Exploiting Compressed Block Size as an Indicator of Future Reuse

Gennady Pekhimenko† Tyler Huberty† Rui Cai† Onur Mutlu†
Phillip B. Gibbons⋆ Michael A. Kozuch⋆ Todd C. Mowry†
†Carnegie Mellon University ⋆Intel Labs Pittsburgh

Abstract

We introduce a set of new Compression-Aware Management Policies (CAMP) *for on-chip caches that employ data compression. Our management policies are based on two key ideas. First, we show that it is possible to build a more efficient management policy for compressed caches if the compressed block size is directly used in calculating the* value (importance) *of a block to the cache. This leads to* Minimal-Value Eviction (MVE)*, a policy that evicts the cache blocks with the least value, based on both the size and the expected future reuse. Second, we show that, in some cases, compressed block size can be used as an efficient indicator of the future reuse of a cache block. We use this idea to build a new insertion policy called* Size-based Insertion Policy (SIP) *that dynamically prioritizes cache blocks using their compressed size as an indicator.*

We compare CAMP (and its global variant G-CAMP) to prior on-chip cache management policies (both size-oblivious and size-aware) and find that our mechanisms are more effective in using compressed block size as an extra dimension in cache management decisions. Our results show that the proposed management policies (i) improve performance (by 4.9%/9.0%/10.2% on average in single-/two-/four-core workload evaluations and up to 20.1%), (ii) decrease off-chip bandwidth consumption (by 8.7% in single-core), and (iii) decrease memory subsystem energy consumption (by 7.2% in single-core) for memory intensive workloads compared to the best prior mechanism. CAMP is effective for a variety of compression algorithms and different cache designs with local and global replacement strategies.

1. Introduction

Off-chip main memory latency and bandwidth are major performance bottlenecks in modern systems. Multiple levels of on-chip caches are used to hide the memory latency and reduce off-chip memory bandwidth demand. Efficient utilization of cache space and consequently better performance is dependent upon the ability of the cache replacement policy to identify and retain useful data. Replacement policies, ranging from traditional [7, 12] to state-of-the-art [19, 21, 30, 33], work using a combination of *eviction* (identifies the block to be removed from the cache), *insertion* (manages the initial block priority), and *promotion* (changes the block priority over time) mechanisms. In replacement policies proposed for conventional cache organizations, these mechanisms work by considering only the locality of the cache blocks.

A promising approach to improving effective cache capacity is to use cache compression (e.g., [3, 5, 10, 17, 27, 39]). In compressed caches, data compression algorithms, e.g., Frequent Pattern Compression (FPC) [4], Base-Delta-Immediate Compression (BDI) [27], and Frequent Value Compression [39], are used to achieve higher effective capacity (storing more blocks of data) and to decrease off-chip bandwidth consumption compared to traditional organizations without compression. This compression generates variable-size cache blocks, with larger blocks consuming more cache space than smaller blocks. However, most cache management policies in these compressed cache designs do not use size in cache management decisions [3, 10, 17, 27, 39]. Only one recent work—ECM [5]—uses the block size information, but its effectiveness is limited by its coarse-grained (big vs. small) view of block size. The need to consider size along with temporal locality is well known in the context of web caches [1, 6, 11, 14, 32], but proposed solutions rely on a recency list of all objects in the web cache [1] or consider frequency of object accesses [11] and are usually prohibitively expensive to implement in hardware for use with on-chip caches.

In this paper, we propose a *Compression-Aware Management Policy (CAMP)* that takes into account compressed cache block size along with temporal locality to improve the performance of compressed caches. Compared to prior work (ECM [5]), our policies use a finer-grained accounting for compressed block size and an optimization-based approach for eviction decisions. Moreover, we find that size is not only a measure of the cost of retaining a given block in the cache, as previous works considered [5], but it is sometimes also *an indicator of block reuse*. CAMP contains two key components, Minimal-Value Eviction (MVE) and Size-based Insertion Policy (SIP), which significantly improve the quality of replacement decisions in compressed caches (see Section 6 for a comprehensive comparison with ECM) at a modest hardware cost.

Minimal-Value Eviction (MVE). MVE is based on the observation that one should evict an uncompressed block with good locality to make/retain room for a set of smaller compressed blocks of the same total size, even if individually those blocks have lesser locality, as long as collectively the set provides more hits cumulatively. A special case of this is that when two blocks have similar locality characteristics, it is preferable to evict the larger cache block. MVE measures the *value* of each block as a combination of its locality properties and size. When an eviction is required (to make space for a new block), MVE picks the block with the least value as the victim.

Size-based Insertion Policy (SIP). SIP is based on our new observation that the compressed size of a cache block can sometimes be used as an indicator of its reuse characteristics. This is because elements belonging to the same data structure and having the same access characteristics are sometimes (but not always) compressed to the same size—e.g., in *bzip2* [35], a compressed block of 34 bytes (with BDI compression [27]) likely belongs to one particular array with narrow values (e.g., small values stored in large data types) as we show in Section 2.3—and these structures more often than not have a specific pattern of access and/or reuse distance.

By dynamically inserting blocks of different sizes with either *high priority*—e.g., in the most-recently-used position for the LRU policy (ensuring blocks stay in cache longer)—or *low priority*—e.g., in the least-recently-used position for the LRU policy (ensuring blocks get evicted quickly unless reused shortly)—SIP learns the reuse characteristics associated with various compressed block sizes and, if such an association

exists, uses this information to maximize the hit ratio.

As demonstrated later in this paper, CAMP (a combination of MVE and SIP) works with both traditional compressed cache designs and compressed caches having decoupled tag and data stores (e.g., V-Way Cache [31] and Indirect Index Cache [16, 17]). It is general enough to be used with different compression mechanisms and requires only modest hardware changes. Compared to prior work, CAMP provides better performance, more efficient cache utilization, reduced off-chip bandwidth consumption, and an overall reduction in the memory subsystem energy requirements.

This paper makes the following contributions:

- We make the observation that the compressed size of a cache block can be indicative of its reuse. We use this observation to develop a new cache insertion policy for compressed caches, the Size-based Insertion Policy (SIP), which uses the size of a compressed block as one of the metrics to predict its potential future reuse.
- We introduce a new compressed cache replacement policy, Minimal-Value Eviction (MVE), which assigns a value to each cache block based on its size and reuse and replaces the set of blocks with the least value.
- We demonstrate that both policies are generally applicable to different compressed cache designs (both with local and global replacement) and can be used with different compression algorithms (FPC [3] and BDI [27]).
- We qualitatively and quantitatively compare CAMP (SIP + MVE) to the conventional LRU policy and three state-of-the-art cache management policies: two size-oblivious policies (RRIP [19] and a policy used in V-Way [31]) and the recent ECM [5]. We observe that CAMP (and its global variant G-CAMP) can considerably (i) improve performance (by 4.9%/9.0%/10.2% on average in single-/two-/four-core workload evaluations and up to 20.1%), (ii) decrease off-chip bandwidth consumption (by 8.7% in single-core), and (iii) decrease memory subsystem energy consumption (by 7.2% in single-core) on average for memory intensive workloads when compared with the best prior mechanism.

2. Motivating Observations

Cache compression [3, 10, 17, 27, 39] is a powerful mechanism that increases effective cache capacity and decreases off-chip bandwidth consumption. In this section, we show that cache compression adds an additional dimension to cache management policy decisions – *the compressed block size* (or simply *the size*), which plays an important role in building more efficient management policies. We do this in three steps.

2.1. Size Matters

In compressed caches, one should design replacement policies that take into account compressed cache block size along with locality to identify victim blocks, because such policies can outperform existing policies that rely only on locality. In fact, Belady's optimal algorithm [7] that relies only on locality (using perfect knowledge to evict the block that will be accessed furthest in the future) is sub-optimal in the context of compressed caches with variable size cache blocks. Figure 1 demonstrates one possible example of such a scenario. In this figure, it is assumed that cache blocks are one of two sizes: (i) uncompressed 64-byte blocks (blocks X and Y) and (ii) compressed 32-byte blocks (blocks A, B, and C). Initially (see ❶), the 160-byte capacity cache contains four blocks: three compressed blocks (A, B, C) and one uncompressed block (Y). Consider the sequence of memory requests X, A, Y, B, and C (see ❷). In this case, after a request for X, Belady's algorithm (based on locality) will evict blocks B and C (to create 64-bytes of free space) that will be accessed furthest into the future. Over the next four accesses, this results in two misses (B and C) and two hits (A and Y).

In contrast, a size-aware replacement policy can detect that it might be better to retain a set of smaller compressed cache blocks that receive more hits cumulatively than a single large (potentially uncompressed) cache block with better locality. For the access pattern discussed above, a size-aware replacement policy makes the decision to retain B and C and evict Y to make space for X (see ❸). As a result, the cache experiences three hits (A, B, and C) and only one miss (Y) and hence outperforms Belady's optimal algorithm.[1] We conclude that using block size information in a compressed cache can lead to better replacement decisions.

2.2. Size Varies

Figure 2 shows the distribution of compressed cache block sizes[2] for a set of representative workloads given a 2MB cache employing the BDI [27] cache compression algorithm (our results with the FPC [3] compression algorithm show similar trends). Even though the size of a compressed block is determined by the compression algorithm, under both designs, **compressed cache block sizes can vary significantly**, both (i) within a single application (i.e., *intra-application*) such as in *astar, povray*, and *gcc* and (ii) between applications (i.e., *inter-application*) such as between *h264ref* and *wrf*.

Size variation within an application suggests that size-aware replacement policies could be effective for individual single-core workloads. Intra-application variation exists because applications have data that belong to different common compressible patterns (e.g., zeros, repeated values, and narrow values [27]) and as a result end up with a mix of compressed cache block sizes. In the case of multiple cores with shared caches, inter-application variation suggests that even if an application has a single dominant compressed cache block size (e.g., *lbm, h264ref* and *wrf*), running these applications together will result in the shared cache experiencing a mix of compressed cache block sizes. Hence, size-aware management of compressed caches can be even more important for efficient cache utilization in multi-core systems (as we demonstrate quantitatively in Section 6.2).

2.3. Size Can Indicate Reuse

We observe that elements belonging to the same data structure (within an application) sometimes lead to cache blocks that compress to the same size. This observation provides a new opportunity: using the compressed size of a cache block as an indicator of data reuse of the block.

[1] Note that if later (see ❹) there are three additional requests to blocks B, Y, and A (all three hits), the final cache state becomes the same as the initial one. Hence, this example can represent steady state within a loop.

[2] Section 5 describes the details of our evaluation methodology for this and other experiments.

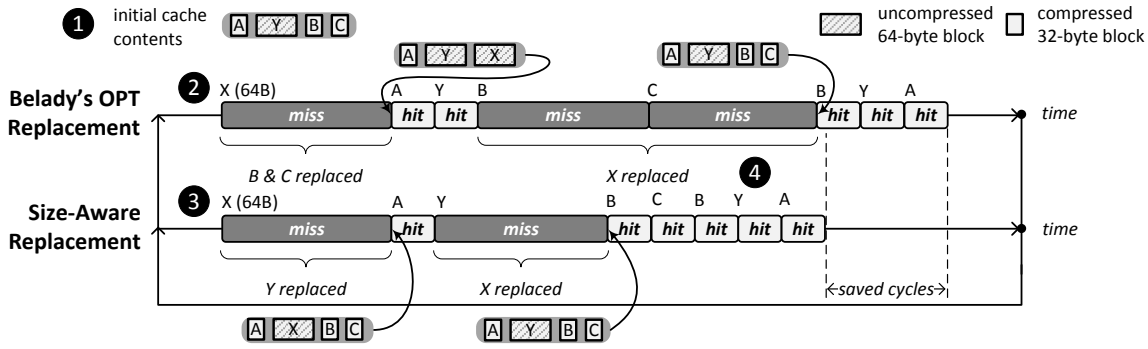


Figure 1: Example demonstrating downside of not including block size information in replacement decisions.

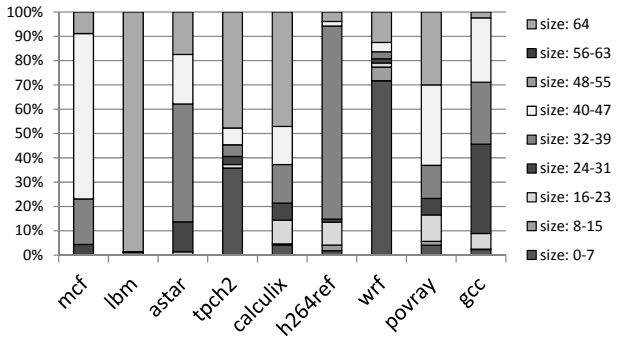


Figure 2: Compressed block size distribution for representative applications with the BDI compression algorithm.

Intuition. We first briefly provide intuition on why there can be a relationship between compressed size and the reuse characteristics of the cache block. As past work has shown, an application's key data structures are typically accessed in a regular fashion, with each data structure having an identifiable access pattern [2]. This regularity in accesses to a data structure can lead to a dominant *reuse distance* [13] range for the cache blocks belonging to the data structure.[3] The same data structure can also have a dominant compressed cache block size, i.e., a majority of the cache blocks containing the data structure can be compressed to one or a few particular sizes. For such a data structure, the compressed cache block size can therefore be a good indicator of the reuse behavior of the cache blocks. In fact, different data structures can have different dominant compressed block sizes and different dominant reuse distances; in such cases, the compressed block size serves as a type of "signature" indicating the reuse pattern of a data structure's cache blocks.

Example to Support the Intuition. To illustrate the connection between compressed size and reuse behavior of data structures intuitively, Figure 3 presents an example loosely based on some of the data structures we observed in *soplex*. There are three data structures in this example: (i) array $A[N]$ of integer indexes that are smaller than value M (well-compressible with BDI [27] to 20-byte cache blocks), (ii) small array $B[16]$ of floating point coefficients (incompressible, 64-byte cache blocks), and (iii) sparse matrix $C[M][N]$ with the main data (very compressible, 1-byte cache blocks). These data structures not only have different compressed block sizes, but also different reuse distances. Accesses to cache blocks for array A occur only once every iteration of the outer loop (long reuse distance). Accesses to cache blocks for array B occur roughly

```
int A[N];       // indices (smaller than M): narrow values
double B[16];   // FP coefficients: incompressible values
double C[M][N];// sparse matrix: many zero values
for (int i=0; i<N; i++) {
   int tmp = A[i];
   for (int j=0; j<N; j++) {
     sum += B[(i+j)%16] * C[tmp][j];
   }
}
```

Figure 3: Code example: size and reuse distance relationship.

every 16^{th} iteration of the inner loop (short reuse distance). Finally, the reuse distance of array C is usually long, although it is data dependent on what indexes are currently stored in array $A[i]$. Hence, this example shows that *compressed block size can indicate the reuse distance of a cache block*: 20-byte blocks (from A) usually have long reuse distance, 64-byte blocks (from B) usually have short reuse distance, and 1-byte blocks (from C) usually have long reuse distance. If a cache learns this relationship, it can prioritize 64-byte blocks over 20-byte and 1-byte blocks in its management policy. As we will show in Section 3.3, our SIP policy learns exactly this kind of relationship, leading to significant performance improvements for several applications (including *soplex*), as shown in Section 6.1.1.[4]

Quantitative Evidence. To verify the relationship between block size and reuse, we have analyzed 23 memory intensive applications' memory access traces (applications described in Section 5). For every cache block within an application, we computed the average distance (measured in memory requests) between the time this block was inserted into the compressed cache and the time when it was reused next. We then accumulate this *reuse distance* information for all different block sizes, where the size of a block is determined with the BDI [27] compression algorithm.

Figures 4 and 5 show the results of this analysis for all (23) memory intensive applications from our workload pool (our methodology is described in Section 5). In 15 of these applications (Figures 4a–4o), compressed block size is an indicator of reuse distance (in other words, it can be used to distinguish blocks with different reuse distances). In eight of the applications (Figures 5a–5h), it is not. Each graph is a scatter plot that shows the reuse distance distribution experienced by various compressed cache block sizes in these applications. There are nine possible compressed block sizes (based on the description from the BDI work [27]). The size of each circle is propor-

[3] Some prior works [18, 20, 28, 38] captured this regularity by learning the relationship between the instruction address and the reuse distance.

[4] Note that our overall policy also accounts for the size of the block, e.g., that a 64-byte block takes up more space in the cache than a 20-byte or 1-byte block, via its MVE policy.

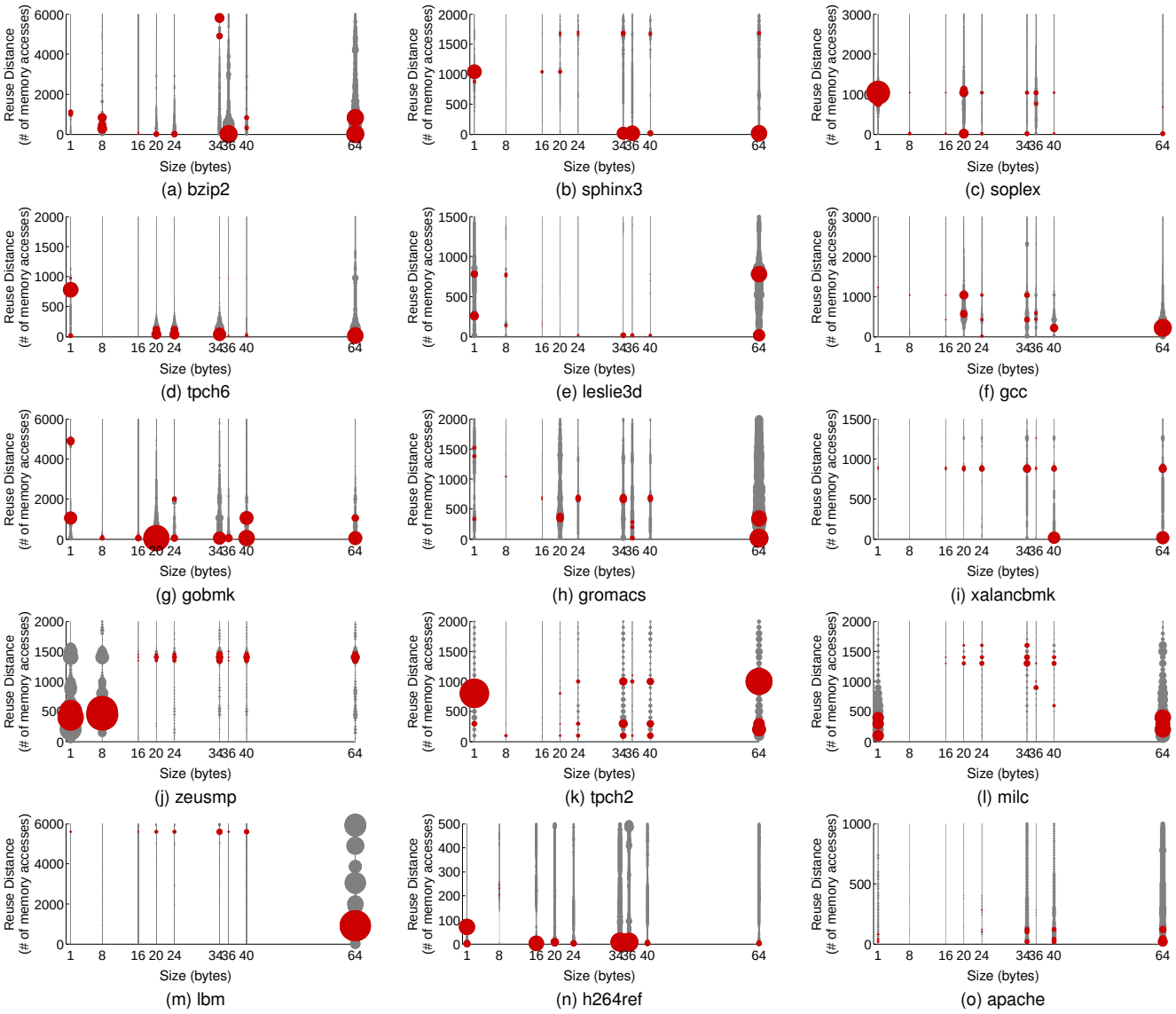


Figure 4: Plots demonstrate the relationship between the compressed block size and reuse distance. Dark red circles correspond to the most frequent reuse distances for every size.

tional to the relative frequency of blocks of a particular size that exhibit a specified reuse distance. The dark red circles indicate the most frequent reuse distances (up to three) for every size.

We make three major observations from these figures. First, there are many applications where block size is an indicator of reuse distance (Figure 4a–4o). For instance, in *bzip2* (Figure 4a), a large number of cache blocks are 8, 36, or 64 (uncompressed) bytes and have a short reuse distance of less than 1000. In contrast, a significant number of blocks are 34 bytes and have a large reuse distance of greater than 5000. This indicates that the 34-byte blocks can be deprioritized by the cache when running *bzip2* to improve performance. Similarly, in *sphinx3*, *tpch6*, and *soplex* (Figures 4b–4d), a significant number of blocks are compressed to 1-byte with a long reuse distance of around 1000, whereas most of the blocks of other sizes have very short reuse distances of less than 100. In general, we observe that data from 15 out of 23 of our evaluated applications show that block size is indicative of reuse. This suggests that a compressed block size can be used as an indicator of future block reuse which in turn can be used to prioritize the blocks of certain sizes (Section 3.3), improving application performance (see the effect on *soplex* in Section 6.1.1).

Second, there are some applications where block size does not have a relationship with reuse distance of the block (e.g., *mcf* and *sjeng*). For example, in *mcf* (Figure 5a), almost all blocks, regardless of their size, have reuse distances around 1500. This means that block size is less effective as an indicator of reuse for such applications (and the mechanism we describe in Section 3.3 will effectively avoid using block size in cache management decisions for such applications).

Third, for applications where block size is indicative of reuse, there is usually not a coarse-grained way to distinguish between block sizes that are indicative of different reuse distances. In other words, simply dividing the blocks into *big* or *small* blocks, as is done in ECM [5], is not enough to identify the different reuse behavior of blocks of different sizes; the distinction between block sizes should be done at a finer granularity. This is evident for *bzip2* (Figure 4a): while 8, 36, and

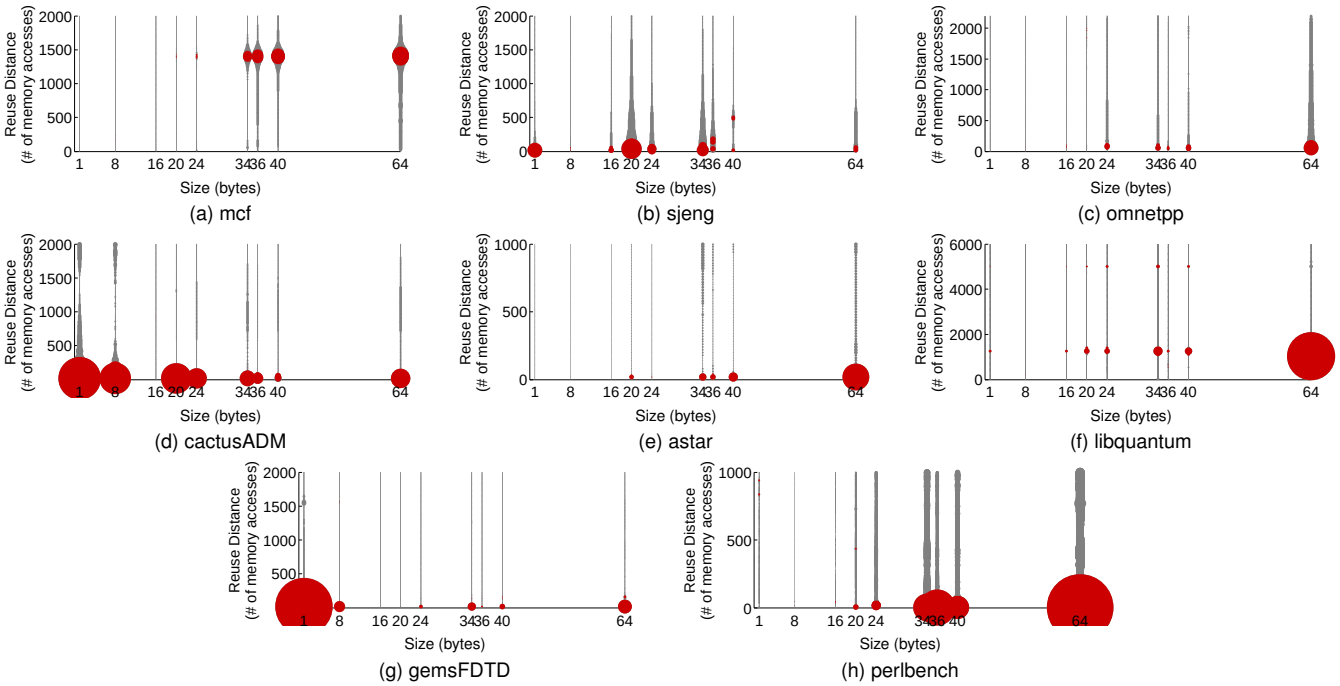


Figure 5: Plots demonstrate the weak relationship between the compressed block size and reuse distance. Dark red circles correspond to the most frequent reuse distances for every size.

64-byte blocks have short reuse distances, a significant fraction of the 34-byte blocks have very long reuse distances (between 5000 and 6000). Hence, there is no single block size threshold that would successfully *divide/distinguish* blocks with high reuse from those with low reuse. Data from other applications (e.g., *soplex*, *leslie3d*, *gcc*) similarly support this.

3. CAMP: Design and Implementation

Our proposed Compression-Aware Management Policy (CAMP) consists of two components: Minimal-Value Eviction (MVE) and Size-based Insertion Policy (SIP). These mechanisms assume a compressed cache structure where the compressed block size is available to the hardware making the insertion and replacement decisions. The tag-store contains double the number of tags and is decoupled from the data-store to allow higher effective capacity (as was proposed in several prior works [3, 10, 27]). We also propose Global CAMP (or G-CAMP), an adaptation of CAMP for a cache with a global replacement policy.

In this section, we first provide the background information needed to understand some of our mechanisms (Section 3.1). Then, we describe the design and implementation of each mechanism in depth (Sections 3.2-3.4). We detail the implementation of our G-CAMP mechanism assuming the structure proposed for the V-Way cache [31]. None of the mechanisms require extensive hardware changes on top of the baseline compressed cache designs (both local and global, see Section 3.5).

3.1. Background

Multiple size-oblivious cache management mechanisms [15, 19, 21, 31] were proposed to improve the performance of conventional on-chip caches (without compression). Among them, we select RRIP [19] as both a comparison point in our evaluations and as a predictor of future re-reference in some of our algorithms (see Section 3.2). This selection is motivated both by the simplicity of the algorithm and its state-of-the-art performance (as shown in [19]).

RRIP. Re-Reference Interval Prediction (RRIP) [19] uses an M-bit saturating counter per cache block as a Re-Reference Prediction Value ($RRPV$) to predict the block's re-reference distance. The key idea behind RRIP is to prioritize the blocks with lower predicted re-reference distance, as these blocks have higher expectation of near-future reuse. Blocks are inserted with a long re-reference interval prediction ($RRPV = 2^M - 2$). On a cache miss, the victim block is a block with a predicted distant re-reference interval ($RRPV = 2^M - 1$). If there is no such block, the $RRPV$ of all blocks is incremented by one and the process repeats until a victim is found. On a cache hit, the $RRPV$ of a block is set to zero (near-immediate re-reference interval). Dynamic RRIP (DRRIP) uses set dueling [30] to select between the aforementioned policy (referred to as SRRIP) and one that inserts blocks with a distant re-reference interval prediction with high probability and inserts blocks with a long re-reference interval prediction with low probability.

V-Way. The Variable-Way, or V-Way [31], cache is a set associative cache with a decoupled tag- and data-store. The goal of V-Way is two-fold: providing a flexible (variable) associativity together with the more efficient global replacement policy. A defining characteristic is that there are more tag-entries than data-entries. Forward and backward pointers are maintained in the tag- and data-store to link the entries. This design enables associativity to effectively vary on a per-set basis by increasing the number of tag-store entries relative to data-store entries. Another benefit is the implementation of a *global replacement policy*, which is able to choose data-victims anywhere in the data-store. This is in contrast to a traditional *local replacement policy*, e.g., [12, 19], which considers only data-store entries within a single set as possible victims. The particular global replacement policy described in V-Way (called Reuse Replacement) consists of a Reuse Counter Table (RCT) with a counter

for each data-store entry. Victim selection is done by starting at a pointer (PTR) to an entry in the RCT and searching for the first counter equal to zero, decrementing each counter while searching, and wrapping around if necessary. A block is inserted with an RCT counter equal to zero and on a hit, the RCT counter for the block is incremented. We use the V-Way design as a foundation for all of our global mechanisms (described in Section 3.4).

3.2. Minimal-Value Eviction (MVE)

The key observation in our MVE policy is that evicting one or more important blocks of larger compressed size may be more beneficial than evicting several more compressible, less important blocks (see Section 2). The idea behind MVE is that each block has a value to the cache. This value is a function of two key parameters: (i) the likelihood of future re-reference and (ii) the compressed block size. For a given <prediction of re-reference, compressed block size> tuple, MVE associates *a value to the block*. Intuitively, a block with higher likelihood of re-reference is more valuable than a block with lower likelihood of re-reference and is assigned a higher value. Similarly, a more compressible block is more valuable than a less compressible block because it takes up fewer segments in the data-store, potentially allowing for the caching of additional useful blocks. The block with the least value in the associativity set is chosen as the next victim for replacement—sometimes multiple blocks need to be evicted to make room for the newly inserted block.

In our implementation of MVE, the value V_i of a cache block i is computed as $V_i = p_i/s_i$, where s_i is the compressed block size of block i and p_i is a predictor of re-reference, such that a larger value of p_i denotes block i is more important and is predicted to be re-referenced sooner in the future. Note that this function matches our intuition and is monotonically increasing with respect to the prediction of re-reference and monotonically decreasing with respect to the size. We consider other functions with these properties (i.e. a weighted linear sum), but find the difference in performance to be negligible.

Our mechanism estimates p_i using RRIP[5] [19] as the predictor of future re-reference due to its simple hardware implementation and state-of-the-art stand-alone performance.[6] As described in Section 3.1, RRIP maintains a re-reference prediction value (RRPV) for each cache block which predicts the re-reference distance. Since a larger RRPV denotes a longer predicted re-reference interval, we compute p_i as $p_i = (RRPV_{MAX} + 1 - RRPV_i)$. Therefore, a block with a predicted short re-reference interval has more value than a comparable block with a predicted long re-reference interval. p_i cannot be zero, because V_i would lose dependence on s_i and become size-oblivious.

Depending on the state of the cache, there are two primary conditions in which a victim block must be selected: (i) the data-store has space for the block to be inserted, but all tags are valid in the tag-directory, or (ii) the data-store does not have space for the block to be inserted (an invalid tag may or may not exist in the tag-directory). In the first case where the data-store is not at capacity, MVE relies solely on the predictor of re-reference or conventional replacement policy, such as RRIP. For the second case, the valid blocks within the set are compared based on V_i and the set of blocks with the least value is evicted to accommodate the block requiring insertion.

[5]Specifically, the version of RRIP that our mechanism uses is SRRIP. We experimented with DRRIP, but found it offered little performance improvement for our mechanisms compared to the additional complexity. All of our evaluations assume an RRPV width $M = 3$.

[6]Other alternatives considered (e.g., [33]) provide only a binary value.

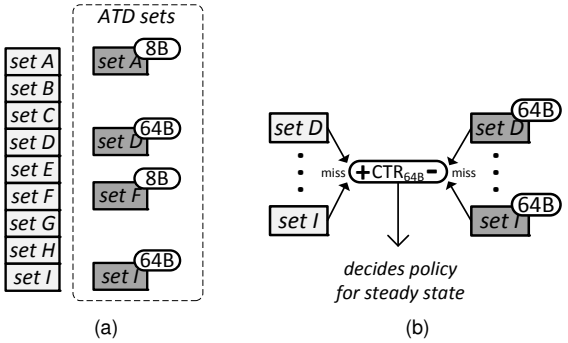


Figure 6: Set selection during training and decision of best insertion policy based on difference in miss rate in MTD/ATD.

MVE likely remains off the critical path, but to simplify the microarchitecture, division in the calculation of V_i is eliminated by bucketing block sizes such that s_i is always a power of two, allowing a simple right-shift operation instead of floating point division. For the purposes of calculating V_i, $s_i = 2$ for blocks of size 0B – 7B, $s_i = 4$ for blocks of size 8B – 15B, $s_i = 8$ for blocks of size 16B – 31B, and so on. The most complex step, comparing blocks by value, can be achieved with a fixed multi-cycle (i.e., a number of cycles less than the cache associativity) parallel comparison.

3.3. Size-based Insertion Policy (SIP)

The key observation behind SIP is that sometimes there is a relation between cache block reuse distance and compressed block size (as shown in Section 2.3). SIP exploits this observation and inserts blocks of certain sizes with higher priority if doing so reduces the cache miss rate. Altering the priority of blocks of certain sizes with short or long reuse distances helps ensure more important blocks stay in the cache.

At run-time, SIP dynamically detects the set of sizes that, when inserted with higher priority, reduce the number of misses relative to a size-oblivious insertion policy. SIP uses a simple mechanism based on dynamic set sampling [30] to make the prioritization decision for various compressed sizes. It selects the best performing policy among competing policies during a periodic training phase and applies that policy during steady state. The observation in dynamic set sampling is that sampling makes it possible to choose the better policy with only a relatively small number of sets selected from the Main Tag Directory (MTD) to have a corresponding set in an Auxiliary Tag Directory (ATD) participating in a tournament. Only the MTD is coupled with the data-store; the ATD is only for deciding which block size(s) should be inserted with high priority. Therefore, there are no performance degradations due to our sampling during training.

Let m be the minimum number of sets that need to be sampled so that dynamic set sampling can determine the best policy with high probability and n be the number of compressible block sizes possible with the compression scheme (e.g., 8B, 16B, 20B, ..., 64B). In SIP, the ATD contains $m \cdot n$ sets, m for each of the n sizes. As shown in Figure 6a, each set in the ATD is assigned one of the n sizes. The insertion policy in

these sets of the ATD differs from the insertion policy in the MTD in that the assigned size is prioritized. For the example in Figure 6a, there are only two possible block sizes. Sets A and F in the ATD prioritize insertions of 8-byte blocks. Sets D and I prioritize the insertion of 64-byte blocks. Sets B, C, E, G, and H are not sampled in the ATD. Both tag directories implement the same replacement policy.

When a set in the MTD that has a corresponding set in the ATD receives a miss, a counter CTR_i is incremented, where i is a size corresponding to the prioritized size in the corresponding ATD set. When an ATD set receives a miss, it decrements CTR_i for the size associated with the policy this set is helping decide. Figure 6b shows the decision of the output of CTR_{64B}.

For each of the possible compressed block sizes, a decision is made independently based on the result of the counter. If CTR_i is negative, prioritizing blocks of size i is negatively affecting miss rate (e.g., the insertion policy in the MTD resulted in fewer misses than the insertion policy in the ATD). Therefore, SIP does not prioritize blocks of size i. Likewise, if CTR_i is positive, prioritizing insertion of blocks of size i is reducing the miss rate and SIP inserts size i blocks with high priority for best performance. For n different sizes, there are 2^n possible insertion schemes and any may be chosen by SIP.

For simplicity and to reduce power consumption, the dynamic set sampling occurs during a periodic training phase[7] at which time the insertion policy of the MTD is unaffected by SIP. At the conclusion of the training phase, a steady state is entered and the MTD adopts the chosen policies and prioritizes the insertion of blocks of sizes for which CTR was positive during training.

SIP is general enough to be applicable to many replacement policies (e.g., LRU, RRIP, etc). In some cases (e.g., LRU), it is more effective to try inserting blocks with lower priority (e.g., LRU position) instead of higher priority as proposed above. We evaluate SIP with RRIP where blocks by default are inserted with a predicted long re-reference interval ($RRPV = 2^M - 2$). Therefore, in the ATD sets, the appropriate sizes are prioritized and inserted with a predicted short re-reference interval ($RRPV = 0$). For a 2MB cache with 2048 sets, we create an ATD with 32 sets for each of 8 possible block sizes. For simplicity, in our implementation we limit the number of sizes to eight by bucketing the sizes into eight size bins (i.e., bin one consists of sizes 0 – 8B, bin two consists of sizes 9 – 16B,..., and bin eight consists of sizes 57 – 64B).

3.4. CAMP for the V-Way Cache

In addition to being an effective mechanism for the traditional compressed cache with local replacement policy, the key ideas behind CAMP are even more effective when applied to a cache with a decoupled tag- and data-store and a global replacement policy. Towards this end, we propose Global SIP (or G-SIP) and Global MVE (or G-MVE). Together, we combine these into Global CAMP (or G-CAMP). For a traditional cache structure, a local replacement policy considers only the blocks within a *single set* for candidates to replace. The V-Way cache [31] (described in Section 3.1), with its decoupled tag- and data-store, enables a global replacement decision where the pool of potential candidates for replacement is much larger, increasing the effectiveness of caching.

[7] In our evaluations, we perform training for 10% of the time. For example, for 100 million cycles every 1 billion cycles.

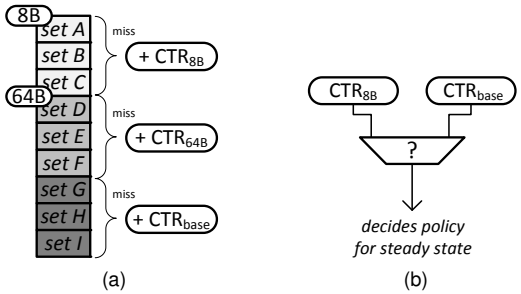


Figure 7: Set selection during training and update of counters on misses to each region.

3.4.1. G-MVE. As in MVE, G-MVE uses a value function to calculate the value of blocks. The changes required are in (i) computing p_i and (ii) selecting a pool of blocks from the large pool of replacement options to consider for one global replacement decision. To compute p_i, we propose using the reuse counters from the Reuse Replacement policy [31] as a predictor of future re-reference. As in the Reuse Replacement policy [31] (see Section 3.1), each data-store entry has a counter. On insertion, a block's counter is set to zero. On a hit, the block's counter is incremented by one indicating its reuse.

For the second change, we implement global replacement by maintaining a pointer (PTR) to a reuse counter entry. Starting at the entry PTR points to, the reuse counters of 64 valid data entries are scanned, decrementing each non-zero counter by one (as in Reuse Replacement policy). The 64 blocks are assigned a value, V_i, and the least-valued block(s) are evicted to accommodate the incoming block. 64 blocks are chosen because it guarantees both an upper bound on latency and that evicting all 64 blocks in the worst case (i.e., all highly compressed blocks) will vacate enough data-store space for the incoming block.

A few applications (i.e., *xalancbmk* [35]) have a majority of blocks of very similar sizes that primarily belong to two size bins of adjacent size. When considering 64 such blocks, certain blocks in the smaller size bin can essentially be "stuck" in the cache (i.e., there is only a very small probability these blocks will be chosen as victim, because a block with the same prediction of re-reference that belongs in the larger size bin is present and will be chosen). This results from the microarchitectural simplifications and approximate nature of the value function and can cause performance degradations in a few cases. We address this shortcoming in G-CAMP in Section 3.4.3.

3.4.2. G-SIP. Dynamic set sampling (used by SIP) motivates that only a select number of sets are required to be sampled to estimate the performance of competing policies [30]. However, this assumption does not hold in a cache with global replacement, because evictions are not limited to the set in which a cache miss occurs and this interferes with sampling. For the V-Way cache, we propose instead a mechanism inspired by set dueling [29] to select the optimal insertion policy.

To apply the set dueling to G-SIP, we need to divide the data-store into n (where n is small and in our evaluations $n = 8$) equal regions. Instead of the replacement policy considering all blocks within the data-store, only the blocks within a particular region are considered. This still allows considerably more replacement options than a traditional cache structure. We observe that this division also simplifies the V-Way cache

design with negligible impact on performance.[8]

During a training phase, each region is assigned a compressed block size to prioritize on insertion. Figure 7a shows this assignment for a simple cache with three regions and two block sizes, 8-byte and 64-byte. The third region is designated as a baseline or control region in which no blocks are inserted with higher priority. When a miss occurs within a region, the CTR counter is incremented for that region. For example, in Figure 7a a miss to set A, B, or C increments CTR_{8B}. Likewise, a miss to set G, H, or I increments CTR_{base} and so on. At the conclusion of the training phase, the region CTR counters are compared (see Figure 7b). If $CTR_i < CTR_{base}$, blocks of size i are inserted with higher priority in steady state in all regions. Therefore, G-SIP detects at runtime the sizes that reduce the miss rate when inserted with higher priority than other blocks.

In our implementation we have divided the data-store into eight regions.[9] This number can be adjusted based on cache size. Because one region is designated as the baseline region, we bin the possible block sizes into seven bins and assign one range of sizes to each region. During the training phase, sizes within this range are inserted with higher priority. The training duration and frequency is as in SIP. Because training is short and infrequent, possible performance losses due to set dueling are limited.

3.4.3. G-CAMP. G-MVE and G-SIP complement each other and can be easily integrated into one comprehensive replacement policy referred to as G-CAMP. We make one improvement over the simple combination of these two orthogonal policies to further improve performance in the few cases where G-MVE degrades performance as described in Section 3.4.1. During the training phase of G-SIP, a region is designated in which blocks are inserted with simple Reuse Replacement instead of G-MVE. At the conclusion of the training phase, the CTR for this region is compared with the control region and if fewer misses were incurred, G-MVE is disabled in steady state in all regions. In G-MVE-friendly applications, it remains enabled.

3.5. Overhead and Complexity Analysis

Table 1 shows the storage overhead in terms of additional bits for our mechanisms (baseline uncompressed cache, BDI compressed cache with LRU, V-Way with and without compression, as well as CAMP and G-CAMP). On top of our reference cache with BDI and LRU (2384kB), MVE does not add any additional metadata and the dynamic set sampling in SIP increases the cache size in bits by only 1.5% (total CAMP size: 2420kB). Adding BDI compression to V-Way cache with 2x tags and 8 regions increases the size from 2384kB to 2499kB. G-MVE/G-SIP/G-CAMP do not add further metadata (with the exception of eight 16-bit counters for set-dueling in G-SIP/G-CAMP). In addition, none of the proposed mechanisms are on the critical path of the execution and the logic is reasonably modest to implement (e.g., comparisons of CTRs). We conclude that the complexity and storage overhead of CAMP are modest.

[8] G-MVE supports regions by simply maintaining one PTR per region.

[9] We conducted an experiment varying the number of regions (and therefore the number of distinct size bins considered) from 4 to 64 and found 8 regions performed best.

	Base	BDI	CAMP	V-Way	V-Way+C	G-CAMP
tag-entry(bits)	21	35([27])	35	29 [a]	43	43
data-entry(bits)	512	512	512	524 [b]	524	524
# tag entries	32768	65536	73728 [c]	65536	65536	65536
tag-store (kB)	84	287	322	237	352	352
other	0	0	8*16 [d]	0	0	8*16
total (kB)	2132	**2384**	**2420**	2384	**2499**	**2499**

Table 1: Storage overhead of different mechanisms for a 2MB L2 cache. "V-Way+C" means V-Way with compression. The number of data entries is the same in all designs (32768).

[a] +8 forward ptr; [b] +3 reuse, +9 reverse ptr; [c] +1/8 set sampling in **SIP**; [d] CTR's in **SIP**

4. Qualitative Comparison with Prior Work

Size-Aware Management in On-Chip Caches. Baek et al. proposed Effective Capacity Maximizer (ECM) [5]. This mechanism employed size-aware insertion and replacement policies for an on-chip compressed cache. Unlike size-oblivious DRRIP [19] on which it is built, ECM inserts big blocks with lower priority than small blocks. The threshold for what is considered a "big" block is determined dynamically at runtime using an equation derived from heuristics and based on the current effective capacity and physical memory usage. During replacement, the biggest block in the eviction pool is selected as the victim.

ECM is the first size-aware policy employed for compressed on-chip caches. We find that this approach has several shortcomings and underperforms relative to our proposed mechanisms (as we show in Section 6). First, the threshold scheme employed by ECM is coarse-grained and, especially in multi-core workloads where a greater diversity of block sizes exists across workloads, considering more sizes (as CAMP does) yields better performance. Second, ECM's mechanism does not consider the relation between block reuse and size, whereas CAMP exploits the new observation that block size and reuse can sometimes be related. Third, ECM's complex threshold definition makes it unclear how to generalize to a cache with global replacement, where size-aware replacement policies demonstrate highest benefit (as shown in Section 6). In contrast, CAMP is easily adapted to work with such caches.

Size-Aware Management in Web Caches. Prior works in web caches have proposed many management strategies that consider object size, e.g., variable document size. ElAarag and Romano [14, 32] provide one of the most comprehensive surveys. While these proposed techniques serve much the same purpose as a management policy for an on-chip cache (e.g., making an informed decision on the optimal victim), they do so in a much different environment. Many proposed mechanisms rely on a recency list of all objects in the cache (e.g., [1]) or consider frequency of object access (e.g., [11]), which are prohibitively expensive techniques for an on-chip cache. In addition, these techniques do not consider a higher density of information that comes with the smaller blocks after compression. This higher density can lead to a higher importance of the smaller blocks for the cache that was mostly ignored in these prior mechanisms.

Some prior works (e.g., [6, 8]) proposed function-based replacement policies that calculate the value of an object much like our proposed MVE policy. In particular, Bahn et al. [6] proposed a mechanism where the *value* of a block is computed as the division of re-reference probability and the relative cost of fetching by size. Similar to other function-based techniques,

Processor	1–4 cores, 4GHz, x86 in-order
L1-D cache	32kB, 64B cache-line, 2-way, 1 cycle, uncompressed
L2 caches	1–16 MB, 64B cache-line, 16-way, 15–48 cycles
Memory	300 cycle latency, 32 MSHRs

Table 2: Major parameters of the simulated system.

however, these inputs cannot efficiently be computed or stored in hardware. Our proposed technique does not suffer from this problem and requires only simple parameters already built into on-chip caches.

5. Methodology

We use an in-house, event-driven 32-bit x86 simulator whose front-end is based on Simics [24]. All configurations have a two-level cache hierarchy, with private L1 caches and a shared, inclusive L2 cache. Major simulation parameters are provided in Table 2. All caches uniformly use a 64B cache block size. All cache latencies were determined using CACTI [36] (assuming a 4GHz frequency). We also checked that these latencies match the existing last level cache implementations from Intel and AMD, when properly scaled to the corresponding frequency.[10] For single-core and multi-core evaluations, we use benchmarks from the SPEC CPU2006 suite [35], two TPC-H queries [37], and an Apache web server. All results are collected by running a representative portion (based on PinPoints [26]) of the benchmarks for 1 billion instructions. We build our energy model based on McPat [23], CACTI [36], and Synopsys Design Compiler with 65nm library (to evaluate the energy of compression/decompression with BDI).

Metrics. We measure performance of our benchmarks using IPC (instruction per cycle), effective compression ratio (effective increase in the cache size, e.g., 1.5 for 2MB cache means effective size of 3MB), and MPKI (misses per kilo instruction). For multi-programmed workloads we use the weighted speedup [34] as the performance metric: $(\sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}})$. Effective compression ratio for all mechanisms is computed without meta-data overhead [27].

We measure the memory subsystem energy that includes the static and dynamic energy consumed by L1 and L2 caches, memory transfers, and DRAM, as well as the energy of BDI's compressor/decompressor units. Energy results are normalized to the energy of the baseline system with a 2MB compressed cache and an LRU replacement policy. BDI was fully implemented in Verilog and synthesized to create some of the energy results used in building our power model.

Our results show that there are benchmarks that are almost insensitive (IPC improvement less than 5% with 32x increase in cache size) to the size of the L2 cache: dealII, povray, calculix, gamess, namd. This typically means that their working sets mostly fit into the L1D cache, leaving almost no potential for any L2/memory optimization. Therefore, we do not present data in detail for these applications, although we verified that our mechanism does not affect their performance.

Parameters of Evaluated Schemes. For FPC (BDI), we used a decompression latency of 5 cycles [4] (1 cycle [27]), respectively. A segment size of 1 byte is used for both designs to get the highest compression ratio as described in [4, 27]. For both FPC and BDI, we assume double the number of tags (and hence the compression ratio cannot be larger than 2.0) compared to the conventional uncompressed cache.

[10] Intel Xeon X5570 (Nehalem) 2.993GHz, 8MB L3 - 35 cycles [25]; AMD Opteron 2.8GHz, 1MB L2 - 13 cycles [9].

6. Results and Analysis

6.1. Single-core Results

6.1.1. Effect on Performance. Figures 8 and 9 show the performance improvement of our proposed cache management policies over the baseline design with a 2MB compressed[11] L2 cache and an LRU replacement policy. Figure 8 compares the performance of CAMP's local version (and its components: MVE and SIP) over (i) the conventional LRU policy [12], (ii) the state-of-the-art size-oblivious RRIP policy [19], and (iii) the recently proposed ECM policy [5]. Figure 9 provides the same comparison for G-CAMP (with its components: G-MVE and G-SIP) over (i) LRU, (ii) RRIP, and (iii) V-Way design policy [31]. Both figures are normalized to the performance of a BDI-cache with LRU replacement. Table 3 summarizes our performance results. Several observations are in order.

Mechanism	LRU	RRIP	ECM
MVE	6.3%/-10.7%	0.9%/-2.7%	0.4%/-3.0%
SIP	7.1%/-10.9%	1.8%/-3.1%	1.3%/-3.3%
CAMP	**8.1%/-13.3%**	**2.7%/-5.6%**	**2.1%/-5.9%**

Mechanism	LRU	RRIP	ECM	V-Way
G-MVE	8.7%/-15.3%	3.2%/-7.8%	2.7%/-8.0%	0.1%/-0.9%
G-SIP	11.2%/-17.5%	5.6%/-10.2%	5.0%/-10.4%	2.3%/-3.3%
G-CAMP	**14.0%/-21.9%**	**8.3%/-15.1%**	**7.7%/-15.3%**	**4.9%/-8.7%**

Table 3: Performance (IPC) / Miss rate (MPKI) comparison between our cache management policies and prior works, 2MB L2 cache. All numbers are pairwise percentage improvements over the corresponding comparison points and averaged across memory intensive applications.

First, our G-CAMP and CAMP policies outperform all prior designs: LRU (by 14.0% and 8.1%), RRIP (by 8.3% and 2.7%), and ECM (by 7.7% and 2.1%) on average across memory intensive applications (*GMeanIntense*, with MPKI > 5). These performance improvements are coming from both components in our designs that significantly decrease applications' miss rate (shown in Table 3). For example, MVE and G-MVE are the primary sources of improvements in *astar*, *sphinx3* and *mcf*, while SIP is effective in *soplex* and *GemsFDTD*. Note that if we examine all applications, then G-CAMP outperforms LRU, RRIP and ECM by 8.9%, 5.4% and 5.1% on average across these applications.

Second, our analysis reveals that the primary reasons for outperforming ECM are (i) ECM's coarse-grain view of the size (only large vs. small blocks are distinguished), and (ii) difficulty in identifying the right threshold for an application. For example, in *soplex* ECM defines every block that is smaller than or equal to 16 bytes as a small block and prioritizes it (based on ECM's threshold formula). This partially helps to improve performance for some important blocks of size 1 and 16, but our SIP mechanism additionally identifies that it is even more important to prioritize blocks of size 20 (these blocks

[11] Unless otherwise stated, we use 2MB BDI [27] compressed cache design.

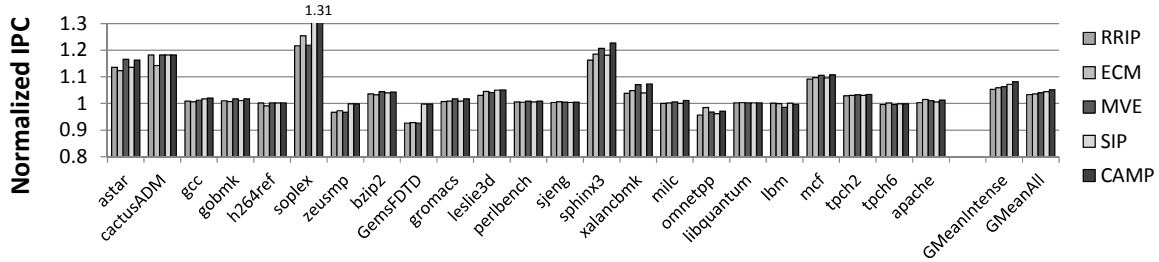


Figure 8: Performance of our local replacement policies vs. RRIP and ECM.

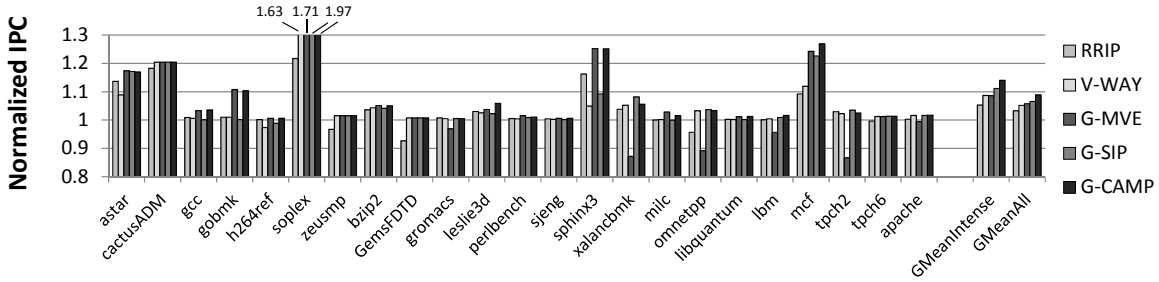


Figure 9: Performance of our global replacement policies vs. RRIP and V-Way.

have a significant fraction of blocks with short reuse distance as we show in Section 2.3). This in turn leads to much better performance in *soplex* by using CAMP (and G-CAMP).

Third, in many applications G-MVE significantly improves performance (e.g., *soplex* and *sphinx3*), but there are some noticeable exceptions (e.g., *xalancbmk*). The main reason for this problem is described in Section 3.4.1. The problem is avoided in our final mechanism (G-CAMP) where we use the set dueling mechanism [29] to dynamically detect such situations and disable G-MVE (for these cases only) within G-CAMP. As a result, our G-CAMP policy gets the best of G-MVE when it is effective and avoids degradations otherwise.

Fourth, global replacement policies (e.g., G-CAMP) are more effective in exploiting the opportunities provided by the compressed block size. G-CAMP not only outperforms local replacement policies (e.g., RRIP), but also global designs like V-Way (by 3.6% on average across all applications and by *4.9%* across memory intensive applications).

Based on our results, we conclude that our proposed cache management policies (G-CAMP and CAMP) are not only effective in delivering performance on top of the existing cache designs with LRU replacement policy, but also provide significant improvement over state-of-the-art mechanisms.

We summarize the performance gains and the decrease in the cache miss rate (MPKI) for all our policies in Table 3.

Sensitivity to the Cache Size. The performance benefits of our policies are significant across a variety of different systems with different cache sizes. Figure 10 shows the performance of the cache designs where (i) L2 cache size varies from 1MB to 16MB, and (ii) the replacement policies vary as well: LRU, RRIP, ECM, V-Way, CAMP, and G-CAMP.[12] Two observations are in order.

First, G-CAMP outperforms all prior approaches for all corresponding cache sizes. The performance improvement varies from 5.3% for 1MB L2 cache to as much as 15.2% for 8MB L2 cache. CAMP also outperforms all local replacement designs (LRU and RRIP).

[12] All results are normalized to the performance of the 1MB L2 compressed cache with LRU replacement policy. Cache access latency is modeled and adjusted appropriately for increasing cache size.

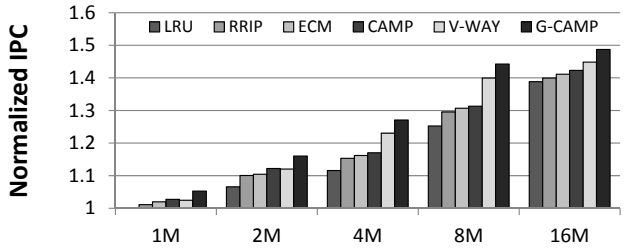


Figure 10: Performance with 1M – 16MB L2 caches.

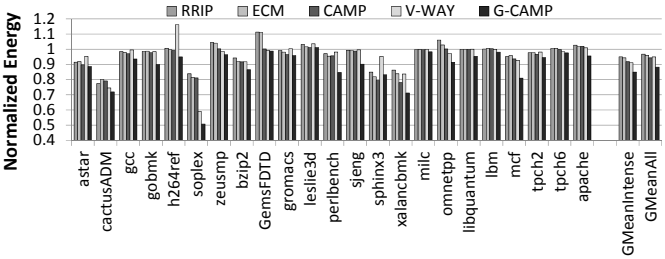


Figure 11: Effect on memory subsystem energy.

Second, the effect of having size-aware cache management policies like G-CAMP in many cases leads to performance that is better than that of a twice as-large cache with conventional LRU replacement policy (e.g, 4MB G-CAMP outperforms 8MB LRU). In some cases (e.g., 8MB), G-CAMP performance is better than that of a twice as-large cache with any other replacement policy. We conclude that our management policies are efficient in achieving the performance of higher capacity last-level-caches without making them physically larger.

6.1.2. Effect on Energy. By decreasing the number of transfers between LLC and DRAM, our management policies also improve the energy consumption of the whole main memory hierarchy. Figure 11 shows this effect on the memory subsystem energy for two of our mechanisms (CAMP and G-CAMP) and three state-of-the-art mechanisms: (i) RRIP, (ii) ECM, and (iii) V-Way. Two observations are in order.

First, as expected, G-CAMP is the most effective in decreasing energy consumption due to the highest decrease in MPKI (described in Table 3). The total reduction in energy consumption over the baseline system is 15.1% on average for memory intensive workloads (11.8% for all applications); 7.2% – over

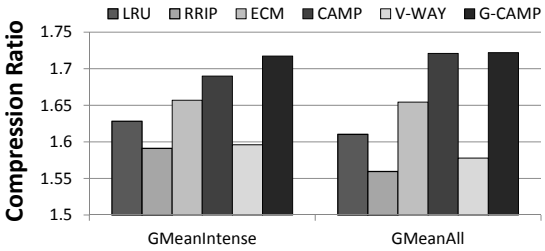


Figure 12: Effect on compression ratio with a 2MB L2 cache.

the best prior mechanism. We conclude that our cache management policies are more effective in decreasing the energy consumption of the main memory subsystem than previously proposed mechanisms.

Second, applications that benefit the most are usually the same applications that also have the highest performance improvement and the highest decrease in off-chip traffic, e.g., *soplex* and *mcf*. At the same time, there are a few exceptions like the *perlbench* workload that demonstrate significant reduction in energy consumed by the memory subsystem, but do not show significant performance improvement (as shown in Figures 8 and 9). For these applications, the main memory subsystem is usually not a bottleneck (for both energy and performance) due to the relatively small working set sizes that fit into the 2MB L2 cache and hence the relative improvements in the main memory subsystem might not have noticeable effects on the overall system performance.

6.1.3. Effect on Cache Capacity. We expect that size-aware cache management policies increase the effective cache capacity by increasing the effective compression ratio. Figure 12 aims to verify this expectation by showing the average compression ratios for applications in our workload pool (both the overall average and the average for memory intensive applications). We make two major observations.

First, as expected, our size-aware mechanisms (CAMP and G-CAMP) significantly improve effective compression ratio over corresponding size-oblivious mechanisms (RRIP and V-Way) – 16.1% and 14.5% (on average across all applications). The primary reason for this is that RRIP and V-Way are designed to be aggressive in prioritizing blocks with potentially higher reuse (better locality). This aggressiveness leads to an even lower average compression ratio than that of the baseline LRU design (but still higher performance shown in Section 6.1.1). Second, both CAMP and G-CAMP outperform ECM by 6.6% and 6.7% on average across all applications for reasons explained in Section 4. We conclude that our policies achieve the highest effective compression in the cache compared to the other three state-of-the-art mechanisms.

6.1.4. Comparison with Uncompressed Cache. Note that the overhead of using cache compression designs mostly consists of the increased number of tags (e.g, 7.6% for BDI [27]). If this number of bits (or even larger number of bits, e.g., 10%) is spent on having a larger L2 cache, for example, on a 2.2MB *uncompressed* L2 cache with RRIP replacement policy, we find that this system's performance is 2.8% lower than the performance of the baseline system with 2MB *compressed* L2 cache and LRU replacement policy, and 12.1% lower than the performance of the system with the same-size cache and G-CAMP policy. We conclude that using a compressed cache with CAMP policies provides a reasonable tradeoff in complexity for the achieved performance.

6.1.5. SIP with Uncompressed Cache. Our SIP policy can be applied to a cache without a compressed data-store, but still with knowledge of a block's compressibility. Evaluating such a design interestingly isolates the effects of smarter replacement and increased cache capacity. The performance improvement of our mechanisms ultimately comes from increasing the utility of the cache which may mean increasing capacity, smarter replacement, or both. Our evaluations with G-SIP without compression shows a 2.2% performance improvement over an uncompressed V-Way cache. This performance comes from better management only, through the use of compressibility as an indicator of reuse.

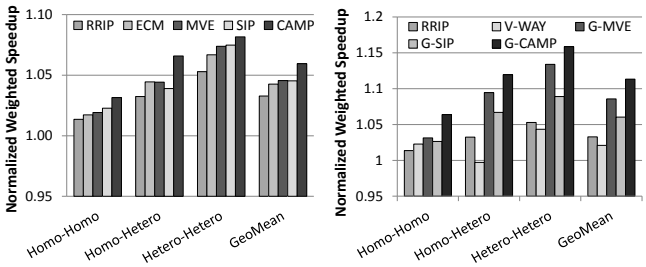
6.2. Multi-core Results

When the cache blocks from the working set of an application are compressed to mostly the same size, it is hard to expect that size-aware cache management policies would provide significant benefit. However, when different applications are running together in the multi-core system with a shared last-level-cache (LLC), there is a high chance that different applications will have different compressed sizes. As a result, we hypothesize that there is much more room for improvement with size-aware management in multi-core systems.

To test this hypothesis, we classify our applications into two distinct categories (*homogeneous* and *heterogeneous*) based on the distributions of the compressed sizes that they have. A homogeneous application is expected to have very few different compressed sizes for its data (when stored in the LLC). A heterogeneous application, on the other hand, has many different sizes. To formalize this classification, we first collect the access counts for different sizes for every application. Then, we mark the size with the highest access count as a "peak" and scale all other access counts with respect to this peak's access count. If a certain size within an application has over 10% of the peak access count, it is also marked as a peak. The total number of peaks is our measure of the application's heterogeneity with respect to the size. If the application's number of peaks exceeds a certain threshold (currently, two in our evaluations), we classify it as a heterogeneous (or simply *Hetero*) application, otherwise, the application is considered to be homogeneous (or simply *Homo*). This classification matches our intuition that applications that only have one or two common sizes (e.g., one size for uncompressed blocks and one size for most of the compressed blocks) should be considered homogeneous. These two classes enable us to construct three different 2-core workload groups: (i) Homo-Homo, (ii) Homo-Hetero, and (iii) Hetero-Hetero. We generate 20 2-core workloads per group (60 total) by randomly selecting applications from different categories (Homo or Hetero).

Figures 13a and 13b show the performance improvement provided by all CAMP designs as well as previously proposed designs: (i) RRIP, (ii) ECM, and (iii) V-Way over a 2MB baseline compressed cache design with LRU replacement policy. We draw three major conclusions.

First, both G-CAMP and CAMP outperform all prior approaches in all categories. Overall, G-CAMP improves system performance by 11.3%/7.8%/6.8% over LRU/RRIP/ECM (and CAMP – by 5.9%/2.5%/1.6% over the same designs). (We also verified that the effect on system fairness [22] by our mechanisms is negligible.)



(a) Local replacement designs. (b) Global replacement designs.

Figure 13: Normalized weighted speedup, 2-cores with 2MB L2.

Second, as we described at the beginning of this section, the more heterogeneity present, the higher the performance improvement with our size-aware management policies. This effect is clearly visible in both figures, and especially for global replacement policies in Figure 13b. G-CAMP achieves the highest improvement (15.9% over LRU and 10.0% over RRIP) when both applications are heterogeneous, and hence there are more opportunities in size-aware replacement.

Third, when comparing relative performance of MVE vs. SIP from Figure 13a and the similar pair of G-MVE vs. G-SIP from Figure 13b, we can notice that in the first pair the relative performance is almost the same, while in the second pair G-MVE is significantly better than G-SIP. The primary reason for this difference is that G-MVE can get more benefit from global cache replacement, because it can easily exploit size variation between different sets. At the same time, G-SIP gets its performance improvement from the relation between the size and corresponding data reuse, which does not significantly change between local and global replacement.

We conducted a similar experiment[13] with 30 4-core workloads and observe similar trends to the 2-core results presented above. G-CAMP outperforms the best prior mechanism by 8.8% on average across all workloads (by 10.2% across memory intensive workloads). We conclude that our policies provide better performance than state-of-the-art mechanisms in a multi-core environment.

6.3. Sensitivity to the Compression Algorithm

So far we have presented results only for caches that use BDI compression [27], but as was described in Section 2, our proposed cache management policies are applicable to different compression algorithms. We verify this by applying our mechanisms to a compressed cache design based on the FPC [3] compression algorithm. Compared to an FPC-compressed cache with LRU replacement, CAMP and G-CAMP significantly improve performance of memory intensive applications by 7.8% and 10.3% respectively. We conclude that our cache management policies are effective for different compression designs where they deliver the highest overall performance when compared to state-of-the-art mechanisms.

7. Conclusion

This paper presents Compression-Aware Management Policies (CAMP) – a set of new and simple, yet efficient *size-aware* replacement policies for compressed on-chip caches that improve system performance and energy efficiency compared to three state-of-the-art cache replacement mechanisms. Our policies are based on two key observations. First, we show that direct incorporation of the compressed cache block size into replacement decisions can be a basis for a more efficient replacement policy. We design such a policy and call it Minimal-Value Eviction due to its key property of replacing the blocks with the least importance (or value) for the cache, where the value is computed from the expected block reuse and size. Second, we find that the compressed block size can be used as an indicator of a block's future reuse in an application. We use this idea to define a new Size-based Insertion Policy that dynamically prioritizes blocks based on their compressed size.

We describe the design and operation of our compressed cache management policies (CAMP and G-CAMP) that can utilize these ideas for caches with both local and global replacement strategies, with minimal hardware changes. We also show that our ideas are generally applicable to different cache compression algorithms. Our extensive evaluations across a variety of workloads and system configurations show that our policies applied to modern last-level-caches (LLC) can improve performance by as much as 4.9%/9.0%/10.2% (on average for memory intensive workloads) for single-core/two-/four-core workloads over the best state-of-the-art replacement mechanisms. In many cases, the effect of using CAMP-based management policies is better than the performance of doubling the LLC capacity. We conclude that CAMP is an efficient and low-complexity management policy for compressed caches in both single- and multi-core systems.

[13] We increased the LLC size to 4MB to provide the same core to cache capacity ratio as with 2-cores.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams, "Removal Policies in Network Caches for World-Wide Web Documents," in *SIGCOMM*, 1996.

[2] K. K. Agaram, S. Keckler, C. Lin, and K. S. McKinley, "Decomposing Memory Performance: Data Structures and Phases," in *ISMM-5*, 2006.

[3] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *ISCA-31*, 2004.

[4] A. R. Alameldeen and D. A. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," *Tech. Rep.*, 2004.

[5] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "ECM: Effective Capacity Maximizer for High-Performance Compressed Caching," in *HPCA-19*, 2013.

[6] H. Bahn, S. H. Noh, S. L. Min, and K. Koh, "Using full Reference History for Efficient Document Replacement in Web Caches," in *USITS*, 1999.

[7] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, pp. 78–101, 1966.

[8] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," in *USENIX Symposium on ITS*, 1997.

[9] J. Chen and W. W. Iii, "Multi-Threading Performance on Commodity Multi-core Processors," in *Proceedings of HPCAsia*, 2007.

[10] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," in *T-VLSI Systems*, 2010.

[11] K. Cheng and Y. Kambayashi, "A Size-Adjusted and Popularity-Aware LRU Replacement Algorithm for Web Caching," in *COMPSAC-24*, 2000.

[12] P. J. Denning, "The Working Set Model for Program Behavior," *Commun. ACM*, 1968.

[13] C. Ding and Y. Zhong, "Predicting Whole-program Locality Through Reuse Distance Analysis," in *PLDI*, 2003.

[14] H. ElAarag and S. Romano, "Comparison of function based web proxy cache replacement strategies," in *SPECTS-12*, 2009.

[15] M. Feng, C. Tian, C. Lin, and R. Gupta, "Dynamic Access Distance Driven Cache Replacement," *TACO*, 2011.

[16] E. G. Hallnor and S. K. Reinhardt, "A Fully Associative Software-Managed Cache Design," in *ISCA-27*, 2000.

[17] E. G. Hallnor and S. K. Reinhardt, "A Unified Compressed Memory Hierarchy," in *HPCA-11*, 2005.

[18] M. Hayenga, A. Nere, and M. Lipasti, "MadCache: A PC-aware Cache Insertion Policy," in *JWAC*, 2010.

[19] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *ISCA-37*, 2010.

[20] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache Replacement Based on Reuse-Distance Prediction," in *ICCD*, 2007.

[21] M. Kharbutli and R. Sheikh, "LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm," in *Transactions on Computers*, 2013.

[22] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO-43*, 2010.

[23] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *MICRO-42*, 2009.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, pp. 50–58, February 2002.

[25] D. Molka, D. Hackenberg, R. Schone, and M. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *PACT-18*, 2009.

[26] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," *MICRO-37*, 2004.

[27] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches," in *PACT-21*, 2012.

[28] T. Piquet, O. Rochecouste, and A. Seznec, "Exploiting Single-Usage for Effective Memory Management," in *ACSAC-07*, 2007.

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA-34*, 2007.

[30] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *ISCA-33*, 2006.

[31] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way Cache: Demand Based Associativity via Global Replacement," in *ISCA-32*, 2005.

[32] S. Romano and H. ElAarag, "A Quantitative Study of Recency and Frequency Based Web Cache Replacement Strategies," in *CNS*, 2008.

[33] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT-21*, 2012.

[34] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," *ASPLOS-9*, 2000.

[35] SPEC CPU2006 Benchmarks, "http://www.spec.org/."

[36] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Laboratories, Tech. Rep. HPL-2008-20, 2008.

[37] Transaction Processing Performance Council, "http://www.tpc.org/."

[38] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A Modified Approach to Data Cache Management," in *MICRO-28*, 1995.

[39] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *MICRO-33*, 2000.