larger than a level of the cache hierarchy, we choose to bypass all accesses from that PC. Table 2 shows the working set map and predicted working set size for a sample stream of references from a single PC. Note that in the final entry, the modification of the working set map from 1100… to 1101… does not increase the predicted size, because there is a cleared bit before the added set bit.

We evaluate our proposal using SpookyHash [16], a public domain non-cryptographic hash function which has a fast software implementation. However, in a hardware implementation we would choose a non-cryptographic hash function that is fast and low-power to implement in hardware, such as one of the H3 class of functions [5,29].

4.2 Empirical Analysis of Probabilistic Counting

To better understand the properties of probabilistic counting, we tested how many distinct random numbers we needed to generate to reach a map with the first 20 bits set; that is, with predicted 2^{20} = 1,048,576 values. We ran 1000 trials. A histogram of our results can be seen in Figure 6.

We find that the average value is slightly higher than predicted: 1,246,372. However, the distribution has a long tail and the median is close to our desired value: 1,028,292. The minimum value is only 58,023 (over 4 binary orders of magnitude off) and the greatest is 7,431,854, or almost 3 binary orders of magnitude in error. Hence, any design decision must take into account that the predicted working set size can be quite different from actual working set size, and that the distribution has a long tail.

5. Hardware Design for Directed Writebacks

We will first describe a naïve version of the design for directed writebacks, and then show how it can be refined. We make two additions to the traditional exclusive cache design. First, we add 2 bits per line to the L1D to indicate whether the block should be directed to memory, to L3, or to L2 upon eviction. Second, we add the *working set size prediction table*, a table of PCs and working set maps, in parallel with the L1D miss access path. In contrast to previous approaches, we make no changes to the L2 or L3. An overview of our design is shown in Figure 7.

The working set size prediction table is a small structure added in parallel with the L1D which is accessed on L1D misses. Each entry contains a working set map, which in our naïve implementation is 32 bits; we discuss the size of the map further in Section 5.5. We also tag each entry with 32 bits of the PC. In total, each entry is 64 bits. Because the prediction table is located at L1D, in contrast to most previous work which places it at the L2 or L3, it must be replicated for every core. However, this has the benefit that it makes it simple to disable the policy for a specific process (similar to a hardware prefetcher).

Our additions do not affect L1D access time because the hash function and prediction table are not on the critical path; the predictor can return the prediction to the L1D after the cache fill is completed, since it is only relevant on

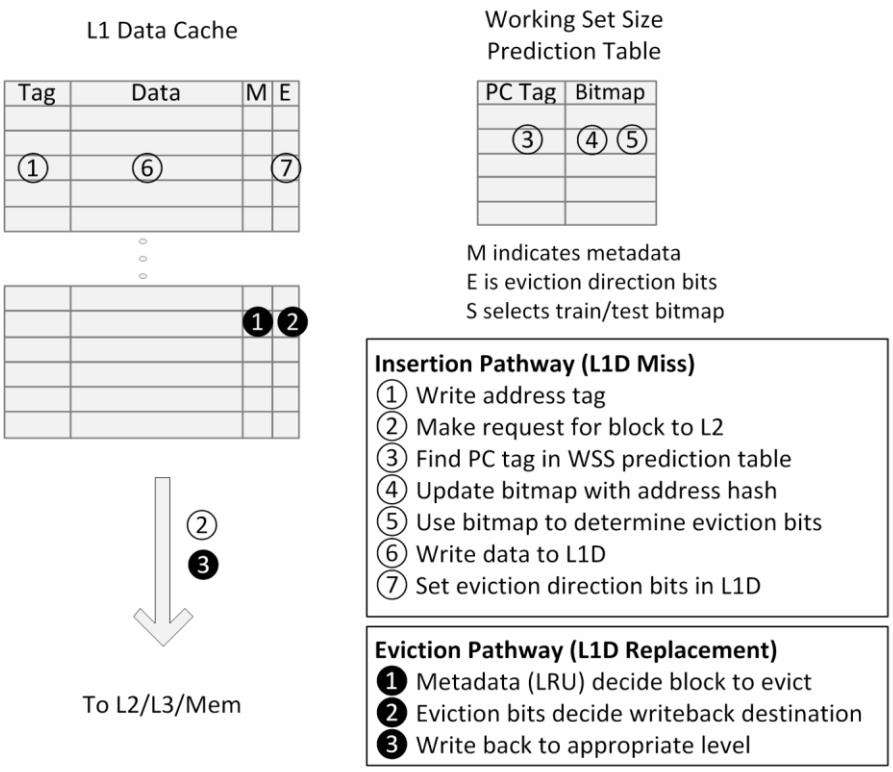

Figure 7. High-level diagram of the L1D and working set size prediction table. The cache and the table are shown as direct-mapped for the sake of clarity.

The hit pathway is not shown, as it is unchanged from the baseline design.

The insertion pathway shows the steps that take place when an L1D access misses and must be inserted in the L1D. Note that some of the steps may overlap (e.g. the data request can overlap the accesses to the working set size prediction table).

The eviction pathway shows the steps that take place when a block is evicted from the L1D. The only change from baseline is checking the eviction direction bits and directing the writeback accordingly.

an eviction. The hashing logic and prediction table are accessed only on L1D miss.

This approach requires no communication between the L1D and other levels of the hierarchy, in contrast with previous work. In the version we model, we make no changes to replacement policies at any level; however, we discuss the possibility of using information from the working set size prediction table to improve insertion and replacement decisions in Section 7.1.

5.1 Working Set Size Selection

We evaluated a range of probabilistic working set sizes to determine when directed writebacks should be sent to each level of the memory hierarchy. Although the caches are a known size in our system – 1MB and 4MB, respectively – we do not want to simply bypass when the working set size is predicted to be greater than 1 or 4MB. As seen in Section 4.2, predicted working set size can have significant error. In addition, the penalty for a block bypassing a cache where it would have hit is far higher than that of writing back a block that will be evicted before use. We predict that because of this asymmetric penalty, we should choose to bypass only when we think there is high probability that the working set size is bigger than the cache. We choose to have directed writebacks to the L2 when the predicted working set size is less than 16MB, to L3 when the predicted working set size is less than 64MB, and to memory otherwise.

5.2 Prediction Table Size

We next determine an appropriate size for the working set size prediction table. We choose to map multiple PCs to the same entry, similar to what a branch prediction table does. However, there is an important distinction between branch predictors and our predictor: the branch predictor learns from feedback, while our predictor does not receive any feedback. In addition, the penalty for erroneously predicting that a block should bypass a level of cache is much greater than failing to bypass an avoidable write. Because sharing entries between different PCs can result in erroneously decision to bypass, we cannot share entries between different PCs.

Instead, we use a small set-associative buffer, tagged by 32 bits of the PC, to store working set maps. We use a modified LRU policy to determine which entry to evict. Because it takes many accesses to populate the buffer, we supplement LRU eviction to bias in favor of keeping entries which correspond to known large working sets.

We find that a 4-way structure with 512 entries has a very small number of populated evictions – for 20 billion instructions, we find that on average 4 and at most 33 populated entries are evicted.

5.3 Reducing Prediction Error

Probabilistic counting has a large standard deviation, as noted in Section 4.2. This error can be reduced by populating multiple maps and combining their results for the estimated working set size. This technique is discussed

in Flajolet and Martin [9] and reduces the error by a factor proportionate to the square root of the number of maps.

To avoid needing multiple hash functions, we use the *log(M)* least significant bits of the hashed value to select one of *M* hash maps. We then expect each individual map to predict 1/*M* of the size of the working set for the PC. To simplify the logic, rather than averaging the number of bits set, we choose to predict the working set size based on a vote between the hash maps.

For our results, we use 4 hash maps. If each map is 32 bits, this results in a total of 128 bits per predictor entry. We predict that the working set is 4 times the largest working set that at least 3 of the 4 maps predict.

5.4 Temporality

Some programs may have phase behavior or PCs that access many addresses, but in a blocked pattern. We would like to distinguish between PCs that access *N* distinct addresses out of *N* accesses, as opposed to those that access *N* distinct addresses in greater than *N* accesses. For example, a PC that accesses 1 GB of addresses but in a blocked fashion of 512 KB at a time should be cached in the L2, but the bitmap will progressively become more populated until our predictor sets the blocks to bypass.

By periodically clearing the bitmaps, we can limit the working set we track to the previous *N* accesses. However, we wish to do this without having to pay the overhead of training every time we clear the map. Therefore, we use two maps: one that is updated upon misses, and the other which is used to decide the destination for the directed writeback. We switch between the two maps with a probability proportional to the size of the working set we wish to predict. For example, if we wish to predict when 1 million distinct accesses have occurred, we switch with a probability of 1 in 4 million. Upon switching, we clear the map.

Because the working set size we wish to predict is different for deciding whether to bypass the L2 and L3, we need a mechanism to reset the bitmap with a different frequency for the two predicted working set sizes. To do this, we simply clear the most significant bits of the working set with a probability proportional to the size of the L2. This allows us to include a notion of temporal locality in our decision.

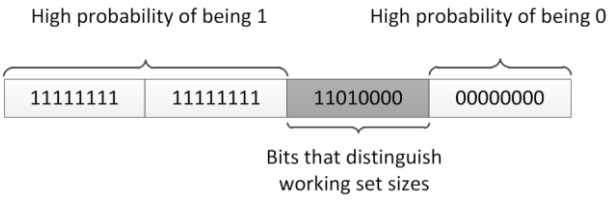

Figure 8. Breakdown of the bitmap: only the highlighted bits are helpful for predicting working set size.

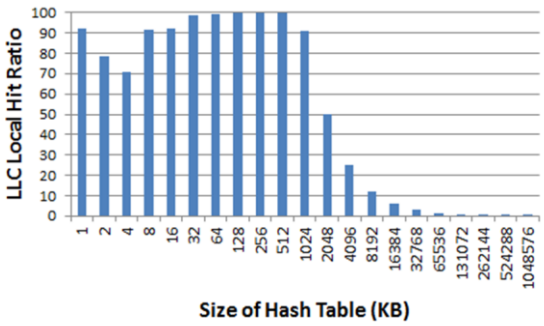

Figure 9. LLC local hit ratio for a hashtable ranging in size from 1KB to 1GB. L1D size is 32KB and LLC Size is 1MB.

5.5 Improvement: Smaller Working Set Maps

In the implementation described above, we use 2 128-bit working set maps per entry, for a total of 256 bits per entry. We will now show how to reduce this to 64 bits per entry.

Since we treat all predicted working sets below 4MB the same, we require that the first 16 bits are set. Because the probability of each bit being set is twice as high as the one to its right, it is very likely that if bits 13, 14, and 15 are set, then 31-16 are set as well. Similarly, 32 bits is sufficient to predict up to 2^{32} blocks, which corresponds to 256GB of memory. Since this is much larger than current cache capacity, these bits could be truncated as well. We show this in Figure 8, where the most significant bits have a very high probability of being set for ranges we are interested in, and the least significant bits have a high probability of being 0. It is only the highlighted bits that help us predict working set size.

We modified the working set map to take a 32 bit hashed address, as before, but to only set a bit in the working set map if the first 1 was between bits 15 and 7, corresponding to a working set size of between 4MB and 1GB. This results in working set maps that are only 8 bits each; with 4 working set maps used in each voting group and a test and train set for each entry, this results in 64 bits per entry.

5.6 Prediction Error Discussion

In general, we err on the side of writing back useless blocks rather than bypassing them. This approach has several implications beyond dealing with the asymmetrical penalty for bypassing useful blocks and writing back useless blocks.

A non-intuitive consideration is that having a working set greater than the cache capacity does not necessarily mean that bypassing all accesses will maximize hit rate. For example, if the addresses accessed are random and the working set is twice the size of the cache, we would expect that writing all blocks to cache would result in a 50% hit rate from that PC, versus 0% when all blocks bypass.

We demonstrate this relationship with the hash table benchmark in Figure 9, using data from real execution on an AMD machine with 32KB L1D and 1MB LLC. As can

be seen, LLC local hit ratio is very high when the working set is too large for the L1D but fits in the LLC: for table sizes of 32KB to 1MB. However, even when the table size is twice the capacity of the LLC (2MB), we see that the hit ratio does not drop to 0; rather it drops to 50%. Even when the hash table size is 16 times the LLC, we still see approximately 6% hit rate.

Ideally, if application working set size is twice the size of a cache, we would like to direct ½ of the evictions to that cache and ½ onwards: if the accesses are randomly distributed, then we would have the 50% hit rate from the baseline case, as well as the 50% reduction in LLC writebacks from the evict-to-memory case. If the accesses are sequential, writing back half the data is even better; we expect to see 50% hit rate vs. 0% hit rate for the no-bypass case.

By clearing the bitmaps probabilistically and selecting our threshold size for bypassing conservatively, we are likely to write blocks back to the caches during some intervals even for PCs that sometimes bypass. However, the above explains why this can actually have benefits in terms of hit rate.

5.7 Variability

Because the working set prediction map is probabilistic, there is increased variability in program execution time. Because of this, our approach may not be well-suited for real-time systems with hard deadlines. However, caches themselves complicate the calculation of worst-case execution times, and our approach is no worse than for other cache prediction strategies.

An extension to prevent unacceptable performance degradation for critical applications or for applications which perform poorly with directed writebacks is to allow the system to disable directed writebacks via an architectural register, similar to how hardware prefetching can be turned off in current systems. This has the additional benefit of allowing the system to control the priorities of different applications; by disabling directed writebacks for an application with strict quality of service guarantees while using directed writeback to prevent other applications from polluting the caches.

5.8 Context Switches

Although our evaluation is Pin-based and as such is limited to user mode, it is important for a hardware design to take context switches into account. Our directed writeback predictor requires ~6KB of state to perform well with our cache hierarchy. This is too much state to save on a context switch; in addition, we would prefer that our approach not require system modifications. However, we also do not want to flush all state on a context switch, because it requires many accesses to repopulate the working set size prediction table. Therefore, we advocate tagging the table by physical address of the PC to allow state to persist across context switches. Then, once the process is rescheduled, it may still have entries persisting in the prediction table, especially since populated entries are less likely to be evicted with our replacement policy.

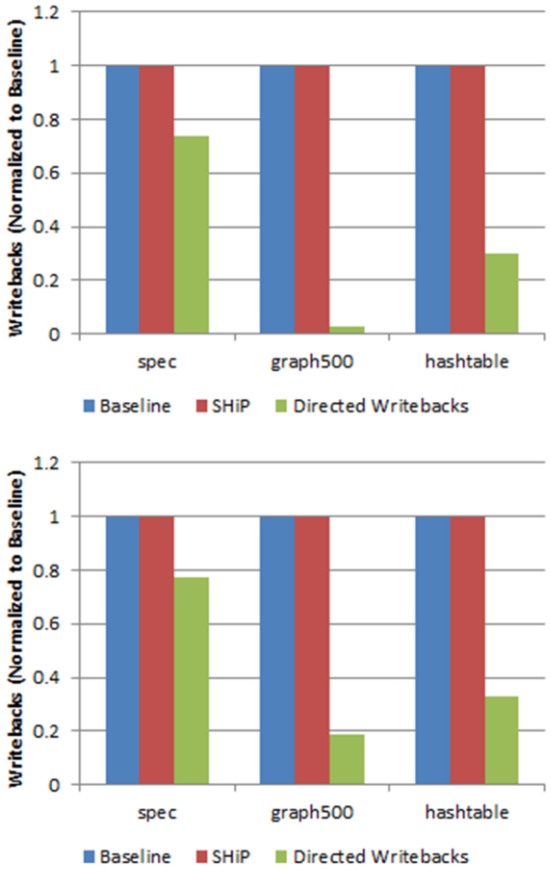

Figure 10. Writebacks to L2 and L3 by configuration, normalized to the baseline.

6. Evaluation

6.1 Methodology

We use a 3-level exclusive cache model fed by Pin traces, as described in Section 2.1. We run each configuration for all pairs of workloads; all runs are multiprogrammed. Our cache configuration uses LRU, and consists of 32LB private L1Ds, 1MB private L2s, and a 4MB shared L3. We use traces with 10 billion instructions, for a total of 20 billion instructions per workload pair.

In addition to running the baseline and our approach, we also compare against SHiP. SHiP uses PC information at the LLC to determine whether a line should be inserted in MRU or LRU position. We implemented SHiP in our model to compare it to our approach, although there are some important differences: SHiP only operates at the LLC and not the L2, and it does not do bypassing. Originally, SHiP was designed for inclusive caches, so we modified it slightly for an exclusive cache hierarchy. We update the Signature History Counter Table on either L3 hit or

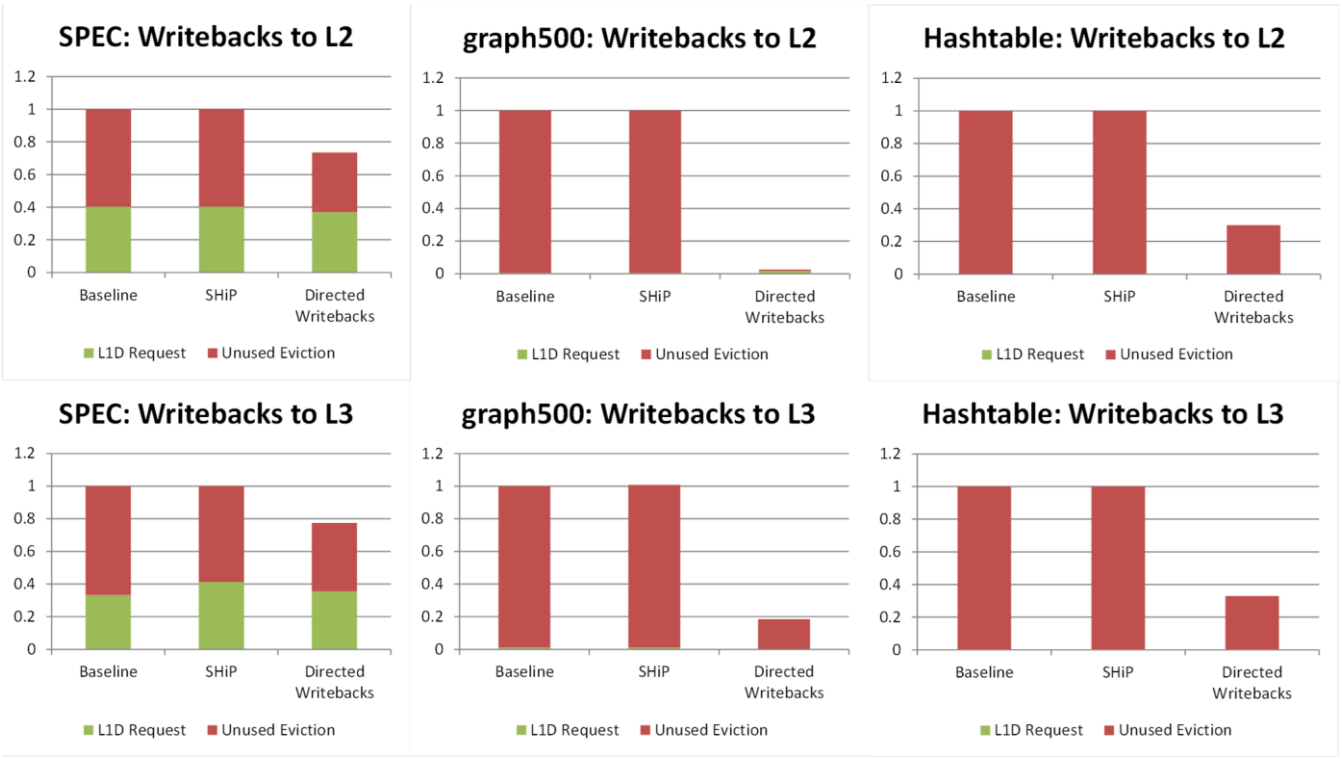

Figure 11. Outcome of blocks written to L2 and L3 in each configuration, normalized to baseline, showing the effect of bypassed blocks in directed writebacks. A block may leave the cache either as a result of a request from the L1D (hit) or by being evicted.

eviction, and we do not use set sampling but rather train with all sets. We use the hashed PC as the signature.

6.2 L2 and L3 Writebacks

We observe the number of writebacks to the L2 and L3 to measure the benefit of directed writebacks compared to the baseline and SHiP, neither of which do any bypassing. The results are shown in Figure 10; lower is better.

For SPEC, the benefits are relatively small; we eliminate approximately a quarter of writebacks at L2 and L3. However, for graph500 and hashtable, where most blocks written back to the L2 and L3 are evicted before use, we see greater benefits. For graph500, directed writebacks eliminates more than 97% of L2 and 80% of L3 writebacks. For hashtable, directed writebacks reduces writes by 70% at L2 and 67% at L3.

Finding: The directed writebacks approach dramatically reduces the number of writebacks to L2 and L3, especially for workloads with very large working sets.

6.3 Usefulness of Stored Blocks

In the previous section we showed that directed writebacks avoids writing many blocks back to L2 and L3; we now show what percentage of the blocks that were bypassed would have been referenced before eviction. We determine the eventual outcome of every block written back to the L2 and L3 in the baseline: it can either leave the level of cache through satisfying a request from the L1D (a hit) or by being evicted. We show the breakdown of block fates in Figure 11; a breakdown by SPEC benchmark is in Figure 12.

Most of the writebacks to L2 and L3 that our technique eliminates are to blocks that would otherwise be evicted unused. We see benefits for all three classes of workloads, but they are much greater benefits for graph500 and the hash table, where the working sets are large. For a few SPEC workloads (perlbench and zeusmp), our technique actually causes an increase in writebacks to the LLC. This occurs when blocks that would have hit in the L2 are bypassed to the L3. However, this effect only occurs for two benchmarks, and in most cases our design decreases the number of writebacks.

We also compare to SHiP, and find that in general, SHiP has a higher L3 hit rate than directed writebacks for SPEC. However, SHiP does not perform bypassing, and thus cannot reduce writebacks. In addition, the large workloads we study are not able to significantly benefit from SHiP.

Finding: Directed writebacks reduce the number of writebacks to L2 and L3, particularly for workloads with very large working sets, such as graph500 and the hash table. In general, the bypassed writebacks do not result in a significant increase in misses, because the bypassed blocks would not have been referenced before eviction.

6.4 Hardware Overhead

The hardware overhead of directed writebacks has two components: the directed writeback prediction table and the

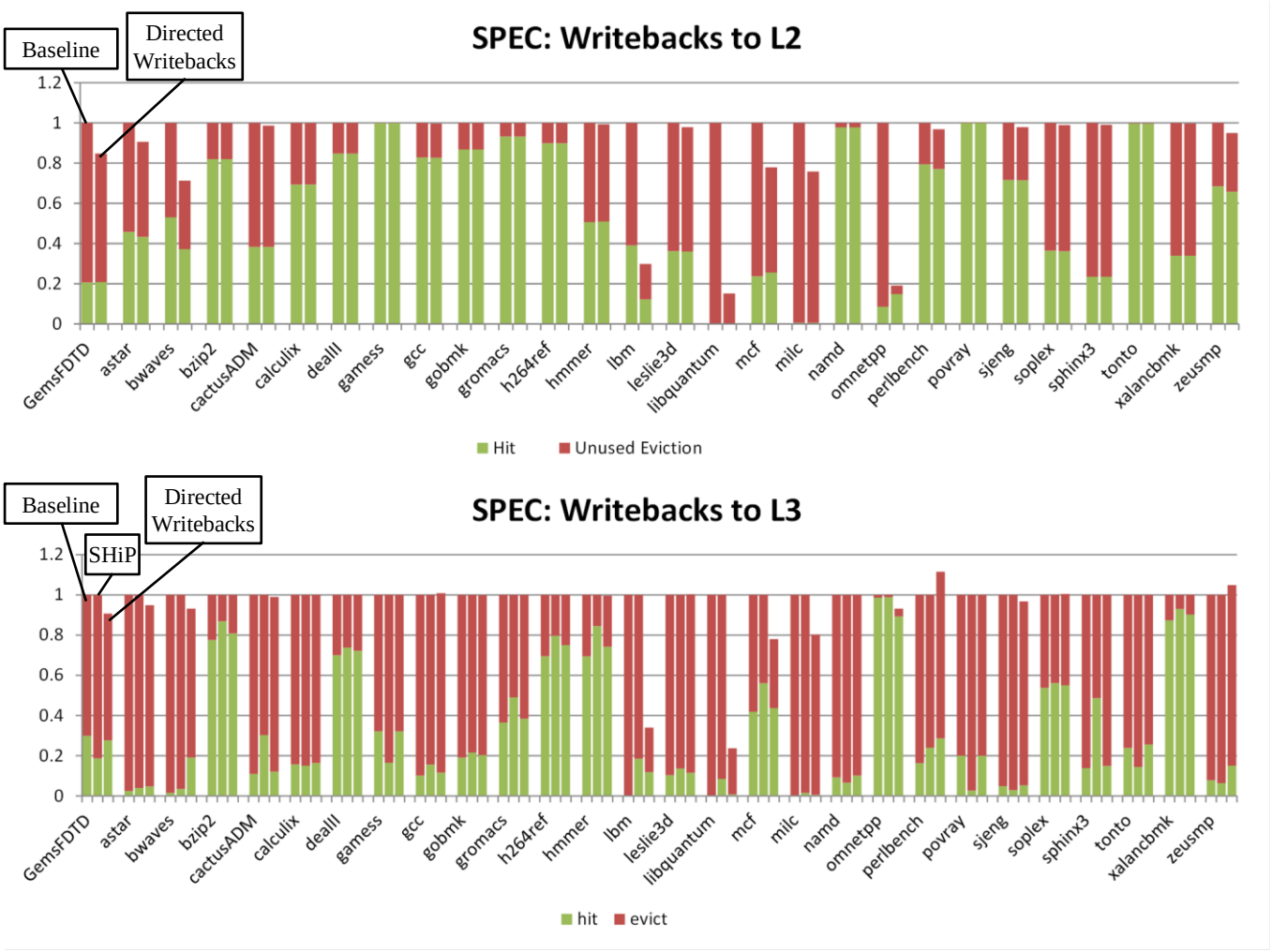

Figure 12. Outcome of blocks written to L2 and L3 in baseline/SHiP, showing effect of bypassed blocks in directed writebacks. Breakdown of SPEC benchmarks. For L2, each set of bars has the baseline on the left and directed writebacks on the right; for L3, there are three bars: baseline, SHiP, and directed writebacks.

added eviction direction bits in the L1D. The prediction table consists of 512 entries, each containing 2 groups of 4 8-bit maps, a 32-bit PC tag, and 1 bit to select between test/train bitmaps. This is a total of approximately 6KB. The eviction direction bits in the L1D are 2 bits per cache line, for a total of 128 bytes for a 32KB cache.

For comparison, there are also two sources of hardware overhead for SHiP: the Signature History Counter Table, and the storage of the 14-bit PC signature at each level of the cache hierarchy. SHiP uses a 16K-entry table, for a total of approximately 32KB of overhead. The extra bits in the cache add approximately 1KB of overhead to each 32-KB L1D, 32KB to each 1 MB L2, and 128KB to the 4 MB L3.

Although SHiP has a larger prediction table, it requires only one for the entire system, while our approach requires one per L1D. In addition, 6KB is a large structure to add at the L1D. However, this is less serious than it appears. This table is accessed only on the miss pathway, which means that it only needs to be as fast as a L2 hit.

7. Extensions

We touch upon two possible extensions of this work.

7.1 Replacement Policy

Although we evaluated this approach only in the context of bypassing writebacks, it is clear that knowing approximate working set size could have other applications as well. For example, it could be used to modify replacement policies by selecting insertion position in an LRU or pseudo-LRU policy.

In particular, blocks associated with PCs with predicted working set sizes close to the size of the cache could be inserted there in LRU position. This avoids pollution but has a smaller penalty for misprediction. Alternately, set dueling could be used to adjust the bypass policy or disable directed writebacks for workloads with a high rate of mispredictions that blocks should bypass.

7.2 Inclusive Caches

It is straightforward to implement directed writebacks in an exclusive cache hierarchy, because when a block is evicted from the L1D, it is guaranteed not to be in the LLC. However this is not the case for inclusive and non-inclusive hierarchies. Here we briefly discuss how directed writebacks can be modified to work with these caches.

For non-inclusive caches, a block may be in both L1D and the LLC. If there is a stale copy in the LLC, we must invalidate it by sending an invalidation when we direct a dirty block to bypass a level of the cache. Even in this case, we will still get some of the benefits of reduced writeback traffic, because an invalidate message is less expensive than writing back data and, if the block is not present, needing to evict another block.

For inclusive caches, the block must already be resident in all levels of the cache hierarchy. However, we can invalidate it early to reduce cache pollution. In addition, if the block is dirty, we can choose to invalidate it at the LLC rather than writing back the updated copy. Alternately, "inclusive" caches with some ways that contain only tags have been proposed to maintain the coherence benefits of inclusive caches while allowing techniques like bypassing [2,3,34]. These caches with dataless ways can be augmented with directed writebacks.

8. Related Work

The idea of preventing useless blocks from polluting the LLC and of implementing cache bypassing for energy and performance reasons is not new. We summarize related work and how ours differs from it.

There have been many cache management proposals that have aimed to discover temporal and spatial locality with the goal of reducing LLC pollution. One prominent example is SHiP [33], which used a variety of policies including PC-based to predict whether lines would have near-immediate or distant reuse intervals. There are several disadvantages to this approach: first, it requires that a signature be stored along with each block in every level of cache and communicated between them; it also does not eliminate useless writes but only helps mitigate their effects.

Jiménez [17] proposes an approach to select between several Insertion-Promotion Vectors (IPVs), which determine order of insertion and promotion of LLC blocks. The IPVs are chosen using an off-line genetic approach, and then selected between with set-dueling. Other approaches that modify insertion policy include Qureshi et al. [27] and Keramidas et al. [20].

The Reuse Cache [2] distinguishes between temporal and re-use locality, and does not store blocks without reuse locality in the LLC. It stores tags for all blocks, but only stores data for ¼ of them – those that show reuse locality. It relies on the insight that one extra miss for a frequently accessed block will not significantly degrade performance. However, it is unsuited for the class of applications that use the LLC well, because it reduces LLC capacity. Gupta et al. [13] propose a similar bypass scheme for inclusive caches.

Guar et al. [11] use trip count and use count to do bypass and insertion in exclusive LLCs. Their approach involves maintaining a per-block trip counter.

PriSM [25] aims to control per-core LLC occupancy by changing eviction probabilities, because this provides more flexibility than way-partitioning.

Kharbutli et al. [23] augment each LLC cache line with an event counter to keep track of number of accesses to a set between accesses to a particular entry, and also track the number of accesses to a cache line. They then use this information to improve the replacement algorithm, as well as doing bypassing. This approach requires 21 bits per cache line, plus a 40KB prediction table. This approach also does not allow setting different policies for different processes.

Sandberg et al. [30] demonstrate that a significant performance improvement can be attained by adding non-temporal prefetch instructions to workloads. Their approach relies on profiling with a representative data set, and requires recompilation, in contrast with ours, which is automatic and in hardware. However, the benefits they demonstrated on real machines are likely to also apply with our approach, since both are ultimately PC-based.

Tyson et al. [32] and Dybdahl and Stenström [7] use counter-based prediction tables to determine when to bypass blocks.

Some previous approaches focus on L1D bypassing. Chi and Dietz [6] present an early work on selective cache bypassing; they use compiler support. Etsion et al. [8] point out that if a resident block in a cache is chosen at random, it is unlikely to be a highly-referenced block, but if an access is chosen at random, it is likely to be to a highly-referenced block. They use this insight to determine the core working set. Johnson et al. [18,19] use a Memory Address Table to count the number of references to a memory region. Rivers et al. [28] compare address vs. PC-based prediction.

Probabilistic counting for cardinality was initially developed by Flajolet and Martin [9] and has subsequently been analyzed and improved upon by many theoreticians. However, to our knowledge, it has not been applied in the field of computer architecture.

9. Conclusion

With directed writebacks, we provide a simple way to reduce the number of writebacks to the L2 and L3, which can be useful for several reasons. It limits interference between workloads, reduces dynamic energy, and reduces the traffic and hence contention between L2, L3, and memory. We show how a small predictor can be used to eliminate writebacks to many blocks that would otherwise be evicted before use, without adding significant overhead. We also show how probabilistic counting can be applied to computer architecture.

10. References

1. Ahn, J., Yoo, S., and Choi, K. DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture. *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture*, (2014).
2. Albericio, J., Ibáñez, P., Vinals, V., and Llabería, J.M. The Reuse Cache: Downsizing the Shared Last-level Cache. *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM (2013), 310–321.
3. Basu, A., Hower, D., Hill, M., and Swift, M. FreshCache: Statically and dynamically exploiting dataless ways. *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, (2013), 286–293.
4. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., and Wood, D.A. The gem5 simulator. *Computer Architecture News (CAN)*, (2011).
5. Carter, J.L. and Wegman, M.N. Universal Classes of Hash Functions (extended abstract). *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, (1977), 106–112.
6. Chi, C.-H. and Dietz, H. Improving cache performance by selective cache bypass. *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, (1989), 277–285 vol.1.
7. Dybdahl, H. and Stenström, P. Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination. In C. Jesshope and C. Egan, eds., *Advances in Computer Systems Architecture*. 2006, 52–66.
8. Etsion, Y. and Feitelson, D. Probabilistic Prediction of Temporal Locality. *Computer Architecture Letters 6*, 1 (2007), 17–20.
9. Flajolet, P. and Martin, G.N. Probabilistic counting. *Foundations of Computer Science, 1983., 24th Annual Symposium on*, (1983), 76–82.
10. Flajolet, P. and Martin, G.N. Probabilistic counting algorithms for data base applications. *Computer and System Sciences 31*, 2 (1985), 182 – 209.
11. Gaur, J., Chaudhuri, M., and Subramoney, S. Bypass and insertion algorithms for exclusive last-level caches. *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, (2011), 81–92.
12. graph500 --The Graph500 List. http://www.graph500.org/.
13. Gupta, S., Gao, H., and Zhou, H. Adaptive Cache Bypassing for Inclusive Last Level Caches. *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, (2013), 1243–1253.
14. Holloway, A. and Sohi, G. Characterization of Problem Stores. *Computer Architecture Letters 3*, 1 (2004), 9–9.
15. Jaleel, A. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www.glue.umd.edu/ajaleel/workload*, (2013).
16. Jenkins, B. *Spookyhash: a 128-bit noncryptographic hash*. 2013.
17. Jimenez, D.A. Insertion and Promotion for Tree-based PseudoLRU Last-level Caches. *Proc. of the 46th Annual IEEE/ACM International Symp. on Microarchitecture*, ACM (2013), 284–296.
18. Johnson, T.L., Connors, D.A., Merten, M.C., and Hwu, W.W. Run-time cache bypassing. *Computers, IEEE Transactions on 48*, 12 (1999), 1338–1354.
19. Johnson, T.L., Merten, M.C., and Hwu, W.W. Run-time Spatial Locality Detection and Optimization. *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, (1997), 57–64.
20. Keramidas, G., Petoumenos, P., and Kaxiras, S. Cache replacement based on reuse-distance prediction. *Computer Design, 2007. ICCD 2007. 25th International Conference on*, (2007), 245–250.
21. Khan, S.M., Daniel A. Jimnez, D.B., and Falsafi, B. Using dead blocks as a virtual victim cache. *PACT '10: Proceedings of International conference on Parallel architectures and compilation techniques*, (2010).
22. Khan, S.M., Jimenez, D.A., and Yingying, T. Sampling Dead Block Prediction for Last-Level Caches. *MICRO'10: Proceedings of International Symposium on Microarchitecture (MICRO)*, (2010).
23. Kharbutli, M. and Solihin, D. Counter-Based Cache Replacement and Bypassing Algorithms. *Computers, IEEE Transactions on 57*, 4 (2008), 433–447.
24. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI'05: ACM SIGPLAN conference on Programming language design and implementation*, ACM (2005).
25. Manikantan, R., Rajan, K., and Govindarajan, R. Probabilistic Shared Cache Management (PriSM). *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, (2012), 428–439.
26. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.
27. Qureshi, M.K., Jaleel, A., Patt, Y.N., Jr, S.C.S., and Emer, J. Adaptive Insertion Policies for High-Performance Caching. *Proceedings of the 34th Annual International Symposium on Computer Architecture*, (2007).
28. Rivers, J.A., Tam, E.S., Tyson, G.S., Davidson, E.S., and Farrens, M. Utilizing Reuse Information in Data Cache Management. *Proceedings of the 12th International Conference on Supercomputing*, ACM (1998), 449–456.
29. Sanchez, D., Yen, L., Hill, M.D., and Sankaralingam, K. Implementing Signatures for Transactional Memory. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (2007).
30. Sandberg, A., Eklöv, D., and Hagersten, E. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, (2010), 1–11.
31. Sim, J., Lee, J., Qureshi, M.K., and Kim, H. FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, (1993), 321–332.
32. Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A.R. A Modified Approach to Data Cache Management. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press (1995), 93–103.
33. Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, J.S.C., and Emer, J. SHiP: Signature-based Hit Predictor for High Performance Caching. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM (2011), 430–441.
34. Zhao, L., Iyer, R., Makineni, S., Newell, D., and Cheng, L. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, (2010).