

Sampling Dead Block Prediction for Last-Level Caches

Samira Khan*, Yingying Tian*, Daniel A. Jiménez*[†]

*Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA

[†]Barcelona Supercomputing Center
Barcelona, Catalonia, Spain

skhan@cs.utsa.edu, ytian@cs.utsa.edu, djimenez@acm.org

Abstract—Last-level caches (LLCs) are large structures with significant power requirements. They can be quite inefficient. On average, a cache block in a 2MB LRU-managed LLC is *dead* 86% of the time, i.e., it will not be referenced again before it is evicted.

This paper introduces *sampling dead block prediction*, a technique that samples program counters (PCs) to determine when a cache block is likely to be dead. Rather than learning from accesses and evictions from every set in the cache, a sampling predictor keeps track of a small number of sets using partial tags. Sampling allows the predictor to use far less state than previous predictors to make predictions with superior accuracy.

Dead block prediction can be used to drive a dead block replacement and bypass optimization. A sampling predictor can reduce the number of LLC misses over LRU by 11.7% for memory-intensive single-thread benchmarks and 23% for multi-core workloads. The reduction in misses yields a geometric mean speedup of 5.9% for single-thread benchmarks and a geometric mean normalized weighted speedup of 12.5% for multi-core workloads. Due to the reduced state and number of accesses, the sampling predictor consumes only 3.1% of the of the dynamic power and 1.2% of the leakage power of a baseline 2MB LLC, comparing favorably with more costly techniques. The sampling predictor can even be used to significantly improve a cache with a default random replacement policy.

I. INTRODUCTION

The miss rate in the last-level cache (LLC) can be reduced by reducing the number of *dead* blocks in the cache. A cache block is *live* from the time of its placement in the cache to the time of its last reference. From the last reference until the block is evicted the block is *dead* [13]. Cache blocks are dead on average 86.2% of the time over a set of memory-intensive benchmarks. Cache efficiency can be improved by replacing dead blocks with live blocks as soon as possible after a block becomes dead, rather than waiting for it to be evicted.

Figure 1 depicts the efficiency of a 1MB 16-way set associative LLC with LRU replacement for the SPEC CPU 2006 benchmark `456.hmmer`. The amount of time each cache block is live is shown as a greyscale intensity. Figure 1(a) shows the unoptimized cache. The darkness shows that many blocks remain dead for a long time. Figure 1(b) shows

improvement in efficiency by driving a replacement and bypass policy with a sampling dead block predictor.

Dead blocks lead to poor cache efficiency [15], [4]. In the least-recently-used (LRU) replacement policy, after the last access, each dead block wastes time moving from the most-recently-used (MRU) to the LRU position before it is evicted.

A. Dead Block Prediction

Dead block prediction can be used to identify blocks that are likely to be dead to drive optimizations that replace them with live data. Performance improves as more live blocks lead to more cache hits. However, current dead block prediction algorithms have a number of problems that make them unsuitable for the LLC:

- They incur a substantial overhead in terms of prediction structures as well as extra cache metadata. For instance, each cache block must be associated with many extra bits of metadata that can change as often as every access to that block. Thus, the improvement in cache efficiency is paid for with extra power and area requirements.
- They rely on an underlying LRU replacement algorithm. However, LRU is prohibitively expensive to implement in a highly associative LLC. Note that this problem extends to other recently proposed improvements to caches, including adaptive insertion policies [19], [7].
- Due to the large number of memory and instruction references tracked by these predictors, they must be either very large or experience a significant amount of destructive interference in their prediction tables resulting in a negative impact on accuracy.
- Predictors that use instruction traces do not work in a realistic scenario involving L1, L2, and L3 caches because a moderately-sized mid-level cache filters out most of the temporal locality.

B. Sampling Dead Block Predictor

This paper presents dead block prediction technique based on *sampling*. Previous dead block predictors find correlations

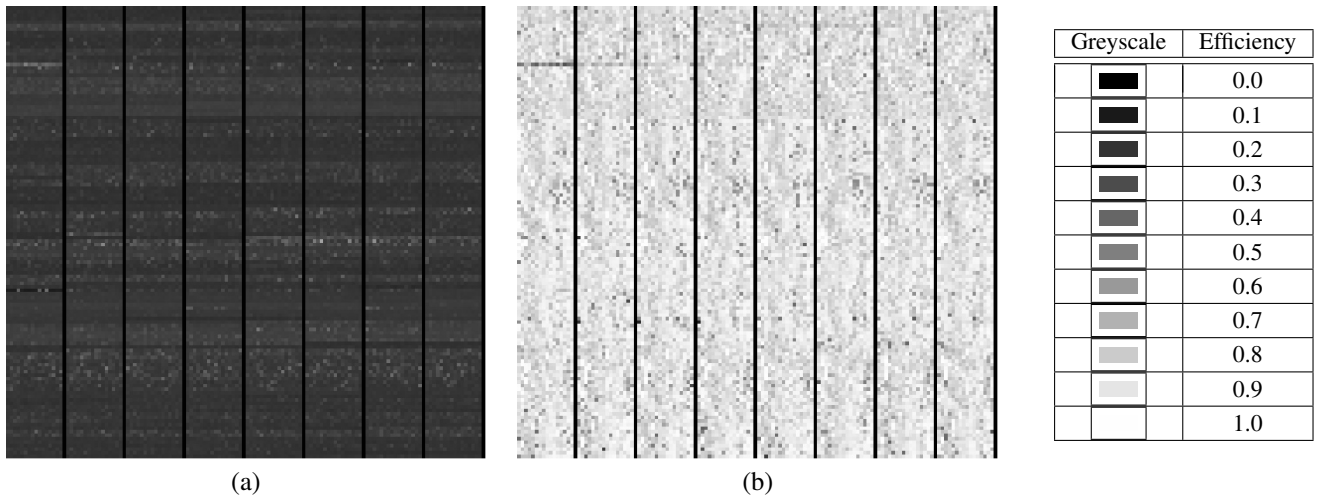


Fig. 1: Dead block replacement and bypass bring the cache to life. Efficiency (i.e. live time ratio) shown as greyscale intensities for `456.hmmer` for (a) a 1MB LRU cache and (b) a dead-block-replaced cache using a sampling predictor. Darker blocks are dead longer. Efficiency is 22% for (a) and 87% for (b).

between observed patterns of memory access instructions and cache evictions, learning when program behavior is likely to lead to a block becoming dead. The new prediction technique exploits four key observations, allowing it to use far less power and area while improving accuracy over previous techniques:

- Memory access patterns are consistent across sets, so it is sufficient to sample a small fraction of references to sets to do accurate prediction. For example, in a 2MB cache with 2,048 sets, sampling references to 1.6% of sets is sufficient to predict with accuracy superior other schemes that learn from every reference. The predictor keeps track of a small number of sets using partial tags replaced by the LRU policy, allowing it to generalize predictions not only to a large LRU cache, but also to a large randomly replaced cache.
- A separate sampling structure, kept outside the LLC, can be configured differently to provide improved accuracy. For instance, we find that, for a 16-way set associative LLC, a 12-way associative sampler provides superior accuracy while consuming less state.
- Previous predictors use reference traces or counters for each block. However, simply using the address of the last memory access instruction provides sufficient prediction accuracy while obviating the need to keep track of access patterns for all blocks. Indeed, we find that trace-based predictors do not work when a mid-level cache filters much of the temporal locality from the stream of memory instructions that access the LLC. Metadata associated with cache blocks is significantly reduced with a proportional reduction in power.
- Dead block predictor accuracy can be improved by using a skewed organization inspired by branch prediction research.

In this paper, sampling prediction is explored in the context of a dead block replacement and bypass policy. That is, the replacement policy will choose a dead block to be replaced before falling back on a default replacement policy such as random or LRU, and a block that is predicted “dead on arrival” will not be placed, i.e., it will bypass the LLC.

The sampling predictor is well-suited to improve the performance of a cache with a default random replacement policy. Sampling prediction reduces average cache misses by 7.5% compared with no improvement for a previously proposed predictor based on counting. This reduction in cache misses for a combination random/dead-block replaced cache results in a 3.4% average speedup over a baseline LRU cache. Other recent placement and replacement policies depend on the LRU policy. The sampling predictor does not.

Because of the small amount of predictor state and cache metadata, the sampling predictor consumes less than half the leakage power of a counting predictor. It uses only 3.1% of the dynamic power of a baseline cache on average, compared with 11% for the counting predictor.

II. RELATED WORK

A. Dead Block Predictors

Previous work introduced several dead block predictors and applied them to problems such as prefetching and block replacement [13], [15], [11], [5], [1].

1) *Trace Based Predictor*: Dead block prediction was introduced by Lai *et al.* [13]. The Lai *et al.* predictor is used to prefetch data into dead blocks in the L1 data cache. This reference trace predictor (hereafter *reftrace*) collects a trace of instructions addresses that access a particular block on the theory that, if a trace leads to the last access for one block then the same trace will lead to the last access for other blocks. The *signature*, or truncated sum of these instruction addresses,

is used to index a prediction table. A trace based predictor is also used to optimize a cache coherence protocol [12], [23] and perform dynamic self-invalidation [14]. A skewed trace-based predictor is used to identify a pool of dead blocks in the L2 cache to be used as a “virtual victim cache” into which LRU victims from hot sets can be stored [10].

2) *Time-Based Predictor*: Hu *et al.* propose a time-based dead block predictor [5] that learns the number of cycles a block is live and predicts it dead if it is not accessed for twice that number of cycles. This predictor is used to prefetch into the L1 cache and filter a victim cache. Abella *et al.* propose a similar predictor [1] based on number of references rather than cycles for reducing cache leakage.

3) *Cache Burst Predictor*: Cache bursts [15] can be used with trace, counting, or time based dead block predictors. A cache burst consists of all the contiguous accesses to a block while in the MRU position. The predictor predicts and updates only on each burst rather than on each reference. Cache bursts predictors limit the number of accesses and updates to the prediction table for L1 caches, improving power relative to previous work. Our sampling predictor also reduces the number of updates. However, cache bursts have been shown to offer little advantage for higher level caches, since most bursts are filtered out by the L1. Furthermore, cache bursts predictors also require significant additional metadata in the cache, while our predictor requires only one additional bit per cache block.

4) *Counting-Based Predictor*: Kharbutli and Solihin propose counting-based predictors for the L2 cache. The Live-time Predictor (LvP) tracks the number of accesses to each block. This value is stored in the predictor on eviction. A block is predicted dead if it has been accessed more often than the previous generation. Each predictor entry has a one-bit confidence counter so that a block is predicted dead if it has been accessed the same number of times in the last two generations. The predictor table is a matrix of access and confidence counters. The rows are indexed using the hashed PC that brought the block into the cache and the columns are indexed using the hashed block address. An Access Interval Predictor (AIP) is also described in the same paper, but we focus on LvP as we find it delivers superior accuracy. The counting based replacement policy chooses the predicted dead block closest to LRU as a victim, or the LRU block if there is no dead block. LvP and AIP are also used to bypass cache blocks which are brought to the cache and never accessed again.

5) *Other Predictors*: Another kind of dead block prediction involves predicting in software [25], [21]. In this approach the compiler collects dead block information and provides hints to the microarchitecture to make cache decisions. If a cache block is likely to be reused again it hints to keep the block in the cache; otherwise, it hints to evict the block.

B. Sampling for Cache Placement and Replacement Policy

Dynamic insertion policy (DIP) uses *set dueling* to adaptively insert blocks in either the MRU or LRU position

depending on which policy gives better performance [19]. Performance is sampled through a small number of dedicated sets, half using MRU placement and the other half using LRU placement. Our predictor also uses sampling to dynamically learn from program behavior, but in addition to sampling the access characteristics of data, it also samples the instructions that lead to that program behavior. We compare our dead-block-predictor-driven replacement and bypass policy to adaptive insertion in Section VII. A memory-level parallelism aware cache replacement policy also uses set-dueling to do sampling, relying on the fact that isolated misses are more costly for performance than parallel misses [20]. Thread Aware Dynamic Insertion Policy (TADIP) uses DIP in a multi-core context [7]. It takes into account the memory requirement of each executing program. It has a dedicated leader set for each core for determining the correct insertion position (MRU/LRU). This way thrashing workloads decide to insert in the LRU position while cache-friendly workloads continue to insert in the MRU position.

Keramidas *et al.* [9] proposed a cache replacement policy that uses sampling-based reuse distance prediction. This policy tries to evict cache blocks that will be reused furthest in the future.

LRU replacement predicts that a block will be referenced in the near future. Re-reference Interval Prediction (RRIP) categorizes blocks as near re-reference, distant re-reference and long re-reference interval blocks [8]. On a miss the block that is predicted to be referenced most far in the future is replaced. Set-dueling is used where one policy inserts blocks with distant re-reference prediction and other one inserts majority of blocks with distant re-reference prediction and infrequently inserts new blocks with a long re-reference interval. RRIP prevents blocks with distant re-reference interval from evicting blocks that have a near re-reference interval. An extension of RRIP to multi-core shared cache is analogous to DIP for shared cache. Each core selects the best re-reference interval (long or distant) using set dueling.

III. A SAMPLING DEAD BLOCK PREDICTOR

In this section we discuss the design of a new sampling-based dead block predictor.

A. A Sampling Partial Tag Array

The sampling predictor keeps a small partial tag array, or *sampler*. Each set in the sampler corresponds to a selected set in the cache; e.g. if the original cache has 2,048 sets, the sampler could keep 32 sets corresponding to every $2,048/32 = 64^{\text{th}}$ cache set. Since correctness of matches is not necessary in the sampler tag array, only the lower-order 15 bits of tags are stored to conserve area and energy. (Nevertheless, we observed no incorrect matches in any of the benchmarks; 15-bit tags are quite sufficient for this application.) As with other dead block predictors, each access to the LLC incurs an access to the predictor. However, predictor is only updated when there is an access or replacement in a cache set with a corresponding

sampler set. This strategy works because the learning acquired through sampling a few sets generalizes to the entire cache.

We find that predictor accuracy improves slightly as more sets are added to the sampler, although too many sets can increase destructive interference in the prediction tables. We would like to have as few sets as possible to save power. We find that 32 sets provide a good trade-off between accuracy and efficiency.

B. Advantages of a Sampler

A sampler decouples the prediction mechanism from the structure of the cache, offering several advantages over previous predictors:

- 1) Since the predictor and sampler are only updated on a small fraction of cache accesses and replacements, the power requirement of the predictor is reduced.
- 2) The replacement policy of the sampler does not have to match that of the cache. For instance, the LLC may use the less costly random replacement policy, but the sampler can still use the deterministic LRU policy. A deterministic policy is easier to learn from because the same sequence of references comes up consistently, uninterrupted by random evictions.
- 3) The associativity of the sampler does not have to match that of the original cache. We have found that, with a 16-way LLC, a 12-way sampler offers better prediction accuracy than a 16-way sampler since blocks that are likely to be dead are evicted sooner. Also, a 12-way sampler consumes less storage and power.

C. Advantage Over Previous Predictors

The retrace predictor keeps a separate signature for every cache block to be used by the predictor when that block is accessed. Counting predictors also store counts with each block. These predictors, as well as cache bursts, have two problems: 1) Every cache block must be associated with a significant amount of metadata. Retrace requires keeping the signature for each block, while counting predictors require keeping a count and other data for each block. 2) Every update to a block incurs a read/modify/write cycle for the metadata. Either a count or a signature must be read, modified, and written. Meeting the timing constraints of this sequence of operations could be problematic, especially in a low-power design.

However, the new predictor uses a trace based only on the PC. That is, rather than predicting whether a block is dead based on the trace of instructions that refer to that block, the predictor only uses the PC of the last instruction that accessed a block. Thus, all predictor state can be kept in the predictor, and only a single additional bit of metadata is needed for each cache block – a bit that indicates whether the block has been predicted as dead. Trace metadata is still stored in the sampler, but since the sample tag array is far smaller than the actual LLC tag array, area and timing are not a problem. The predictor state is only modified on the small fraction of accesses to sampler sets.

Figure 2 gives a block diagram of the retrace and sampling predictors, showing the times during which they are accessed and updated. The sampling predictor is accessed as frequently as the retrace predictor, but it is updated far less often.

D. Key Difference with PC-Only Sampler

The retrace predictor could also use just the PC to index its prediction table. Each time a block is accessed, the signature could immediately be used to update the predictor indicating that the block is live. However, retrace would still need to keep a signature for each cache block for the time between the last access to the block and the eviction of the block to update the predictor on an eviction. In addition to the storage requirement, retrace would also need a 16-bit channel between the stream of instructions and the LLC.

The sampling predictor does not have this problem. It only needs to keep the PC signature for tags tracked in the sampler. There are far fewer of these sampler signatures, i.e. 1,536 for a 12-way 32-set sampler, compared with the number of signatures for the entire cache with the retrace predictor, i.e. 32,768. The sampler needs only a one-bit channel to the LLC to provide predictions.

E. A Skewed Organization

The sampling predictor uses the idea of a skewed organization [22], [16] to reduce the impact of conflicts in the table. The predictor keeps three 4,096-entry tables of 2-bit counters, each indexed by a different hash of a 15-bit signature. Each access to the predictor yields three counter values whose sum is used as a confidence compared with a threshold; if the threshold is met, then the corresponding block is predicted dead. This organization offers an improvement over the previous dead block predictors because unrelated signatures might conflict in one table but are less likely to conflict in three tables, so the effect of destructive conflicts is reduced. A happy consequence of using the skewed predictor is more sensitive confidence estimation: with three tables, we have nine confidence levels to choose from instead of just four with a single table. We find that a threshold of eight gives the best accuracy. Figure 3 shows the difference in the design of the retrace and skewed predictors.

F. Multiple Cores

The sampling predictor described in this paper is used unmodified for both single-thread and multi-core workloads. The same 32-set sampling predictor is used for the 2MB single-core cache as well as the 8MB quad-core cache. There is no special tuning for multi-core workloads.

IV. A COMPARISON OF PREDICTOR STORAGE AND POWER

In this section, we discuss the storage requirements of dead block predictors: the retrace predictor, the counting predictor, the sampling predictor.

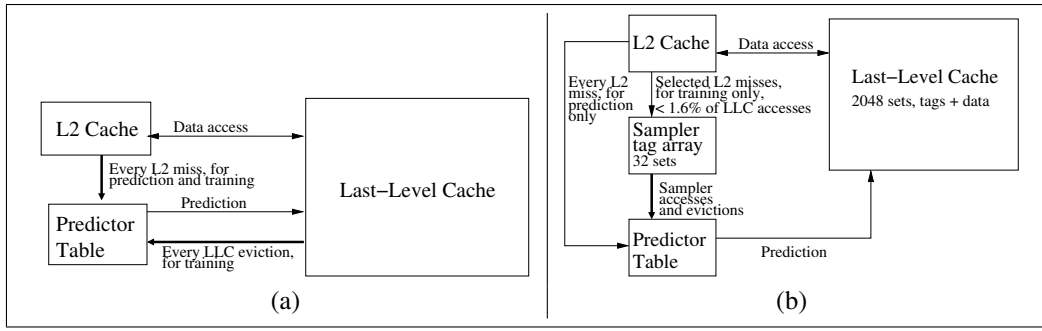


Fig. 2: Refrtrace dead block predictor (a), and new dead block predictor with sampler tag array (b). The sampler and dead block predictor table are updated for 1.6% of the accesses to the LLC, while the refrtrace predictor is updated on every access.

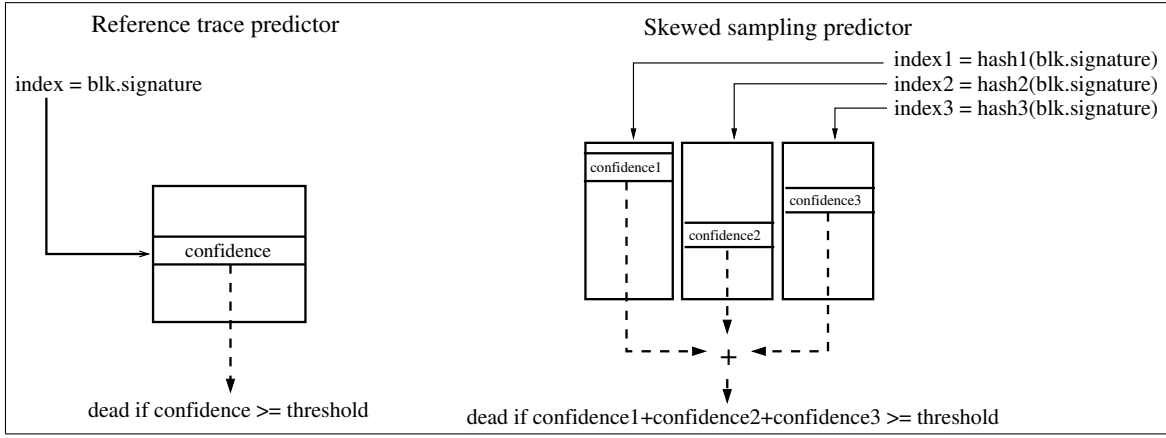


Fig. 3: Block diagrams for dead block predictors

A. Reference Trace Predictor

For this study, we use a refrtrace predictor indexed using 15-bit signature. Thus, the prediction table contains 2^{15} two-bit counters, or 8KB. We find diminishing returns in terms of accuracy from larger tables, so we limit our refrtrace predictor to this size. Each cache block is associated with two extra fields: the 15-bit signature arising from the most recent sequence of accesses to that block, and another bit that indicates whether the block has been predicted as dead. With a 2MB cache with 64B blocks, this works out to 64KB of extra metadata in the cache. Thus, the total amount of state for the refrtrace predictor is 72KB, or 3.5% of the data capacity of the LLC.

B. Counting Predictor

The counting-based Live-time Predictor (LvP) predictor [11] is a 256×256 table of entries, each of which includes the following fields: a four-bit counter keeping track of the number of accesses to a block and a one-bit confidence counter. Thus, the predictor uses 40KB of storage. In addition, each cache block is augmented with the following metadata: an eight-bit hashed PC, a four-bit counter keeping track of the number of times the cache block is accessed, a four-bit number

of accesses to the block from the last time the block was in the cache, and a one-bit confidence counter. This works out to approximately 68KB of extra metadata in the cache. Thus, the total amount of state for the counting predictor is 108KB, or 5.3% of the data capacity of the LLC.

C. Sampling Predictor

The three prediction tables for the skewed dead block predictor are each 4,096 two-bit counters so they consume 3KB of storage.

We model a sampler with 32 sets. Each set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, one prediction bit, one valid bit, and four bits to maintain LRU position information, consuming 6.75KB of storage¹. Each cache line also holds one extra bit of metadata. Thus, the sampling predictor consumes 13.75KB of storage, which is less than 1% of the capacity of a 2MB LLC. Table I summarizes the storage requirements of each predictor.

¹For an LRU cache, why do we not simply use the tags already in the cache? Since the sampler is 12-way associative and does not experience bypass, there will not always be a correspondence between tags in a sampler set and the same set in the cache. Also, we would like to access and update the sampler without waiting for the tag array latency.

Predictor	Predictor Structures	Cache Metadata	Total Storage
refrace	8KB table	16 bits \times 32K blocks = 64KB	72KB
counting	256 \times 256 table, 5-bit entries = 40KB	17 bits \times 32K blocks = 68KB	108KB
sampler	3 \times 1KB tables + 6.75KB sampler = 9.75	1 bit \times 32K blocks = 4KB	13.75KB

TABLE I: Storage overhead for the various predictors

D. Predictor Power

The sampling predictor uses far less storage than the other predictors, but part of its design includes sets of associative tags. Thus, it behooves us to account for the potential impact of this structure on power. Table II shows the results of CACTI 5.3 simulations [24] to determine the leakage and dynamic power of the various components of each predictor. The sampler was modeled as the tag array of a cache with as many sets as the sampler, with only the tag power being reported. The predictor tables for the pattern sample predictors was modeled as a tagless RAM with three banks accessed simultaneously, while the predictor table for the retrace predictor was modeled as a single bank 8KB tagless RAM. The prediction table for the counting predictor was conservatively modeled as a 32KB tagless RAM. To attribute extra power to cache metadata, we modeled the 2MB LLC both with and without the extra metadata, represented as extra bits in the data array, and report the difference between the two.

1) *Dynamic Power*: When it is being accessed, the dynamic power of the sampling predictor is 57% of the dynamic power of the retrace predictor, and only 28% of the dynamic power of the counting predictor. The baseline LLC itself has a dynamic power of 2.75W. Thus, the sampling predictor consumes 3.1% of the power budget of the baseline cache, while the counting predictor consumes 11% of it. Note that CACTI reports peak dynamic power. Since the sampler is accessed only on a small fraction of LLC accesses, the actual dynamic power for the sampling predictor would far lower.

2) *Leakage Power*: For many programs, dynamic power of the predictors will not be an important issue since the LLC might be accessed infrequently compared with other structures. However, leakage power is always a concern. The sampling predictor has a leakage power that is only 40% of the retrace predictor, and only 25% of the counting predictor. This is primarily due to the reduction in cache metadata required by the predictor. As a percentage of the 0.512W total leakage power of the LLC, the sampling predictor uses only 1.2%, while the counting predictor uses 4.7% and the retrace predictor uses 2.9%.

E. Latency

CACTI simulations show that the latency reading and writing the structures related to the sampling predictor fit well within the timing constraints of the LLC. It is particularly fast compared with the retrace predictor since it does not need to read/modify/write metadata in the LLC cache on every access.

V. A DEAD-BLOCK DRIVEN REPLACEMENT AND BYPASS POLICY

We evaluate dead block predictors in the context of a combined dead block replacement and bypassing optimization [11]. When it is time to choose a victim block, a predicted dead block may be chosen instead of a random or LRU block. If there is no predicted dead block, a random or LRU block may be evicted.

If a block to be placed in a set will be used further in the future than any block currently in the set, then it makes sense to decline placing it [17]. That is, the block should bypass the cache. Bypassing can reduce misses in LLCs, especially for programs where most of the temporal locality is captured by the first-level cache. Dead block predictors can be used to implement bypassing: if a block is predicted dead on its first access then it is not placed in the cache.

A. Dead Block Replacement and Bypassing with Default Random Replacement

In Section VII, we show results for dead-block replacement and bypass using a default random replacement policy. We show that this scheme has very low overhead and significant performance and power advantages.

What does it mean for a block to be dead in a randomly replaced cache? The concept of a dead block is well-defined even for a randomly-replaced cache: a block is dead if it will be evicted before it is used again. However, predicting whether a block is dead is now a matter of predicting the outcome of a random event. The goal is not necessarily to identify with 100% certainty which block will be evicted before it is used next, but to identify a block that has a high probability of not being used again soon.

B. Predictor Update in the Optimization

One question naturally arises: should the predictor learn from evictions that it caused? We find that for retrace and for the sampling predictor, allowing the predictor to learn from its own evictions results in slightly improved average miss rates and performance over not doing so. We believe that this feedback allows the predictor to more quickly generalize patterns learned for some sets to other sets. On the other hand, we find no benefit from letting a tag “bypass” the sampler, i.e., tags from all accesses to sampled sets are placed into the sampler.

VI. EXPERIMENTAL METHODOLOGY

This section outlines the experimental methodology used in this study.

Predictor	Prediction Structures Power		Extra Metadata Power		Total Power	
	leakage	dynamic	leakage	dynamic	leakage	dynamic
retrace	0.002	0.030	0.013	0.120	0.015	0.150
counting	0.010	0.175	0.014	0.127	0.024	0.302
sampler	0.005	0.078	0.001	0.008	0.006	0.086

TABLE II: Dynamic and leakage power for predictor components. All figures are in Watts.

Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD	Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD
astar	2.275	2.062	1.829	185B	bwaves	0.088	0.088	3.918	680B
bzip2	0.836	0.589	2.713	368B	cactusADM	13.529	13.348	1.088	81B
calculix	0.006	0.006	3.976	4433B	dealII	0.031	0.031	3.844	1387B
gamess	0.005	0.005	3.888	48B	gcc	0.640	0.524	2.879	64B
GemsFDTD	13.208	10.846	0.818	1060B	gobmk	0.121	0.121	3.017	133B
gromacs	0.357	0.336	3.061	1B	h264ref	0.060	0.060	3.699	8B
hmmer	1.032	0.609	3.017	942B	lbm	25.189	20.803	0.891	13B
leslie3d	7.231	5.898	0.931	176B	libquantum	23.729	22.64	0.558	2666B
mcf	56.755	45.061	0.298	370B	milc	15.624	15.392	0.696	272B
namd	0.047	0.047	3.809	1527B	omnetpp	13.594	10.470	0.577	477B
perlbench	0.789	0.628	2.175	541B	povray	0.004	0.004	2.908	160B
sjeng	0.318	0.317	3.156	477B	soplex	25.242	16.848	0.559	382B
sphinx3	11.586	8.519	0.655	3195B	tonto	0.046	0.046	3.472	44B
wrf	5.040	4.434	0.934	2694B	xalancbmk	18.288	10.885	0.311	178B
zeusmp	4.567	3.956	1.230	405B					

TABLE III: The 29 SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). Benchmarks in the subset in **boldface** .

A. Simulation Environment

The simulator is a modified version of CMP\$im, a memory-system simulator that is accurate to within 4% of a detailed cycle-accurate simulator [6]. The version we used was provided with the JILP Cache Replacement Championship [2]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction and dead block predictor accuracy. The experiments model a 16-way set-associative last-level cache to remain consistent with other previous work [12], [15], [19], [20]. The microarchitectural parameters closely model Intel Core i7 (Nehalem) with the following parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way L3: 2MB/core. Each benchmark is compiled for the x86_64 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C, C++, and FORTRAN.

We use SPEC CPU 2006 benchmarks. We use SimPoint [18] to identify a single one billion instruction characteristic interval (i.e. *simpoint*) of each benchmark. Each benchmark is run with the first `ref` input provided by the `runspec` command.

1) *Single-Thread Workloads*: For single-core experiments, the infrastructure simulates one billion instructions. We simu-

late a 2MB LLC for the single-thread workloads. In keeping with the methodology of recent cache papers [12], [13], [20], [19], [15], [9], [11], [7], [8], we choose a memory-intensive subset of the benchmarks. We use the following criterion: a benchmark is included in the subset if the number of misses in the LLC decreases by at least 1% when using the optimal replacement and bypass policy instead of LRU².

Table III shows each benchmark with the baseline LLC misses per 1000 instructions (MPKI), optimal MPKI, baseline instructions-per-cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint.

2) *Multi-Core Workloads*: Table IV shows ten mixes of SPEC CPU 2006 simpoints chosen four at a time with a variety of memory behaviors characterized in the table by cache sensitivity curves. We use these mixes for quad-core simulations. Each benchmark runs simultaneously with the others, restarting after one billion instructions, until all of the benchmarks have executed at least one billion instructions. We simulate an 8MB shared LLC for the multi-core workloads.

²Ten of the 29 SPEC CPU 2006 benchmarks experience no significant reduction in misses even with optimal replacement. Our technique causes no change in performance in these benchmarks.











Name	Benchmarks	Cache Sensitivity Curve
mix1	mcf hmmer libquantum omnetpp	
mix2	gobmk soplex libquantum lbm	
mix3	zeusmp leslie3d libquantum xalancbmk	
mix4	gamess cactusADM soplex libquantum	
mix5	bzip2 gamess mcf sphinx3	
mix6	gcc calculix libquantum sphinx3	
mix7	perlbench milc hmmer lbm	
mix8	bzip2 gcc gobmk lbm	
mix9	gamess mcf tonto xalancbmk	
mix10	milc namd sphinx3 xalancbmk	

TABLE IV: Multi-core workload mixes with cache sensitivity curves giving LLC misses per 1000 instructions (MPKI) on the y -axis for last-level cache sizes 128KB through 32MB on the x -axis.

For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 8MB cache, we compute IPC_i . Then we find $SingleIPC_i$ as the IPC of the same program running in isolation with an 8MB cache with LRU replacement. Then we compute the weighted IPC as $\sum IPC_i / SingleIPC_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

B. Optimal Replacement and Bypass Policy

For simulating misses, we also compare with an optimal block replacement and bypass policy. That is, we enhance Belady’s MIN replacement policy [3] with a bypass policy that refuses to place a block in a set when that block’s next access will not occur until after the next accesses to all other blocks in the set. We use trace-based simulation to determine the optimal number of misses using the same sequence of memory accesses made by the out-of-order simulator. The out-of-order simulator does not include the optimal replacement and bypass policy so we report optimal numbers only for cache miss reduction and not for speedup.

VII. EXPERIMENTAL RESULTS

In this section we discuss results of our experiments. In the graphs that follow, several techniques are referred to with abbreviated names. Table V gives a legend for these names.

For TDBP and CDBP, we simulate a dead block bypass and replacement policy just as described previously, dropping in the retrace and counting predictors, respectively, in place of our sampling predictor.

A. Dead Block Replacement with LRU Baseline

We explore the use of sampling prediction to drive replacement and bypass in a default LRU replaced cache compared with several other techniques for the single-thread benchmarks.

1) *LLC Misses*: Figure 4 shows LLC cache misses normalized to a 2MB LRU cache for each benchmark. On average, dynamic insertion (DIP) reduces cache misses to 93.9% of the baseline LRU, a reduction of by 6.1%. RRIP reduces misses by 8.1%. The retrace-predictor-driven policy (TDBP) *increases* average misses on average by 8.0% (mostly due to 473.astar), decreasing misses on only 11 of the 19 benchmarks. CDBP reduces average misses by 4.6%. The sampling predictor reduces average misses by 11.7%. The optimal policy reduces misses by 18.6% over LRU; thus, the sampling predictor achieves 63% of the improvement of the optimal policy.

2) *Speedup*: Reducing cache misses translates into improved performance. Figure 5 shows the speedup (i.e. new IPC divided by old IPC) over LRU for the predictor-driven policies with a default LRU cache.

DIP improves performance by a geometric mean of 3.1%. TDBP provides a speedup on some benchmarks and a slowdown on others, resulting in a geometric mean speedup of approximately 0%. The counting predictor delivers a geometric mean speedup of 2.3%, and does not significantly slow down any benchmarks. RRIP yields an average speedup of 4.1%. The sampling predictor gives a geometric mean speedup of 5.9%. It improves performance by at least 4% for eight of the benchmarks, as opposed to only five benchmarks for RRIP and CDBP and two for TDBP. The sampling predictor delivers performance superior to each of the other techniques tested.

Speedup and cache misses are particularly poor for 473.astar. As we will see in Section VII-C, dead block prediction accuracy is bad for this benchmark. However, the sampling predictor minimizes the damage by making fewer predictions than the other predictors.

3) *Poor Performance for Trace-Based Predictor*: Note that the retrace predictor performs quite poorly compared with its observed behavior in previous work [15]. In that work, retrace was used for L1 or L2 caches with significant temporal locality in streams of reference reaching the predictor. Retrace learns from these streams of temporal locality. In this work, the predictor optimizes the LLC in which most temporal locality has been filtered by the 256KB middle-level cache. In this situation, it is easier for the predictor to try to simply learn the last PC to reference a block rather than a sparse reference

Name	Technique
Sampler	Dead block bypass and replacement with sampling predictor, default LRU policy
TDBP	Dead block bypass and replacement with retrace, default LRU policy
CDBP	Dead block bypass and replacement with counting predictor, default LRU policy
DIP	Dynamic Insertion Policy, default LRU policy.
RRIP	Re-reference interval prediction
TADIP	Thread-aware DIP, default LRU policy
Random Sampler	Dead block bypass and replacement with sampling predictor, default random policy
Random CDBP	Dead block bypass and replacement with counting predictor, default random policy.
Optimal	Optimal replacement and bypass policy as described in Section VI-B.

TABLE V: Legend for various cache optimization techniques.

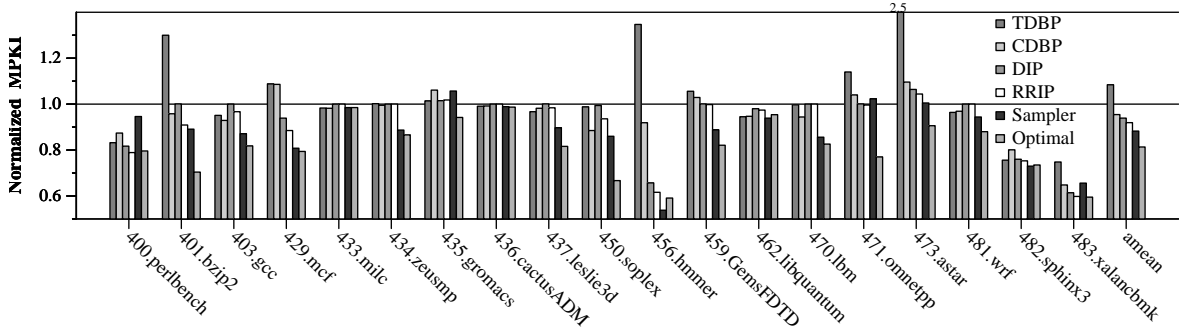


Fig. 4: Reduction in LLC misses for various policies.

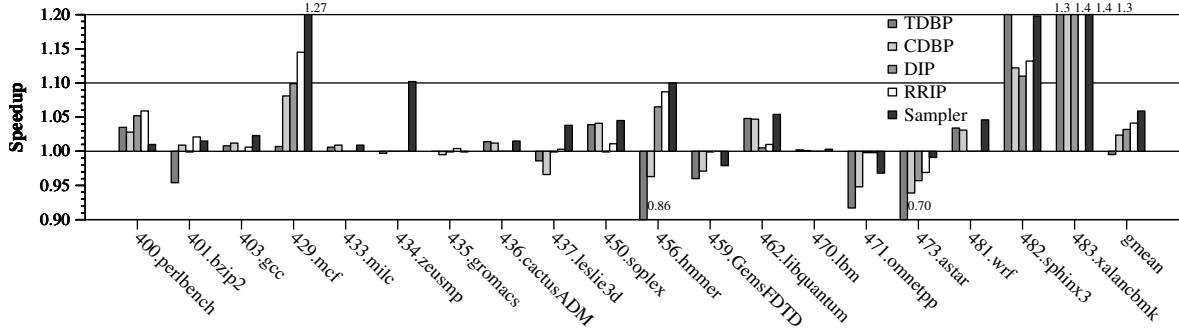


Fig. 5: Speedup for various policies

trace that might not be repeated often enough to learn from. Note that we have simulated retrace correctly with access to the original source code used for the cache bursts paper. In our simulations and in previous work, retrace works quite well when there is no middle-level cache to filter the temporal locality between the small L1 and large LLC. However, many real systems have middle-level caches.

4) *Contribution of Components*: Aside from using only the last PC, there are three other components that contribute to our predictor’s performance with dead block replacement and bypass (DBRB): 1) using a sampler, 2) using reduced associativity in the sampler, and 3) using a skewed predictor. Figure 6 shows the speedup achieved on the single-thread benchmarks for every feasible combination of presence or absence of these components. We find that these three components interact synergistically to improve performance.

The PC-only predictor (“DBRB alone”) without any of the other enhancements achieves a speedup of 3.4% over the LRU

baseline. This predictor is equivalent to the retrace predictor using the last PC instead of the trace signature. Adding a skewed predictor with three tables (“DBRB+3 tables”), each one-fourth the size of the single-table predictor, results in a *reduced* speedup of 2.3%. The advantage of a skewed predictor is its ability to improve accuracy in the presence of a moderate amount of conflict. However, with no sampler to filter the onslaught of a large working set of PCs, the skewed predictor experiences significant conflict with a commensurate reduction in coverage and accuracy.

The sampler with no other enhancements (“DBRB+sampler”) yields a speedup of 3.8%. The improvement over DBRB-only is due to the filtering effect on the predictor: learning from far fewer examples is sufficient to learn the general behavior of the program, but results in much less conflict in the prediction table. Adding the skewed predictor to this scenario (“DBRB+sampler+3 tables”) slightly improves speedup to 4.0%, addressing the

remaining moderate conflict in the predictor.

Reducing the associativity in the sampler from 16 to 12 gives a significant bump in performance. With no skewed predictor, the speedup (“DBRB+sampler+12-way”) improves to 5.6%. Reducing the associativity in the predictor allows the sampler to learn more quickly as tags spend less time in sampler sets. Putting together all three components results in the speedup of 5.9%.

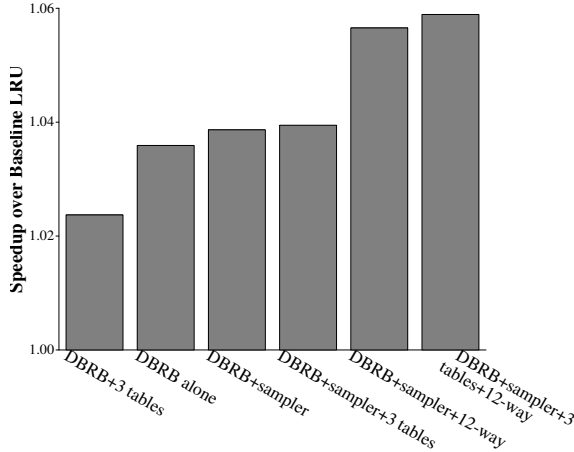


Fig. 6: Contribution to speedup of sampling, reduced associativity, and skewed prediction

B. Dead Block Replacement with Random Baseline

In this subsection we explore the use of the sampling predictor to do dead-block replacement and bypass with a baseline randomly-replaced cache. When a block is replaced, a predicted dead block is chosen, or if there is none, a random block is chosen. We compare with the CDBP and random replacement. We do not compare with TDBP, RRIP or DIP because these policies depend on a baseline LRU replacement policy and become meaningless in a randomly-replaced cache.

1) *LLC Misses*: Figure 7 shows the normalized number of cache misses for the various policies with a default random-replacement policy. The figures are normalized to the same baseline LRU cache from the previous graphs, so the numbers are directly comparable. CDBP reduces misses for some benchmarks but increases misses for others, resulting in no net benefit. Random replacement by itself increases misses an average of 2.5% over the LRU baseline. The sampling predictor yields an average normalized MPKI of 0.925, an improvement of 7.5% over the LRU baseline. Note that the sampling predictor with a default random-replacement policy requires only one bit of metadata associated with individual cache lines. Amortizing the cost of the predictor storage over the cache lines (computed in Section IV), the replacement and bypass policy requires only 1.71 bits per cache line to deliver 7.5% fewer misses than the LRU policy.

2) *Speedup*: Figure 8 shows the speedup for the predictor-driven policies with a default random cache. CDBP yields an almost negligible speedup of 0.1%. Random replacement by

itself results in a 1.1% slowdown. The sampling predictor gives a speedup of 3.4% over the LRU baseline. Thus, a sampling predictor can be used to improve performance with a default randomly-replaced cache.

C. Prediction Accuracy and Coverage

Mispredictions come in two varieties: false positives and false negatives. False positives are more harmful because they wrongly allow an optimization to use a live block for some other purpose, causing a miss. The coverage of a predictor is ratio of positive predictions to all predictions. If a predictor is consulted on every cache access, then the coverage is the fraction of cache accesses when the optimization may be applied. Higher coverage means more opportunity for the optimization. Figure 9 shows the coverage and false positive rates for the various predictors given a default LRU cache. On average, retrace predicts that a block is dead for 88% of LLC accesses, and is wrong about that prediction for 19.9% of cache accesses. The counting predictor has a coverage of 67% and is wrong 7.19% of the time. The sampling predictor has a coverage of 59% and a low false positive rate of 3.0%, explaining why it has the highest average speedup among the predictors. The benchmark `473.astar` exhibits apparently unpredictable behavior. No predictor has good accuracy for this benchmark. However, the sampling predictor has very low coverage for this benchmark, minimizing the damage caused by potential false positives.

D. Multiple Cores Sharing a Last-Level Cache

Figure 10(a) shows the normalized weighted speedup achieved by the various techniques on the multi-core workloads with an 8MB last-level cache and a default LRU policy. The normalized weighted speedup over all 10 workloads ranges from 6.4% to 24.2% for the sampler, with a geometric mean of 12.5%, compared with 10% for CDBP, 7.6% for TADIP, 5.6% for TDBP, and 4.5% for the multi-core version of RRIP.

Figure 10(b) shows the normalized weighted speedup for techniques with a default random policy. The speedups are still normalized to a default 8MB LRU cache. The random sampler achieves a geometric mean normalized weighted speedup of 7%, compared with 6% for random CDBP and no benefit for random replacement by itself. All of the other techniques discussed rely on the presence of a default LRU policy and do not make sense in the context of a default random replacement policy.

The average normalized MPKIs (not graphed for lack of space) are 0.77 for the sampler, 0.79 for CDBP, 0.85 for TADIP, 0.95 for TDBP, 0.82 for the random sampler, 0.93 for multi-core RRIP, and 0.84 for random CDBP. The sampler reduces misses by 23% on average.

VIII. CONCLUSION AND FUTURE WORK

Sampling prediction can improve performance for last-level caches while reducing the power requirements over previous techniques. For future work, we plan to investigate sampling

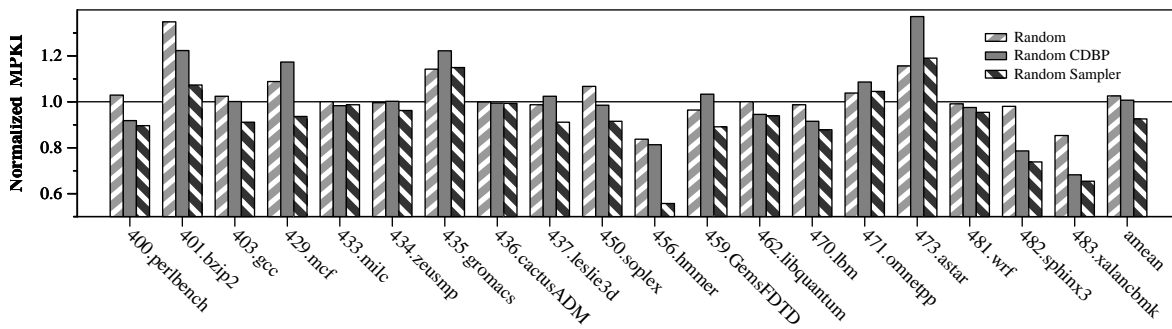


Fig. 7: LLC misses per kilo-instruction for various policies

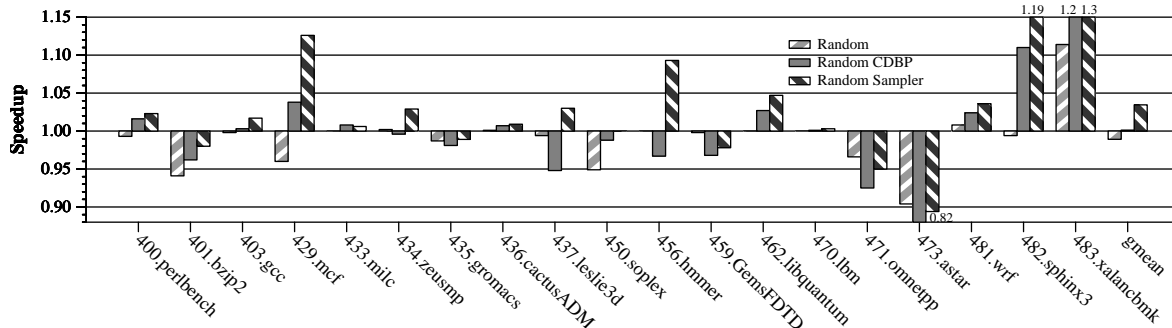


Fig. 8: Speedup for various replacement policies with a default random cache

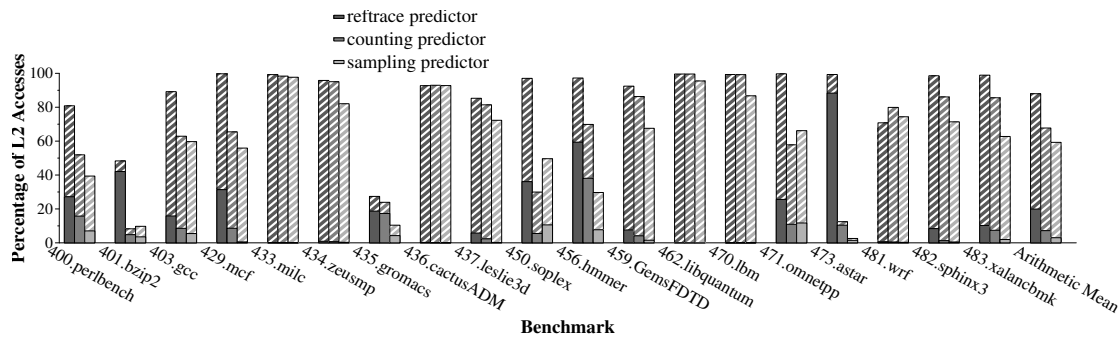


Fig. 9: Coverage and false positive rates for the various predictors

techniques for counting predictors as well as cache-bursts predictors [15] at all levels of the memory hierarchy. We plan to evaluate multi-threaded workloads with significant sharing of data to see what improvements can be made to the sampling predictor in this context. The skewed organization boosts the accuracy of the sampling predictor. In future work we plan to apply other techniques derived from branch prediction to dead block prediction. We plan to investigate the use of sampling predictors for optimizations other than replacement and bypass.

IX. ACKNOWLEDGEMENTS

Daniel A. Jiménez, Samira M. Khan, and Yingying Tian are supported by grants from the National Science Foundation: CCF-0931874 and CRI-0751138 as well as a grant from the Norman Hackerman Advanced Research Program, NHARP-010115-0079-2009. We thank Doug Burger for his helpful

feedback on an earlier version of this work. We thank the anonymous reviewers for their invaluable feedback.

REFERENCES

- [1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O’Boyle. IATAC: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, 2005.
- [2] Alaa R. Alameldeen, Aamer Jaleel, Moinuddin Qureshi, and Joel Emer. 1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship. <http://www.jilp.org/jwac-1/>.
- [3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [4] D. Burger, J. R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general-purpose microprocessors. *Technical Report 1261*, 1995.
- [5] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002.

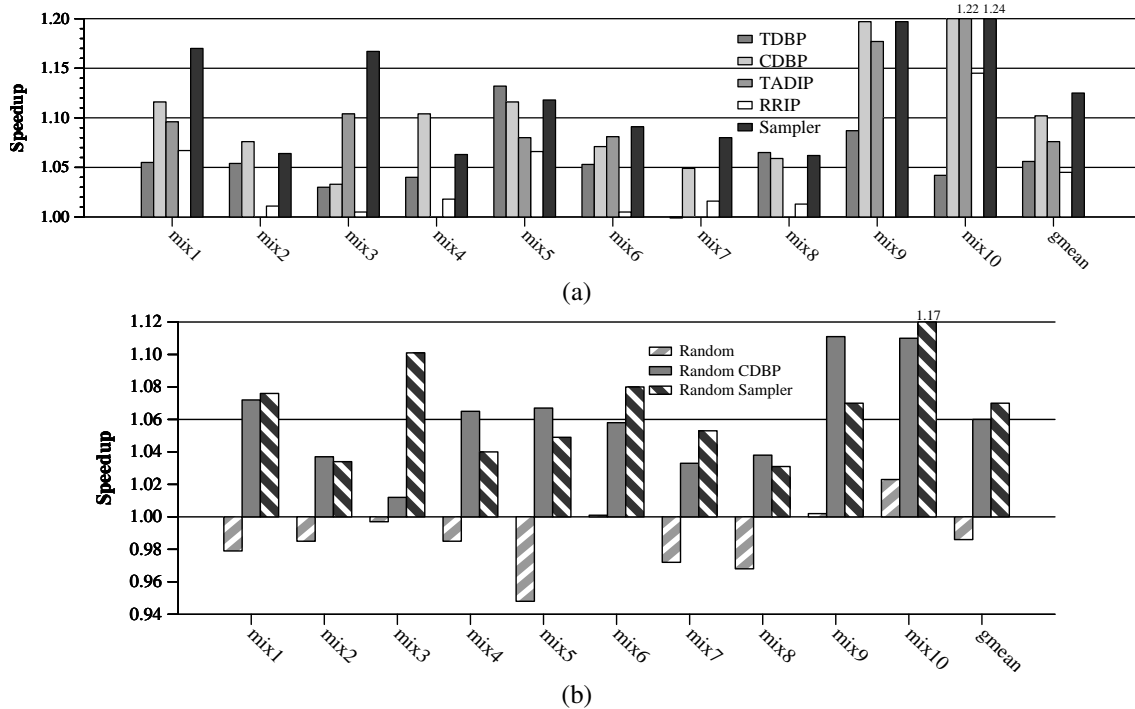


Fig. 10: Weighted speedup for multi-core workloads normalized to LRU for (a) a default LRU cache and (b) a default randomly replaced cache.

- [6] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. CMP\$im: A pin-based on-the-fly single/multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2008)*, June 2008.
- [7] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon Stealy Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 2008 International Conference on Parallel Architectures and Compiler Techniques (PACT)*, September 2008.
- [8] Aamer Jaleel, Kevin Theobald, Simon Stealy Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*, June 2010.
- [9] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, pages 245–250, 2007.
- [10] Samira M. Khan, Daniel A. Jiménez, Babak Falsafi, and Doug Burger. Using dead blocks as a virtual victim cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technologies (PACT)*, 2010.
- [11] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.
- [12] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *International Symposium on Computer Architecture*, pages 139 – 148, 2000.
- [13] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.
- [14] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 23(2):48–59, 1995.
- [15] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [16] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.
- [17] David M. Nicol, Albert G. Greenberg, and Boris D. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. In *IEEE Transactions on Parallel and Distributed Systems*, volume vol. 5, pages 849–859, August 1994.
- [18] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.
- [19] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Stealy Jr., and Joel S. Emer. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA 2007)*, June 9-13, 2007, San Diego, California, USA. ACM, 2007.
- [20] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.
- [21] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. *Annual Workshop on Interaction between Compilers and Computer Architecture*, 0:46–57, 2005.
- [22] André Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993.
- [23] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 37–45, New York, NY, USA, 2004. ACM.
- [24] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical report, HP Tech Report HPL-2008-20, 2008.
- [25] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 199, Los Alamitos, CA, USA, 2002. IEEE Computer Society.