ECM:Effective Capacity Maximizer for High-Performance Compressed Caching

Seungcheol Baek[†], Hyung Gyu Lee[‡], Chrysostomos Nicopoulos[§], Junghee Lee[†] and Jongman Kim[†]
[†]Georgia Institute of Technology, USA, {bsc11235, junghee.lee, jkim}@gatech.edu
[‡]Daegu University, South Korea, hglee@daegu.ac.kr
[§]University of Cyprus, nicopoulos@ucy.ac.cy

Abstract

Compressed Last-Level Cache (LLC) architectures have been proposed to enhance system performance by efficiently increasing the effective capacity of the cache, without physically increasing the cache size. In a compressed cache, the cacheline size varies depending on the achieved compression ratio. We observe that this size information gives a useful hint when selecting a victim, which can lead to increased cache performance. However, no replacement policy tailored to compressed LLCs has been investigated so far. This paper introduces the notion of size-aware compressed cache management as a way to maximize the performance of compressed caches. Toward this end, the Effective Capacity Maximizer (ECM) scheme is introduced, which targets compressed LLCs. The proposed mechanism revolves around three fundamental principles: Size-Aware Insertion (SAI), a Dynamically Adjustable Threshold Scheme (DATS), and Size-Aware Replacement (SAR). By adjusting the eviction criteria, based on the compressed data size, one may increase the effective cache capacity and minimize the miss penalty. Extensive simulations with memory traces from real applications running on a full-system simulator demonstrate significant improvements compared to compressed cache schemes employing the conventional Least-Recently Used (LRU) and Dynamic Re-Reference Interval Prediction (DRRIP) [11] replacement policies. Specifically, ECM shows an average effective capacity increase of 15% over LRU and 18.8% over DRRIP, an average cache miss reduction of 9.4% over LRU and 3.9% over DRRIP, and an average system performance improvement of 6.2% over LRU and 3.3% over DRRIP.

1. Introduction

The seemingly irreversible shift toward multi-core microprocessor architectures has given rise to the Chip Multi-Processor (CMP) archetype. This new paradigm has magnified the "memory wall" phenomenon, which is the result of the increasing performance gap between logic and off-chip memory devices. As a result, the need for a high-performance Last-Level Cache (LLC) has become imperative, since a well-designed LLC is one of the most effective ways to bridge the logic-memory chasm. With burgeoning transistor integration densities [1], architects have partly responded to this challenge by dedicating increasing portions of the CPU's real estate to the LLC. Such large LLCs are becoming common in the workstation and server segments. Even though modern architectures provide large LLCs to improve system performance, the size is not large enough to completely hide slow off-chip memory latencies, because the working-set size of most applications tends to increase over time, as well. Thus, effectively utilizing a given size of cache has been one of the most important research challenges so far in the field of microprocessor design.

The *compression* of LLCs is one of the most attractive solutions to cope with increasingly large working sets, because storing *compressed* data in a cache increases the effective (logical) cache capacity, without physically increasing the cache size. Accordingly, this increased effective cache capacity can hold a larger working set and thereby improve the system performance significantly. This benefit has led researchers to develop various compressed LLC architectures by designing efficient compression algorithms [4, 6, 7, 13, 18, 21], or by architecting compression-aware cache structures to ease the allocation and management of variable-sized compressed cachelines [2, 3, 6, 8, 9, 12, 20, 21]. Most of the prior efforts, however, focused on minimizing the inherent deficiencies of compression-based schemes, such as compression/decompression latency, address remapping, and compaction overhead. While these studies have led to significant improvements and have enabled the efficient use of compressed LLCs, no prior work has developed a cache management policy (mainly cacheline replacement policy) tailored to compressed caches. In other words, existing cache management policies for compressed caches employ the traditional cacheline replacement policies, whereby only locality information is considered. However, our experiments in this work will clearly show that the conventional cache replacement policies cannot fully utilize the benefits of the compressed cache, because they fail to account for the cachelines' variable size.

Unlike a conventional cache architecture, the physical size of a stored cacheline *after compression* varies, based on the achieved compression ratio. This means that the eviction overhead (miss penalty) in a compressed LLC will also vary, based on the size of the evicted and evictee cachelines. This work will demonstrate that if the size information of the compressed cacheline is considered in the cache management process, the increase in the effective capacity of the compressed LLC will be maximized. A detailed motivational example will be presented in Section 3. Consequently, in addition to *locality* information, the cacheline *size* information should also be one of the prime determinants in identifying the appropriate victim cacheline in compressed caches. When properly combined, these two properties would maximize the effective cache capacity and minimize the eviction overhead.

Motivated by this realization, this paper proposes a *size-aware cache management policy*, called **Effective Capacity Maximizer (ECM)**, which is targeted at *compressed LLC architectures*. ECM revolves around three fundamental and closely intertwined policies: *Size-Aware Insertion (SAI), a Dynamically Adjustable Threshold Scheme (DATS)*, and *Size-Aware Replacement (SAR)*. If the size of a compressed cacheline is larger than a pre-defined threshold, the

SAI policy (i.e., insertion) gives said incoming cacheline higher priority to be evicted. DATS changes the threshold in real-time using workload characteristics. The SAR policy (i.e., eviction) selects the largest cacheline within a victim pool, in order to minimize the eviction overhead and maximize the effective capacity. ECM is very lightweight in terms of hardware implementation, and all mechanisms involved mostly re-use existing components within the compressed LLC infrastructure.

Extensive simulations with memory traces extracted from real multi-threaded workloads running on a full-system simulation framework validate the efficacy and efficiency of the proposed ECM mechanism. Specifically, simulations with a 2 MB compressed LLC configured as physically 4-way set associative, and logically up to 16-way, indicate that ECM increases the effective cache capacity in a compressed LLC by an average of 15.0% and 18.8%, as compared to the Least Recently Used (LRU) and Dynamic Re-Reference Interval (DRRIP) [11] policies, respectively. This effective capacity improvement increases cache performance by reducing the number of misses by an average of 9.4% over LRU and 3.9% over DRRIP. As a result, the ECM technique improves overall system performance (IPC) by 6.2% and 3.3%, as compared to a compressed LLC using the LRU and DRRIP replacement policies, respectively.

The rest of the paper is organized as follows: Section 2 provides background information and related work in LLC compression techniques and cache replacement policies, and decoupled variable-segment cache architectures. Section 3 presents a motivational example for the importance of cacheline size information in the cache management policy of compressed caches. Section 4 delves into the description, implementation, and analysis of the proposed ECM. Section 5 describes the employed evaluation framework, while Section 6 presents the various experiments and accompanying analysis. Finally, Section 7 concludes the paper.

2. Background & Related Work

2.1. Compression Techniques and Cache Replacement Policies in LLCs

Compression is an efficient way to increase the logical capacity of the memory system without increasing its physical capacity. Such compression is known to be very useful for the LLC memory architecture, where the capacity is one of the most sensitive factors. There have been many studies on the employment of compression techniques within the LLC micro-architecture. The focus so far has been on designing the compression-aware cache structure, rather than the compression algorithm itself. Existing approaches can be classified into two broad categories: (1) variable-segment caches, and (2) fixed-segment caches.

Variable-segment caches aim to maximize the effective capacity of the cache, even though they require non-negligible space for remapping and compaction, which is pure overhead [3, 20]. Since the number of occupied segments by the compressed data is variable, based on the compression ratio, the latter directly impacts the effective capacity. On the other hand, fixed-segment caches aim to exploit compression without any severe cacheline compaction or manipulation overhead, by fixing the number of segments occupied by the compressed cacheline (usually up to 2 or 4) [2, 6, 12, 21]. Due to the fixed segment size, if the compressed data size is very small, some segments might have empty space, which is not available to other cachelines. Our experiments indicate that this situation happens frequently under most compression algorithms – irrespective of their complexity. Most popular compression algorithms exploit *zero-value* compression [4, 18, 21], which achieves high compressibility, but still produces a lot of unusable empty space within the cache. This implies that compression algorithms do not fully utilize the physical space of the cache, even when the compression scheme is highly efficient. Since our focus is on maximizing the effective capacity, as well as reducing the cache miss penalty, we will mainly exploit the *variable-segment* cache architecture for the rest of this paper. Note, however, that the proposed ECM will also work in a *fixed-segment* cache architecture.

Besides the cache size, the cacheline replacement policy (when cache misses occur) is also an important factor that affects system performance. There has been extensive research in developing efficient cache management policies [10, 11, 14, 17, 19]. Generally, it is accepted that the LRU replacement policy (and its approximations) behaves relatively well with most applications and access patterns (e.g., recency-friendly and sequential streaming). However, LRU incurs performance degradation under thrashing and mixed-access patterns. To address these problems, one recent study proposed a Dynamic Re-Reference Interval Prediction (DRRIP) replacement mechanism [11]. DRRIP comprises two cache management policies: Bimodal RRIP (BRRIP) and Static RRIP (SRRIP). The BRRIP policy specifically addresses the performance of thrashing access patterns, while the SRRIP policy aims to improve the performance of mixed-access patterns. Since the *Set Dueling* method [17] is used to select the best performing among the two policies, the performance of DRRIP is shaped by how well it performs under the two aforementioned cache management policies.

Even though these advanced cache management algorithms enhance the cache performance beyond the conventional LRU policy – with low implementation cost – they still focus on exploiting locality (re-reference rate) information only. However, in compressed cache architectures, where the size of each cacheline is variable and a function of the compression ratio, the size information should also be considered, because it directly determines the effective cache size and the miss penalty. Nevertheless, no previous work attempted to develop a cache replacement policy targeting compressed caches. To the best of our knowledge, this is the first work to propose a compressed cache architecture augmented with a customized cache management policy that is aware of both the variable cacheline size information and the locality information.

2.2. Decoupled Variable-Segment Cache Architectures

Figure 1 illustrates the structure of a single set of a decoupled variable-segment cache architecture, assuming a 64B cacheline size [3]. While each cache set is physically 4-way set associative, logical 16-way set associativity

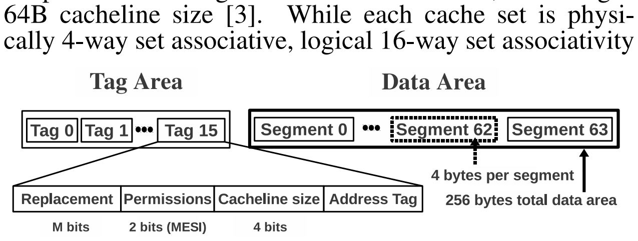


Figure 1. One set of a decoupled variable-segment cache.

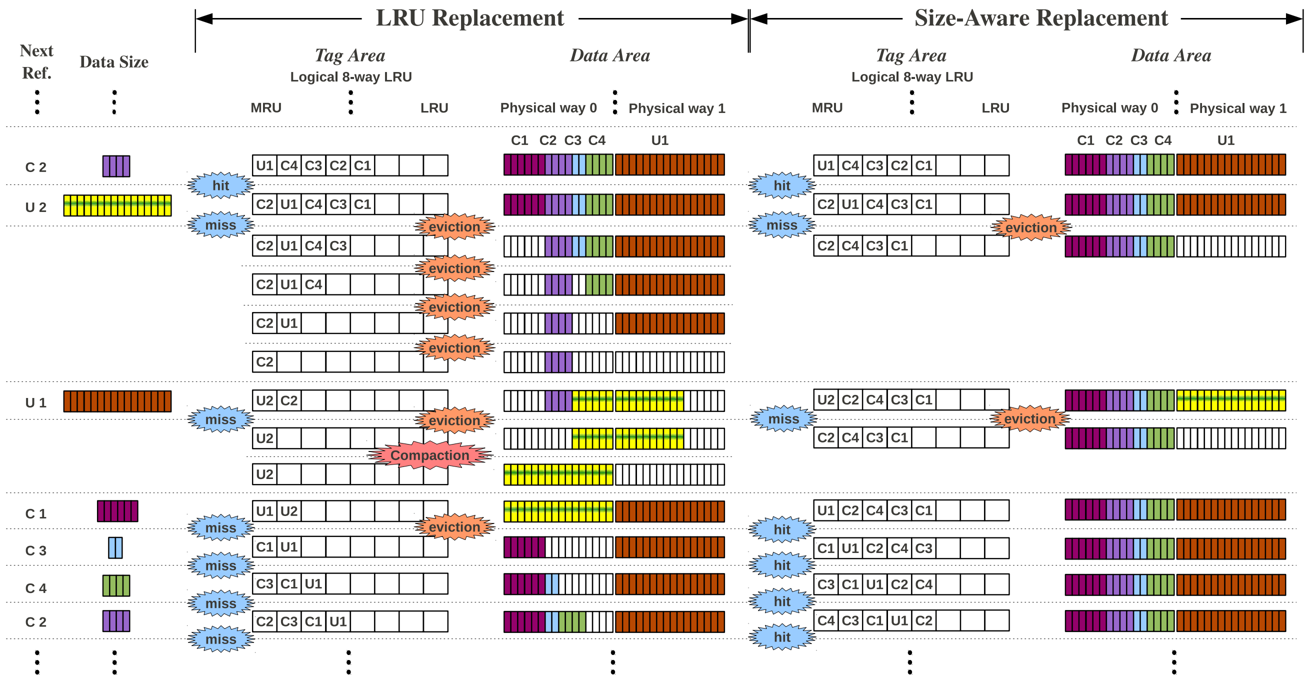


Figure 2. The behavior of the LRU replacement policy and a size-aware replacement policy in a compressed decoupled variable-segment cache design.

can be achieved by using more tags alongside a variable-segmented data area. More specifically, each set of the cache is broken into 64 segments and the size of each segment is only 4 bytes (single-word). The effective capacity for a single set is given by

$$\text{physically 4-way} \leq \text{effective capacity} \leq \text{logically 16-way}$$

where *physically 4-way* comes from the Data Area, and *logically 16-way* is achieved through the Tag Area. For instance, if there are only uncompressed cachelines, the cache would operate like a typical 4-way set associative cache. Conversely, if there are only highly compressed cachelines, the cache would operate with 16-way set associativity. Therefore, each set can potentially increase its effective capacity by up to four times (when storing 16 compressed cachelines). Of course, one could have more than 16 cachelines in a set with larger tag space. The scalability of the mechanism proposed in this work with the number of logical ways will be explored in Section 6.5. Data segments are stored contiguously in Address Tag order. The offset for the first data segment of cacheline k (in a particular set) is

$$\text{segment\_offset}(k) = \sum_{i=0}^{k-1} \text{actual\_size}(i)$$

A cacheline's actual size is determined by the *Cacheline size* tag in the Tag Area (Figure 1).

3. Motivation for a Size-Aware Cache Replacement Policy

In conventional cache architectures, the replacement policies are optimized to minimize the off-chip memory accesses, based on data access patterns. However, as previously mentioned, in a compressed cache, the size information of the cacheline is equally important and should be exploited in the optimization of both the cache structure itself and the cache replacement policy. Without this combined information, many of the benefits of compression could be lost. In this section, we demonstrate, through an example, why the size information should be considered in compressed caches.

Figure 2 shows the behavior of a compressed cache under the LRU replacement policy and a size-aware replacement policy. In this example, the cache is physically configured as a 2-way set associative cache, but logically configured as an 8-way set associative decoupled variable-segment cache. When U2 – a new cacheline – is requested, a cache miss occurs. Four cachelines (C1, C3, C4, and U1) should be evicted under the LRU replacement policy. These evictions will lead to an almost *four-fold* increase in the miss penalty, as compared to the penalty of a conventional cache without compression. After the evictions of said 4 cachelines, C2 will also be evicted when the second U1 request arrives. In addition, a compaction process to allocate U1 should be performed in the data area, which results in non-negligible overhead in decoupled variable-segment caches. More importantly, at this moment, the Data Area contains only 2 cachelines, which is the minimum of its potential effective capacity. Even worse, the cache will experience 4 consecutive misses in the next 4 requests (C1, C3, C4, and C2). Hence, in this conventional configuration, a total of 6 cache misses and 6 cacheline evictions are observed, and the average number of cachelines that the cache set holds is 3.29.

On the other hand, we can vastly improve this situation by considering the *size* information of the evicted cacheline. As opposed to the previous example, we evict only U1 – which is the largest-size cacheline in the Data Area – when U2 first arrives. C1, C3, C4, and C2 can stay in the cache. Even though U2 will again be replaced by U1 at the next request for U1, the following four requests (C1, C3, C4, and C2) after U1 will hit in the cache. Thus, in total, only

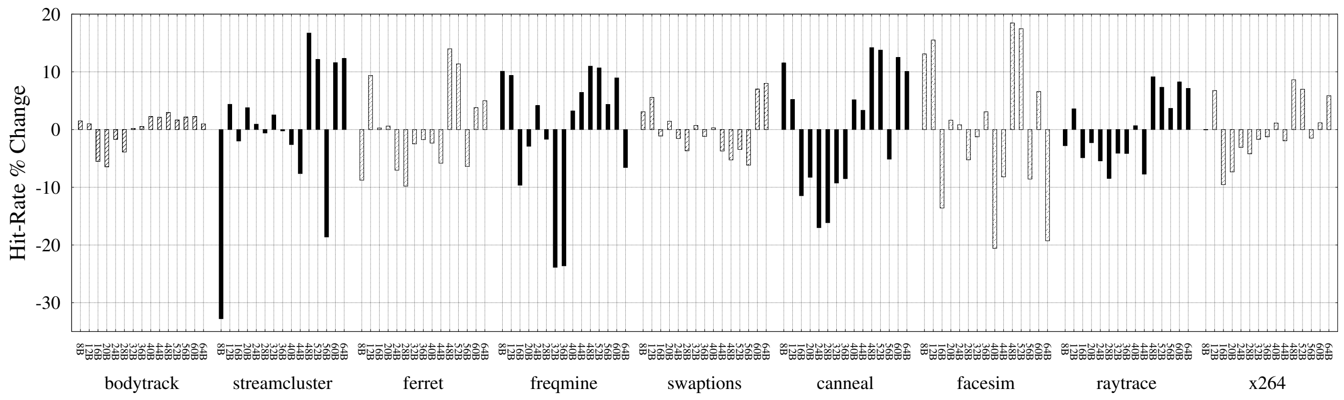


Figure 3. Hit-rate percentage change for each cacheline size over several PARSEC benchmarks [5].

2 cache misses and 2 cacheline evictions are observed, and the average number of cachelines in a cache set increases to 5.0 in this configuration.

This example emphatically demonstrates that it is imperative to consider the variable cacheline size in compressed caches, in order to maximize the performance enhancement afforded by the compression scheme.

4. The Effective Capacity Maximizer (ECM) Cache Management Scheme

As described in the previous section, the exploitation of cacheline size information when selecting a victim in compressed LLCs is a necessary condition to maximize the benefits of compression; namely, enhancing the effective capacity – which implies a reduction in the number of misses – and reducing the miss penalty. The simplest way of using the size information is to always evict the biggest-size cacheline first. However, our experiments have shown that this simple eviction policy may yield many negative side effects, because it does not consider the locality information of the data at all. For example, the biggest-size cacheline, which happens to exhibit high temporal locality, will be evicted before it is reused in the cache. Conversely, a highly compressed smallest-size cacheline having low temporal locality will stay almost indefinitely in the cache occupying valuable space. Therefore, using the cacheline size in isolation when selecting a victim will not be helpful in maximizing the cache performance. This clearly indicates that the size information should be considered *together* with locality.

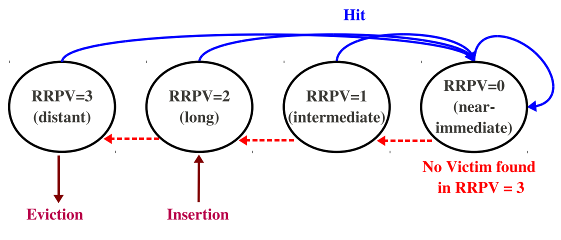
Based on this observation, we propose a size- and locality- aware cache management scheme, called Effective Capacity Maximizer (ECM), to further enhance the performance of compressed LLCs. ECM's operation revolves around three policies: Size-Aware Insertion (SAI), a Dynamically Adjustable Threshold Scheme (DATS), and Size-Aware Replacement (SAR). These basic policies can be used in conjunction with most previous replacement policies. However, in this paper, we are partially exploiting the basic framework from RRIP [11], because it supports relatively high performance with low implementation cost, as compared to other conventional implementations. In addition, the RRIP framework can easily provide most of the information required by ECM.

4.1. The Size-Aware Insertion (SAI) Policy

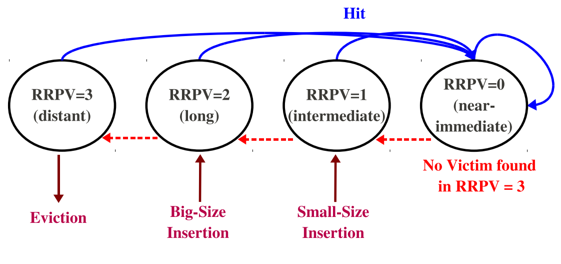
4.1.1. The Potential Conflicts Between Locality and Size Information: The main challenge in this work is the effective *simultaneous* consideration of the *locality* information and the *size* information, in order to maximize the benefits afforded by compression. Even though the locality and size information of a certain cacheline can vary over the execution timeline, one can simply classify a cacheline at a given time, t, into four possible status types: (1) high locality and small size, (2) high locality and big size (almost uncompressed), (3) low locality and small size, and (4) low locality and big size. For types (1) and (4), the decision is clear: keep (1) as long as possible, while evict (4) as soon as possible. If the biggest portion of the application favors these two types, the size-aware policy can easily be applied. On the other hand, for types (2) and (3), where the size and locality information are conflicting, a more complicated decision scheme is needed. Otherwise, the cache will experience performance degradation, even if the effective capacity increases through the consideration of size information. The solution for type (3) can still be relatively simple. We can evict this line type without any severe consequences, because the possibility of reusing it is low. However, the solution for type (2) is not easy, because the possibility of reusing such lines is high, but they occupy many segments (large space) in the cache. In order to consider this problematic case, a threshold-based classification process will be performed.

Before proceeding with the description of the main idea, we first analyze the relationship between the locality and size information and identify which types of cachelines are more dominant over others in real application workloads. To extract this information, several applications selected from the PARSEC benchmark suite [5] are executed under the assumption of a compressed cache using a 3-bit DRRIP[1]. The details of our evaluation framework are presented in Section 5. Figure 3 shows the hit ratio variation, based on the cacheline size (in bytes). We measure the hit ratio separately by cacheline size and normalize the values to the average hit ratio of the cache. So, if the bar is above zero, a larger number of hits (higher locality) is observed for that specific cacheline size. Instead, if a bar is below zero, fewer hits (low locality) are observed. Depending on the application, high locality is observed both in small-size cachelines – type (1) – and in big-size cachelines – type (2). Further,

[1]In this paper, we use 32-entry *Set Dueling Monitors*, a 10-bit single-policy selection (PSEL) counter, and ϵ=1/3 for DRRIP [10, 11, 17].



(a) RRIP framework



(b) SAI framework

Figure 4. The RRIP [11] and SAI Policy frameworks with a 2-bit (M=2) Re-Reference Prediction Value (RRPV).

type (3) and (4) cases are observed throughout the graph as well. This means that there is no clear relationship between the size and the locality, but a non-negligible number of type (2) cases (high locality and big size) are observed when the cacheline size is more than 48 B in most applications. Hence, our policy aims to maximize the benefits even under these conflicting cases.

4.1.2. The Re-Reference Interval Prediction (RRIP) Framework: Figure 4(a) shows the *Re-Reference Interval Prediction* (RRIP [11]) framework. In order to store one of 2^M possible *Re-Reference Prediction Values* (RRPVs), M bits are used for each cacheline in the RRIP policy. If a cacheline stores an RRPV of zero, it is predicted to be re-referenced (re-used) in the *near-immediate* future. On the other hand, if a cacheline stores an RRPV of $2^M - 1$, the cacheline is predicted to be re-referenced in the *distant* future. In other words, smaller RRPV cachelines are expected to be re-referenced sooner than cachelines with larger RRPV. Therefore, on a cache miss, RRIP selects a victim among the cachelines whose RRPV is $2^M - 1$ (*distant* re-reference interval). If there is no cacheline with an RRPV of $2^M - 1$, the RRPVs of all cachelines are increased by 1, and this increasing process is repeated until a victim cacheline is found. On a hit, the RRIP's promotion policy updates the RRPV of the cacheline to zero by predicting the cacheline to be re-referenced in the *near-immediate* future.

When new data is initially fetched from the memory device into a specific cacheline, the Static RRIP (SRRIP) policy sets the cacheline's initial RRPV as $2^M - 2$, which indicates *long* re-reference interval, instead of *distant* re-reference interval, to allow SRRIP more time to learn and improve the re-reference prediction. However, when the re-reference interval of all the cachelines is larger than the available cache, SRRIP causes cache thrashing with no hits. To address such scenarios, Bimodal RRIP (BRRIP) sets the majority of new cachelines' initial RRPVs as $2^M - 1$ (*distant* re-reference interval prediction), and it infrequently inserts new cachelines with RRPV of $2^M - 2$ (*long* re-reference interval prediction). Dynamic RRIP (DRRIP) determines which policy is best suited for an application between SRRIP and BRRIP using *Set Dueling* [17].

4.1.3. The Size-Aware Insertion (SAI) Policy: The main goal of the proposed SAI policy is to give the big-size cacheline a higher chance of *eviction* while minimizing the *conflict with locality*. For this purpose, we simply classify the cachelines into two types: big-size cachelines and small-size cachelines. We then adjust the insertion point of the cacheline's re-reference interval based on this classification. The classified big-size cachelines are allocated with higher RRPV than the small-size cachelines to force the big-size cachelines to be evicted soon. However, if we set the RRPV of big-size cachelines as $2^M - 1$ (*distant* re-reference interval prediction) like BRRIP, the big-size cachelines with high locality may also be evicted soon, due to the lack of time to learn locality information. Therefore, we set the RRPV of big-size cachelines as $2^M - 2$ (*long* re-reference interval prediction) and that of small-size cachelines as $2^M - 3$ (*intermediate* re-reference interval prediction), so that all the cachelines have enough time to learn locality information. The SAI policy framework is shown in Figure 4(b). Since the only modification is the insertion point (RRPV) of the RRIP chain, no significant alteration of the conventional approach is necessary to implement the SAI policy.

4.2. The Dynamically Adjustable Threshold Scheme (DATS)

Another critical issue in this scheme is how to classify a cacheline as either big-size or small-size. This classification is very important to balance the big-size cacheline thrashing effect and the re-reference interval prediction.

4.2.1. The Static Threshold Scheme (STS): In the SAI policy (Section 4.1.3), we can compare the size of the cacheline with a predefined threshold, T_h ($2 \leq T_h < 16$, where 16 is the size of the uncompressed cacheline in number of segments, and 2 is the number of segments occupied by the smallest possible compressed cacheline). Note that each segment is 4 B long, and *a threshold is defined based on a number of segments*. If the threshold is set too high, it means that the re-reference rate will be given more weight than the size information, while a lower threshold value means the exact opposite. There is no strong correlation between cacheline-size and re-reference interval, as observed in Figure 3, so, through a sensitivity analysis study

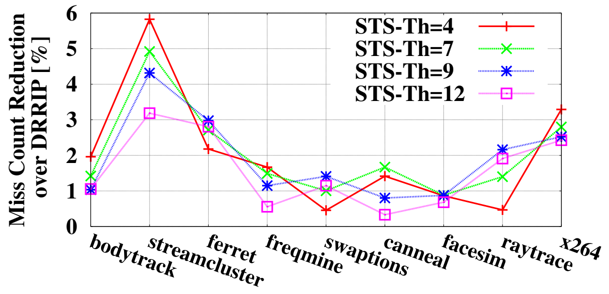


Figure 5. The Static Threshold Scheme (STS) performance in each PARSEC application [5]. Each curve represents a different cacheline size threshold, ranging from 4 segments (16 B) to 11 segments (44 B).

of statically setting the size-threshold value, we can empirically identify the threshold value that shows the best performance. We refer to this policy as the *Static Threshold Scheme (STS)*.

Even though one can find the performance-optimal threshold value through sensitivity analysis of each application, it is still not easy to implement this solution in real systems, because this threshold value varies with the applications. Figure 5 shows the STS performance for several selected *static threshold values* in each application. The y-axis represents the percent miss count reduction in a compressed LLC, compared to the DRRIP. As Figure 5 indicates, a *static threshold value* – which shows best or worst performance – varies with the application. In other words, a certain *static threshold value* can be best for an application, but it also can be worst for another application. In fact, for example, STS-Th=4 (a static threshold value set at 4 segments, i.e., 16 B) shows best performance with `streamcluster`, but worst performance with `raytrace`.

4.2.2. The Dynamically Adjustable Threshold Scheme (DATS): The optimal threshold value varies with *not only* the application, but also the execution timeline of each application. Figure 6 shows the *effective capacity* fluctuation and *physical memory usage* of `streamcluster` using the LRU replacement policy. The y-axis represents the cache size of the compressed LLC, which is physically 2MB, but can increase up to 8MB (logically), by overlaying a logical 16-way configuration onto a physically 4-way setup. The x-axis shows the timeline (cycles) for 0.9 billion cycles. Note that, in a single set of a compressed cache, the number of valid logical tags represents the *effective capacity*, while the total number of valid data segments in the data area represents the *physical memory usage*. Figure 6 highlights the fact that there are several sections of high effective capacity (indicating many small-size cachelines are being stored), and several sections of low effective capacity (indicating big-size cachelines are being stored). Obviously, the threshold value should be different for these sections.

Thus, a *Dynamically Adjustable Threshold Scheme* (DATS) is required, which changes the threshold value based on real-time *effective capacity* information and *physical memory usage* in a set, simultaneously. The threshold value is updated every time a new cacheline is inserted in a set (the threshold is calculated on a per-set basis). In order to efficiently determine the dynamically changing T_h while minimizing the implementation overhead in a set, we derive an equation as follows:

$$T_h = \left\lceil \left(\left(\frac{NS_{uc} \times W_P}{W_L} \right) + NS_{uc} - \left[NS_{uc} \times \left(1 - \frac{NT_v}{W_L} \right) \right] \right) \times \left(\frac{Size_{total}}{NS_{uc} \times W_P} \right) \right\rceil$$

NS_{uc} indicates the number of segments occupied by an uncompressed cacheline, while W_P and W_L indicate the physical number of ways and the logical number of ways in a set, respectively. These three terms are constant, i.e, NS_{uc}=16, W_P=4, and W_L=16 in this study. NT_v indicates the number of valid tags within a set (i.e., the *effective capacity*) and $Size_{total}$ indicates the total number of valid segments in the data area of a set (i.e., the *physical memory usage*). Only these two variables are defined when a new cacheline is inserted. The term $(NS_{uc} \times W_P)$ represents the total number of segments in the data area (i.e., 64). The right-most term in the equation above ($Size_{total}$ / $(NS_{uc} \times W_P)$) represents the physical memory usage ratio. The left-most term (($NS_{uc} \times W_P$) / W_L) indicates the average number of segments in a logical way (i.e., 4), and this term is inserted as a bias. Based on the sum of this bias and NS_{uc}, the threshold T_h will decrease as NT_v increases and $Size_{total}$ decreases.

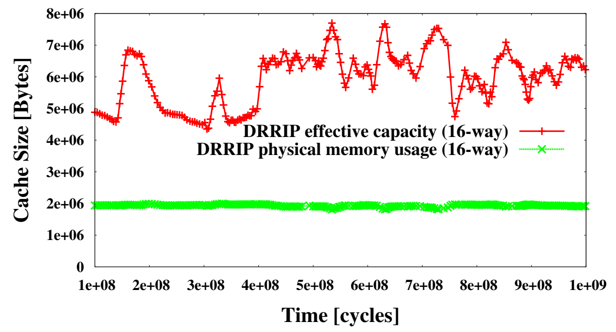


Figure 6. The effective capacity fluctuation and physical memory usage of `streamcluster`, when using the LRU replacement policy.

If the tags are fully used (NT_v=W_L=16), it means that the average size of cachelines in a set is relatively small, just like the high effective capacity sections shown in Figure 6. In high effective capacity sections, we need to change the threshold value with a sufficiently small-size value, so that we can classify a cacheline as either big-size or small-size in these sections. Furthermore, we also need to take into account the physical memory usage, in order to change the threshold more precisely, because, even if the tags are fully used in a set, the physical memory usage can range from *32 segments* (128 B) to *64 segments* (256 B). Therefore, we need to change the threshold with not only NT_v, but also $Size_{total}$. For example, when $Size_{total}$=*32 segments*, then the threshold value will be $(4 + 16 - 16) \times (32/64) = 2$, while for *64 segments* the threshold will be $(4 + 0) \times (64/64) = 4$, as per the equation above.

On the contrary, in the low effective capacity sections – which indicate the presence of several big-size cachelines – we need to change the threshold value with a larger size. In these sections, we also need to take into account the physical data area size, because the small number of valid tags does not reflect the fact that there are big-size cachelines in the data area all the time. For example, if there are 8 valid tags, then the physical data area size can range from *16 segments* (64 B) to *64 segments* (256 B). Therefore, based on NT_v and $Size_{total}$, the threshold value will be $(4 + 16 - 8) \times (16/64) = 3$ for *16 segments*, or $(4 + 16 - 8) \times (64/64) = 12$ for *64 segments*. If there are only uncompressed lines (i.e, NT_v=W_P=4 and $Size_{total}$=64 segments), the threshold value will be $(4 + 16 - 4) \times (64/64) = 16$. Thus, the size-aware insertion mechanism will be disabled, because no cacheline occupies more than 16 segments.

Since the NT_v and $Size_{total}$ parameters (1) are defined at design-time using the tag area information, (2) they have to be read, in order to check for a hit or miss when a new request arrives, and (3) they do not require any additional storage elements (the threshold is updated whenever a cacheline is inserted), this technique can be implemented in hardware fairly easily and efficiently. Although the above equation is derived from heuristics, we will demonstrate that it effectively balances the classification rate of the various compressed cachelines.

4.3. The Size-Aware Replacement (SAR) Policy

In the baseline RRIP [11] framework depicted in Figure 4, the victim is selected among the cachelines whose RRPV is 2^M-1 (*distant* re-reference interval; we refer to these cachelines as an *eviction pool*). Conventional methods tend to select the left-most victim (as shown in the figure), or a random victim, if there is more than one cacheline in the eviction pool. This is a reasonable choice, because all the cachelines in the eviction pool have already been "studied" in terms of their re-reference interval, and they have been predicted to be re-used in the *distant* future.

However, in *compressed* LLCs, the size information is an equally important factor, which significantly affects the performance of the cache. Thus, we exploit the size information again in this policy. As previously explained, the RRPVs of the cachelines in the eviction pool indicate *distant* re-reference intervals (they belong to types (3) and (4) of Section 4.1.1). Thus, we may consider the size information alone. In other words, the SAR policy simply selects the biggest-size cacheline as the victim from the eviction pool. Even though this is a very straightforward methodology, it is very effective in compensating for the weakness of the coarse-grained (binary) classification performed under the STS or DATS policies. Remember that STS and DATS classify new cachelines as big-size or small-size, based on a threshold. For example, let us assume that there are two cachelines having the same re-reference interval; one is quite bigger than the other, but due to the coarse-grained classification of STS and/or DATS, they were classified under the same type of cacheline (either big-size, or small-size). This means that their probability of being selected as victims is also the same, even though their actual sizes are not the same. Under the proposed scheme, the bigger-size cacheline will be evicted earlier than the smaller one if both cachelines find themselves in the eviction pool.

By evicting the largest-size cacheline first, SAR not only makes as much space as possible for future cachelines, but it also reduces the off-chip memory write requests. This is because SAR minimizes the number of evicted cachelines. In other words, without SAR, the number of evicted cacheline can increase so as to make enough space for a new cacheline, as described in Section 3. As a result, the number of flushed cachelines can also increase, which implies that the write-back buffer may stall more frequently.

The SAR mechanism selects the biggest-sized cacheline as a victim from the eviction pool, by using a 4-bit cacheline size information tag situated in the tag area. Hence, SAR requires 15 4-bit comparators in order to identify the biggest-sized cacheline. The same comparators are also used by the DATS mechanism, in order to classify cachelines as big-size, or small-size. This is, in fact, the only overhead incurred. Since the proposed ECM scheme is intended to be used on top of the RRIP [11] framework and a decoupled variable-segment cache architecture [3], all pertinent structures are already in place by said mechanisms. Other than the 15 comparators, *no significant alteration or additional storage is required.*

5. Experimental Methodology

5.1. Simulation Framework

We have developed a trace-driven simulator to evaluate the proposed ECM mechanism. Since the memory access sequence and the compression ratio are the most important factors for this evaluation, all traces have been extracted from the Simics [15] full-system simulator, extended with GEMS [16], while simulating a quad-core processor with a two-level cache hierarchy. Nine multi-threaded benchmarks from the PARSEC benchmark suite [5] are selected, and each benchmark runs for 300 million instructions. The L1 caches use an LRU replacement policy, and our study focuses on the cache management of the LLC. All the details of the simulation parameters are described in Table 1.

Table 1. Simulated System Parameters.

Number of CMP Cores	4
Processor Core Type	UltraSPARC-III+, 2 GHz
L1 caches (Private)	I- and D-caches 32 KB, 4-way, 64 B
L1 response latency	3 cycles
L2 caches (Shared)	2 MB, 4-way, 64 B, NUCA, MESI
L2 response latency	20 cycles
L2 read hit overhead	5 cycles for decompression
L2 writeback buffer	8 entries
Compaction overhead	16 cycles
DRAM memory	DDR2 4 GB
Memory response latency	450 cycles

5.2. The Compression Technique Employed in the LLC

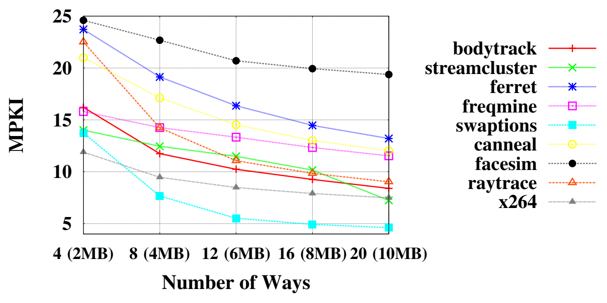
The baseline architecture targeted in this paper is one using compression only in the LLC. No compression is assumed in the L1 caches, or in main memory. As a compression algorithm for the LLC, we choose Frequent Pattern Compression (FPC) [4] – a bit-level compression algorithm – because its compression performance is relatively high, with reasonable compression overhead, both in terms of delay and implementation cost. We apply this FPC within a word, so a compressed cacheline's size varies from 8B (maximally compressed) to 64B (uncompressed). The maximally compressed cacheline occupies only 2 segments, while the uncompressed cacheline occupies 16 segments in the decoupled variable-segment LLC, as illustrated in Figure 1. A 2MB physically 4-way LLC is used. We set the maximum number of ways in a set to 16 (logical ways), so the effective capacity can increase up to 8 MB, which is a significant improvement.

In a compressed LLC, data compaction is required when the cache has room for upcoming requests (read miss, write hit, and write miss), but not in consecutive segments. In the case of compaction on a read miss, we do not account for the compaction overhead, because the compaction can be done while the data is being fetched from the main memory. This implies that no additional delay is incurred due to compaction. However, for both write hits and misses, the cacheline size changes and compaction is necessary in most requests. Without completing a compaction, data cannot be written in the cache. Thus, the compaction overhead (shown in Table 1) for write requests will be accurately accounted for in our evaluation.

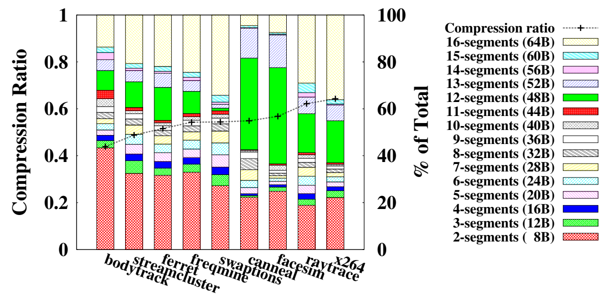
6. Evaluation and Analysis

6.1. Workload Characteristics

Before we proceed with the evaluation, it is important to analyze the salient workload characteristics that are important in the study of the proposed ECM mechanism. These characteristics are the set-associativity sensitivity, the compression ratio, the cacheline size proportions, and the locality attributes of each cacheline size. When we overlay a



(a) Cache Set-Associativity Sensitivity



(b) Compression Ratio and Cacheline Proportions

Figure 7. Salient characteristics of the application workloads used in this study.

logically 16-way cache on a physically 4-way 2 MB LLC using compression, the maximum effective capacity is increased up to 8 MB. This effect is, essentially, the same as increasing the number of ways without changing the number of sets, as shown in Figure 7(a). The y-axis shows the number of misses per thousand instructions (MPKI), while the x-axis shows the number of ways. This set-associativity sensitivity shows how much we can reduce the number of misses by increasing the effective capacity for an application. Figure 7(b) shows the compression ratios and uncompressed/compressed cacheline proportions for the 9 evaluated benchmarks. A lower compression ratio indicates higher compressibility. The benchmark `bodytrack` – which is the most compressible application – has over 40% of the total cache lines compressed within 2 segments (8 B). As previous compression studies have indicated, the main reason for this high compressibility is the large number of zero values. The 16-segment bars (64 B) indicate the incompressible cache line ratio; it accounts for 16% of `bodytrack`. It may appear better to evict a big-size cacheline to store a lot of small-size cachelines. However, merely evicting a big-size cacheline can increase the miss rate, because of possibly high locality, as described in Section 4. As indicated in Figure 3, the hit-rate changes for each cacheline size, and each cacheline size exhibits different locality. Usually, big-size cachelines also have high locality, but this conclusion is strictly application-dependent.

6.2. Dissecting the Attributes of the ECM Mechanism

In order to fully exploit the benefits of compression in the LLC, the goal of the proposed ECM technique is to maximize the effective cache capacity by considering *locality* and *size* information *simultaneously*. To achieve this goal, ECM gives big-size cachelines a higher probability of eviction and selects the largest-size cachelines that do not exhibit good locality as victims. Therefore, both big-size and small-size cacheline insertion points, and the threshold values used to classify new cachelines, are very important design parameters. In this subsection, we first analyze the insertion points for big-size and small-size cachelines, and then we demonstrate the performance of DATS, as compared to STS, for various threshold values.

6.2.1. Insertion Sensitivity Study for ECM: Figure 8 shows the sensitivity of ECM as we statically change the width of the M-bit register (holding the RRPV) on big-size/small-size cacheline insertion. The x-axis represents all 2^M possible RRPVs for big-size/small-size cacheline insertion with M=2 and 3, and this study is similar to the sensitivity study by Jaleel et al. [11]. The 'BS-INS=b' and 'SS-INS=s' labels (where 'b' and 's' are numbers) indicate ECM configurations where size-classified missing cachelines are inserted with an RRPV of 'b' for big-size (BS-INS=b) and 's' for small-size (SS-INS=s). The term 'D=d' (where 'd' is a number) indicates the distance of RRPV between 'b' and 's.' For each ECM configuration, the y-axis shows the maximum, average, and minimum values for the percent miss count reduction in the compressed LLC across all the applications.

As illustrated in Figure 8, we set a higher RRPV of 'b' than 's,' so that big-size cachelines have a higher probability of *eviction*. As we decrease the RRPV of 'b' and 's' (closer to the *near-immediate* re-reference interval), the maximum percent miss count reduction also diminishes, because cachelines that do not have temporal locality may pollute the compressed LLC with an extended travel time within the RRIP chain. Likewise, with a constant RRPV of 'b,' decreasing the RRPV of 's' shows a smaller percent miss count reduction, for the same reason. Regarding the average percent miss count reduction, the constant RRPV 'b' of $2^M - 2$ (*long* re-reference interval) shows the best performance, because the RRPV of $2^M - 1$ (*distant* re-reference interval) cannot provide enough time for high locality cachelines to be re-referenced. Therefore, the average percent miss count reduction is maximized when predicting a missing cacheline with an RRPV of $2^M - 2$ (*long*) for big-size cachelines, and $2^M - 3$ (*intermediate*) for small-size cachelines. For the rest of the paper, unless otherwise stated, we only provide results for ECM with the RRPV set to $2^M - 2$ for big-size cachelines and $2^M - 3$ for small-size cachelines, when M=3, D=1.

6.2.2. Evaluating the Performance of DATS: Figure 9 represents STS-best, STS-worst, and DATS values for the percent miss count reduction, as compared to DRRIP. The STS-best and STS-worst values are selected among the *static threshold value* T_h, where $2 \leq T_h < 16$, for each application. Note that the *static threshold value* – for the STS-best and STS-worst results – varies with each application. As Figure 9 indicates, DATS always exhibits better performance than STS-worst, and it is very close to the performance of STS-best. Specifically, DATS shows 32.9% average miss count reduction over STS-worst, and STS-best shows 8.3% average miss count reduction over DATS. Even though DATS does not exhibit the best overall performance, it clearly performs exceptionally well without requiring offline profiling of the workloads.

As previously mentioned in Section 4, DATS determines a new threshold on a per-set basis whenever a cacheline is inserted in a set. Thus, DATS will be especially useful

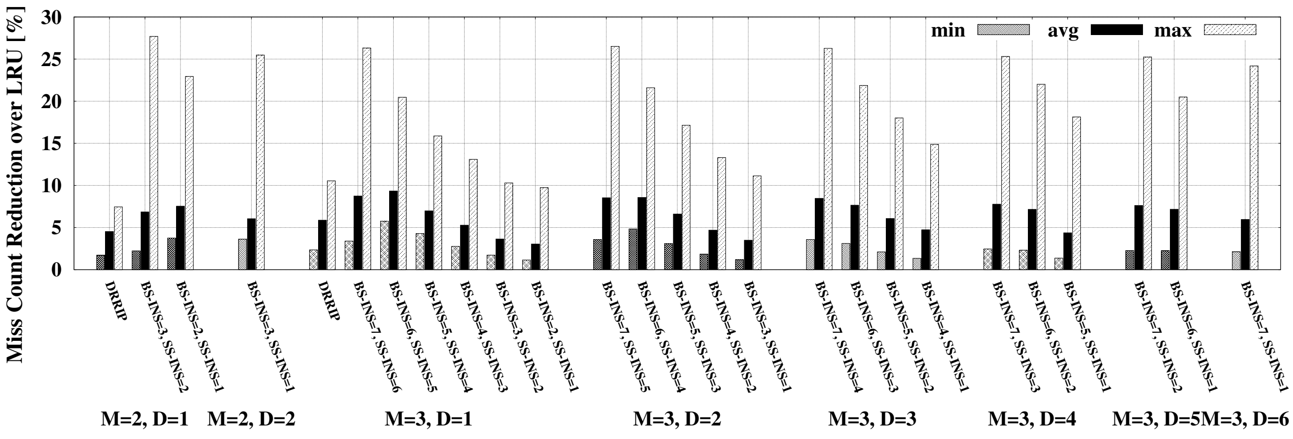


Figure 8. Insertion sensitivity study for the proposed Effective Capacity Maximizer (ECM) mechanism. [BS-INS: Big-Size cacheline INSertion; SS-INS: Small-Size cacheline INSertion; M=Size of RRPV value in number of bits; D=Distance in RRPV value between BS-INS=b and SS-INS=s, where 'b' and 's' are numbers]

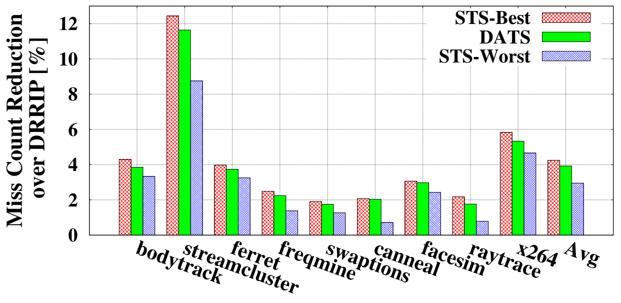


Figure 9. Performance comparison between the Dynamically Adjustable Threshold Scheme (DATS) and the Static Threshold Scheme (STS).

in environments running multi-programmed workloads, because a single *static threshold value* would not benefit all workloads. Furthermore, the optimal threshold value would fluctuate more over time, because a set may be accessed by multiple applications with disparate attributes. Hence, it is reasonable to expect DATS to prove even more instrumental and effective under multi-programmed workloads. Given the conclusions extracted from this experiment, we henceforth provide results only for SAI (insertion) and ECM with DATS.

6.3. Assessing the ECM Cache Management Scheme

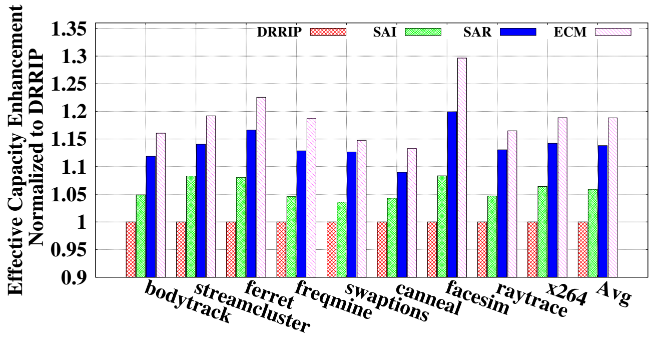
Figure 10 summarizes the compressed LLC performance results of 3-bit (i.e., 3-bit RRPVs) SAI, SAR, and ECM setups, normalized to the performance of DRRIP, for all evaluated applications. The SAR scheme is implemented on DRRIP. Note that **ECM combines the SAI and SAR schemes in a unified framework**. The results for SAI and SAR refer to systems that use said techniques in *isolation*. Figure 10(a) shows the effective capacity enhancement. SAR shows higher effective capacity enhancement than SAI. This is because SAI gives big-size cachelines a higher probability of eviction, while SAR evicts the biggest-size cacheline first (i.e., SAR is more oriented toward size information). ECM shows the best effective capacity enhancement, because it gives higher priority to big-size cachelines using SAI, and it evicts the biggest cacheline first using SAR. SAI, SAR, and ECM show 5.9%, 13.8%, and 18.8% average effective capacity enhancement, respectively.

The effective capacity enhancement in an application is not always proportional to the compression ratio, because big-size cachelines may stay longer in the cache (if they have higher locality) than small-size cachelines. Therefore, when we compare the proposed ECM mechanism's effective capacity enhancement to DRRIP, it is important to note that the achieved enhancement depends on the hit-rate changes of each cacheline size, as depicted in Figure 3, and on the cacheline size proportions, as shown in Figure 7. The `facesim` benchmark shows the best effective capacity enhancement, 29.6%, because `facesim` exhibits high locality on 8 B and 48 B size cachelines (which are abundant), and low locality on the incompressible (biggest-size) cachelines.

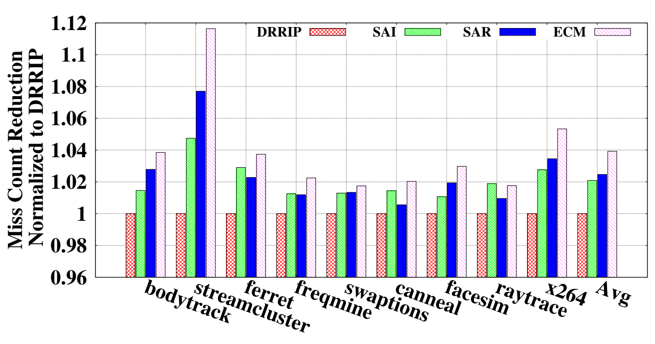
Figure 10(b) shows the miss count reduction. Even though the effective capacity enhancement of SAI is less than SAR across all the applications, some applications show better miss count reduction – such as `ferret`, `canneal`, and `raytrace` – because SAI considers locality and size information at the same time, while SAR considers only size information. ECM shows the best performance across all applications with 3.9% average miss count reduction, while SAI and SAR show 2.1% and 2.5% reductions, respectively.

Figure 10(c) shows the reduction in write-back delay cycles. SAI shows 2.9% average write-back delay cycle reduction, because SAI reduces the off-chip memory write requests by not only reducing the number of misses, but also by giving higher probability of eviction to big-size cachelines. This reduces the probability of small-size cachelines (which are numerous) to be evicted. This probability is further reduced by SAR, which selects the largest-size cacheline within the victim pool, so that the off-chip memory write requests are minimized, as explained in Section 3. SAR shows 10.8% average write-back delay cycle reduction, while ECM shows 12.9% reduction ***by exploiting both SAI and SAR***.

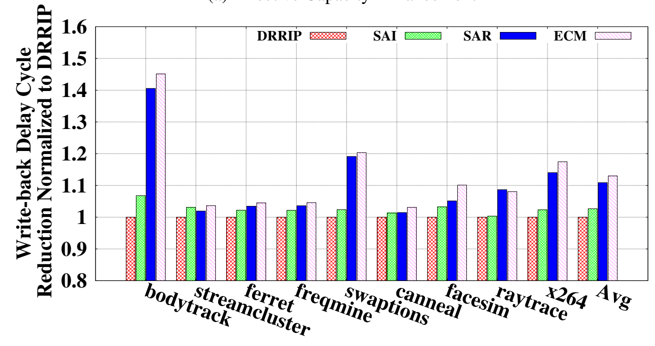
To sum up, by considering locality and size information at the same time, ECM shows an increase in the effective cache capacity, a decrease in miss count, and a reduction in the write-back delay cycles. Most importantly, the in-



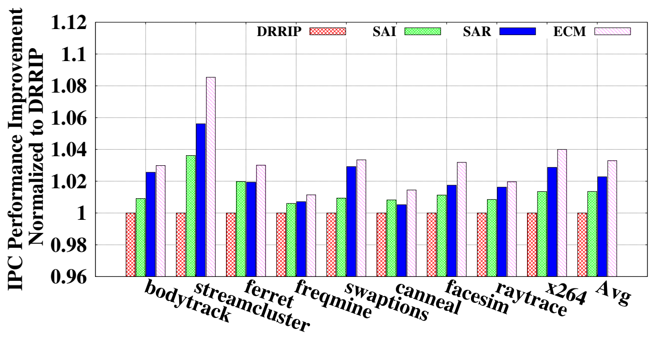
(a) Effective Capacity Enhancement



(b) Cache Performance Improvement



(c) Write-back Delay Cycle Reduction



(d) System Performance Improvement

Figure 10. The overall performance results of the proposed ECM mechanism (combined SAI+SAR), as compared to DRRIP, SAI (alone), and SAR (alone).

creased effective capacity allows further reduction in the number of misses, which allows ECM to outperform DRRIP by reducing the miss penalty. Overall, SAI, SAR, and ECM outperform DRRIP by an average of 1.4%, 2.3%, and 3.3% in Instructions-Per-Cycle (IPC) performance, respectively, as indicated in Figure 10(d).

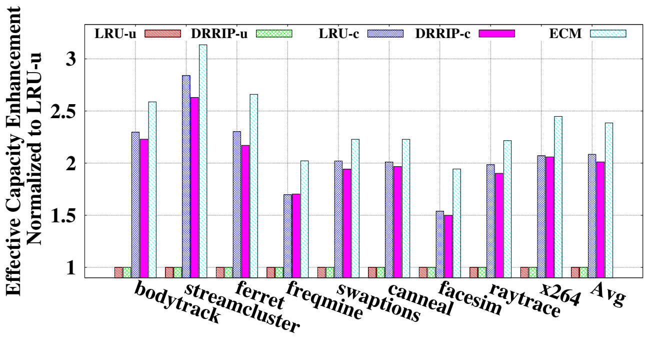
6.4. Performance Comparison with Existing Techniques

In this sub-section, we compare the proposed ECM mechanism with LRU without compression (LRU-u), LRU with compression (LRU-c), DRRIP without compression (DRRIP-u), and DRRIP with compression (DRRIP-c). Figure 11(a) shows the effective capacity normalized to the LRU-u, which indicates a 2 MB uncompressed LLC. When we apply LRU-c, the average effective capacity enhancement is 108.5%, while DRRIP-c achieves a 101.2% enhancement. The proposed ECM technique improves the effective capacity by an average of 138.6%, at around 4.77 MB, as compared to the uncompressed LLC. The proposed ECM mechanism shows an average effective capacity increase of 15% over LRU-c.

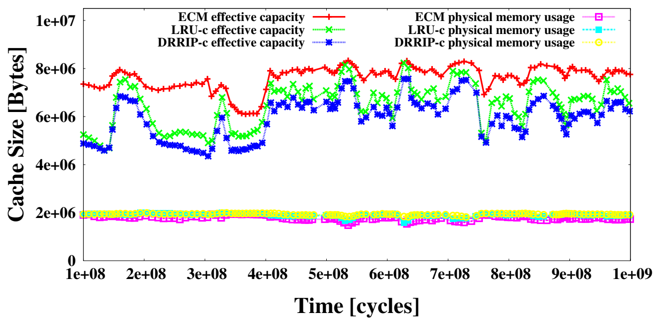
DRRIP-c, which only considers locality and does not consider the compressed cacheline size information, shows less effective capacity enhancement than LRU-c across all the applications. Figure 11(b) shows the *effective capacity* fluctuation with *physical memory usage* of `streamcluster`, which experiences the least effective capacity enhancement under DRRIP-c, as opposed to LRU-c, at around -7.3%. During 0.9 billion cycles, DRRIP-c shows lower effective capacity than LRU-c, but higher physical memory usage. This indicates that big-size cachelines are frequently re-referenced while residing in the compressed LLC, and they occupy more physical area than small-size cachelines. In fact, big-size cachelines in `streamcluster` exhibit higher hit rates than small-size cachelines, as indicated in Figure 3. In sharp contrast, ECM maximizes the effective capacity uniformly throughout the timeline by considering locality and size information at the same time.

Figure 11(c) shows the miss count reduction, as compared to LRU-u. The `swaption` and `raytrace` benchmarks are ranked first and second in terms of miss count reduction, with 77.6% and 64.9% decreases, respectively, over LRU-u. This is because they are the most set-associativity-sensitive applications, as indicated in Figure 7(a). Thus, they benefit the most from enhancements in the effective cache capacity. As we observed in Figure 11(a), DRRIP-c yields lower effective capacity enhancement than LRU. Despite this attribute, DRRIP-c achieves higher miss count reduction than LRU-c: DRRIP-c exhibits a 37.9% average miss count reduction over LRU-u, while LRU-c shows a 31.1% reduction. The proposed ECM mechanism shows an average miss count reduction of 43.4% over LRU-u and 9.4% over LRU-c.

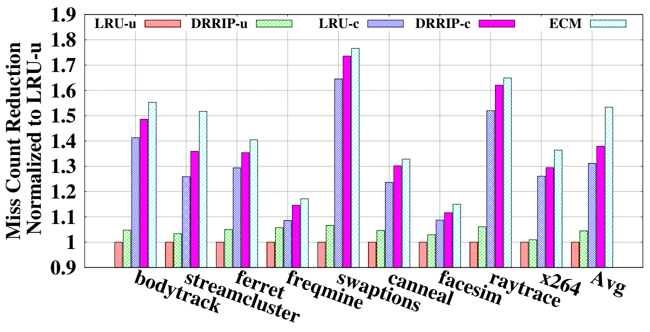
Figure 11(d) shows the IPC performance normalized to LRU-u. The `swaption` benchmark shows the best IPC performance improvement at 41.7%, because of a large miss count reduction over LRU-u. The `streamcluster` benchmark is ranked second, with a 39.9% IPC performance improvement. The `facesim` and `freqmine` benchmarks show IPC performance *degradation* when we use LRU-c. On the contrary, by using ECM, `facesim` can actually exploit compression and it shows a small 3.2% IPC performance improvement. However, `freqmine` shows a 3.1% IPC performance degradation even under ECM. These problematic benchmarks are applications that do not benefit from compression and suffer from excessive decompression overhead. For such applications, an adaptive cache compression technique may be used, like the one proposed by



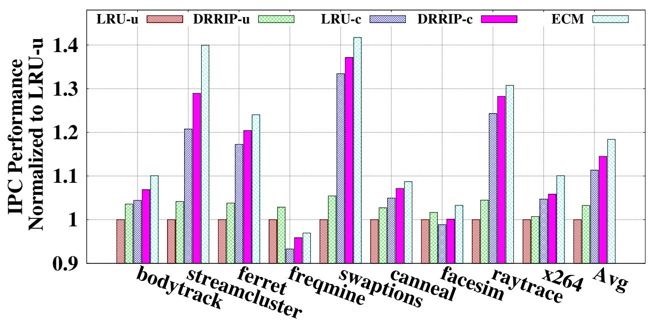
(a) Effective Capacity



(b) Memory Usage of the `streamcluster` Benchmark



(c) Miss Count Reduction



(d) System Performance

Figure 11. Comparing ECM with existing techniques.

Alameldeen et al. [3]. Regardless, over all benchmarks, the proposed ECM scheme shows an 18.4% average IPC improvement over LRU-u, while LRU-c shows an 11.3% average IPC improvement. The proposed ECM mechanism shows a 6.2% average IPC improvement over LRU-c.

6.5. The Performance Sensitivity of ECM to Cache Size and Logical Set Associativity

Figure 12(a) shows the IPC performance under ECM, normalized to LLCs (with LRU-u) of different sizes: 1 MB, 2 MB, 4 MB, 8 MB, and 16 MB. As always, all LLCs are logically 16-way overlaid on top of a physically 4-way cache. Therefore, each cache's effective capacity size increases up to 4 MB, 8 MB, 16 MB, 32 MB, and 64 MB, respectively, when we apply a compression technique. The y-axis shows the average IPC performance normalized to the uncompressed LLC of the respective LLC size, under LRU replacement. ECM outperforms LRU-c by 4.2-6.4% for various cache sizes. Moreover, ECM shows the best results – 23.6% IPC performance improvement – when compared to an 8 MB LLC with LRU-u.

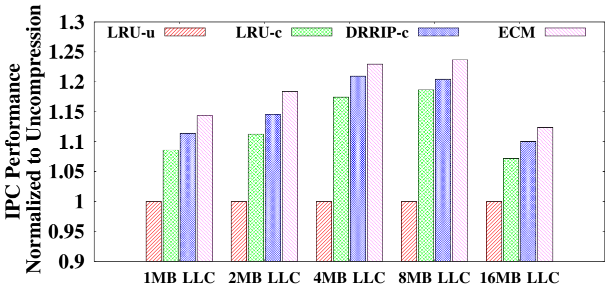
We also conducted an ECM sensitivity study for different *logical* way sizes: 8-way, 12-way, 16-way, and 20-way on a 2 MB physically 4-way cache. Therefore, the effective capacity increases up to 4 MB, 6 MB, 8 MB, and 10 MB, respectively. The results are shown in Figure 12(b). The y-axis shows the average IPC performance normalized to LRU-u for each respective logical-way size. ECM outperforms LRU-c by 4.1-7.8% for various logical-way sizes. If we increase the number of logical ways, we can increase the ECM improvement over LRU. Clearly, these results show that ECM is scalable with both the cache size and the number of logical ways.

7. Conclusion

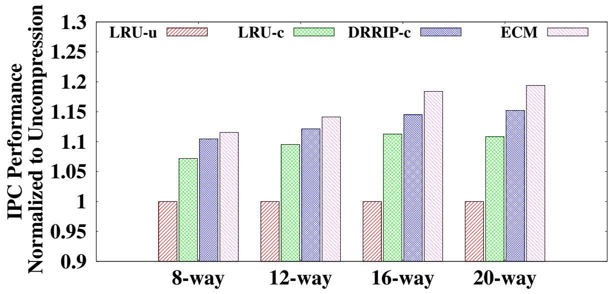
Rapidly escalating transistor integration densities have accentuated the perennial divergence between logic and memory performance. The "memory wall" phenomenon hinders the performance gains that may be reaped from the abundance of on-chip computational resources. Consequently, the design of the memory hierarchy, and specifically the Last-Level Cache (LLC), has garnered special attention from computer architects over the last several years. Despite the steady increase in the size of on-chip LLCs, the latter can still benefit from even larger capacities. An efficient way to increase the *effective* cache capacity without increasing the *physical* size is to compress the LLC. In compressed caches, the cacheline size becomes variable and depends on the achieved compression ratio.

In this work, we demonstrate that the size information of each compressed cacheline can be very helpful when trying to choose an eviction victim. However, there are currently no replacement policies tailored to compressed LLCs, which can make use of such size information. This paper aims to maximize the performance of compressed LLCs through the use of a size-aware cache management policy. The proposed Effective Capacity Maximizer (ECM) mechanism employs a triptych of policies: (1) Size-Aware Insertion (SAI), (2) a Dynamically Adjustable Threshold Scheme (DATS) to classify cachelines, and (3) Size-Aware Replacement (SAR).

ECM's operation adjusts the eviction criteria based on the compressed data size at any given time. This dynamic scheme is shown to substantially increase the effective cache capacity and minimize the miss penalty. Through the use of extensive simulations with memory traces extracted from real application workloads running in a full-system simulation environment, ECM is demonstrated to achieve significant improvement over the LRU and DRRIP



(a) Sensitivity to the LLC Size with 16 Logical Ways



(b) Sensitivity to the Logical Ways with a 2 MB LLC

Figure 12. The performance sensitivity of ECM to cache size and logical set associativity.

replacement policies. Specifically, ECM exhibits an average effective capacity increase of 15% over LRU and 18.8% over DRRIP, and an average cache miss reduction of 9.4% over LRU and 3.9% over DRRIP. These gains combine to yield 6.2% and 3.3% average system performance improvement over LRU and DRRIP, respectively.

8. Acknowledgments

This work was partially supported by the ETRI R&D program of KCC(Korea Communications Commission), Korea[11921-03001, "Development of Beyond Smart TV Technology"]. It also falls under the Cyprus Research Promotion Foundation's Grant ΤΠΕ/ΠΛΗΡΟ/0609(BIE)/09 (co-funded by the Republic of Cyprus and the European Regional Development Fund).

References

[1] International Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2010.

[2] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia. Compression in cache design. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 190–201, New York, NY, USA, 2007. ACM.

[3] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 212–, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. In *Technical Report 1500*, University of Wisconsin-Madison, 2004. Computer Sciences Department.

[5] R. Bagrodia and *et al.* Parsec: A parallel simulation environment for complex systems. *Computer*, 31:77–85, October 1998.

[6] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(8):1196 –1208, aug. 2010.

[7] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proceedings of the Conference on Data Compression*, DCC '96, pages 200–, Washington, DC, USA, 1996. IEEE Computer Society.

[8] E. Hallnor and S. Reinhardt. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 201 – 212, feb. 2005.

[9] E. G. Hallnor and S. K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, WMPI '04, pages 9–15, New York, NY, USA, 2004. ACM.

[10] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[11] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[12] S. Kim, J. Lee, J. Kim, and S. Hong. Residue cache: a low-energy low-area l2 cache architecture via compression and partial hits. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 420–429, New York, NY, USA, 2011. ACM.

[13] J.-S. Lee, W.-K. Hong, and S.-D. Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *J. Syst. Archit.*, 46(15):1365–1382, Dec. 2000.

[14] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[15] P. S. Magnusson and *et al.* Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33:2005, 2005.

[17] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[18] L. Villa, M. Zhang, and K. Asanović. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 214–220, New York, NY, USA, 2000. ACM.

[19] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. Ship: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 430–441, New York, NY, USA, 2011. ACM.

[20] Y. Xie and G. Loh. Thread-aware dynamic shared cache compression in multi-core processors. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 135 –141, oct. 2011.

[21] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 258–265, New York, NY, USA, 2000. ACM.