Futility Scaling: High-Associativity Cache Partitioning

Ruisheng Wang Lizhong Chen
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA
ruishenw,lizhongc@usc.edu

***Abstract*—As shared last level caches are widely used in many-core CMPs to boost system performance, partitioning a large shared cache among multiple concurrently running applications becomes increasingly important in order to reduce destructive interference. However, while recent works start to show the promise of using replacement-based partitioning schemes, such existing schemes either suffer from severe associativity degradation when the number of partitions is high, or lack the ability to precisely partition the whole cache which leads to decreased resource efficiency.**

In this paper, we propose Futility Scaling (FS), a novel replacement-based cache partitioning scheme that can precisely partition the whole cache while still maintain high associativity even with a large number of partitions. The futility of a cache line represents the uselessness of the line to application performance and can be ranked in different ways by various policies, e.g., LRU and LFU. The idea of FS is to control the size of a partition by properly scaling the futility of its cache lines. We study the properties of FS on both associativity and sizing in an analytical framework, and we present a feedback-based implementation of FS that incurs little overhead in practice. Simulation results show that FS improves performance over previously proposed Vantage and PriSM by up to 6.0% and 13.7%, respectively.

I. Introduction

In contemporary chip multiprocessors (CMPs), large shared last-level caches (LLCs) are often used to close the performance gap between the need of processors for fast data access and the high latency of main memory. While sharing LLCs among multiple concurrently executing threads is generally beneficial for increasing resource utilization, unregulated sharing can compromise performance isolation and thus violate per-thread QoS requirements. In order to avoid potentially destructive interference induced by uncontrolled sharing, cache capacity partitioning has been proposed for LLC management [1–10].

A complete solution for partitioning shared cache capacity consists of an allocation policy that decides the size of each partition in order to satisfy specified QoS objectives [6, 7] and an enforcement scheme that decides how to actually enforce those sizes [9, 10]. While various allocation policies have been studied extensively [1–8], it is still challenging to design an efficient and effective enforcement scheme, particularly for supporting hundreds or thousands of fine-grained partitions. In this work, we focus on designing a cache partitioning enforcement scheme.

An ideal cache partitioning enforcement scheme should support (1) *smooth resizing:* a partition can be expanded/shrunk smoothly without incurring large overhead (i.e., no data flushing or migrating); (2) *precise sizing:* each partition should occupy the *exact* (i.e., no less and no more than) amount of cache space allocated to it; and (3) *high associativity:* the associativity of a partition should not be reduced as the number of partitions increases.

Generally, there are two approaches to enforce the partitioning of a cache: partitioning by constraining cache line placement and partitioning by controlling cache line replacement. Placement-based partitioning schemes (e.g., way-partitioning [11], reconfigurable caches [12] and molecular caches [13]) are unable to resize smoothly due to the inherent resizing penalty caused by line migration. Recently, replacement-based partitioning schemes are gaining increasing attention because they incur little resizing penalty and can scale to CMPs with a large number of fine-grain partitions. However, the existing replacement-based partitioning schemes either suffer diminishing cache associativity as the number of partitions increases (e.g., CQVP [4] and PriSM [10]) or cannot precisely partition the whole cache (e.g., Vantage [9]).

In this paper, we propose *Futility Scaling (FS)*, a replacement-based partitioning scheme that can precisely partition the whole cache while still maintain high associativity even with a large number of partitions. The futility of a cache line represents how useless a line is to its application's performance and can be ranked in different ways by various policies (e.g., LRU, LFU or OPT [14]). By scaling the futility of cache lines belonging to different partitions, FS can adjust the evictions of different partitions and thereby control the sizes of those partitions. By always evicting cache lines with the largest scaled futility from a *full* list of candidates, FS can retain a large amount of useful lines and hence, maintain high associativity.

Our contributions are summarized as follows:

1) We study the associativity degradation problem in a replacement-based partitioning scheme for the first time. A Partitioning-First (PF) scheme is used as a baseline to show how seriously associativity can be degraded in a replacement-based partitioning scheme as the number of partitions increases (Section III).
2) We propose a novel *Futility Scaling (FS)* scheme and

studied its properties on both associativity and sizing in an analytical framework. Our analysis shows that the associativity of each partition in FS is independent of the number of partitions and the actual size of each partition in FS is statistically very close to its target size (Section IV).

3) We propose a practical feedback-based design for implementing FS. Our design largely maintains the analytical properties of FS while incurring little storage overhead, i.e., five registers per partition besides the 1.5% overall state overhead for implementing previously proposed coarse-grain timestamp-based LRU [9] (Section V).
4) We evaluate the proposed FS scheme on a QoS-enabled 32-core CMP. Simulation results show that, due to its properties of both high associativity and precise sizing, FS improves performance over previously proposed Vantage [9] and PriSM [10] by up to 6.0% and 13.7%, respectively (Section VIII).

II. Background

A. Allocation Policy and Enforcement Scheme

Generally, cache capacity management has two components: an allocation policy and an enforcement scheme. An allocation policy (often implemented in software) translates high-level QoS objectives into cache capacity resource assignments. An enforcement scheme (often implemented in hardware) controls the actual low-level fine-grained hardware resource to guarantee that each thread occupies the amount of cache space assigned by the allocation policy [7, 15]. Many cache capacity allocation policies have been proposed, including "Utilitarian" policies [2] that maximize throughput [3], "Communist" policies [2] that improve fairness [1, 8], and "Elitist" policies [5] that guarantee QoS requirements for high-priority threads [6, 7]. However, the efficacy of those allocation policies is largely determined by the underlying cache partitioning enforcement schemes. For example, even when a thread is allocated the exact amount of space in a shared context as it has used in a private context, the performance of this thread could still decrease, either because the partitioning enforcement scheme cannot actually guarantee this thread to occupy the exact amount of assigned cache space, or the associativity of the partition for this thread is largely reduced.

Ideally, a cache partitioning enforcement scheme should have the following three properties:

1) *Smooth Resizing*: This means that the size of each partition can be dynamically adjusted with little performance penalty, i.e., no cache data flushing or line migration. Smooth resizing enables high flexibility for an allocation policy to change cache assignments according to applications' dynamic behaviors in short intervals.
2) *Precise Sizing:* This indicates that the actual size of each partition should be close to its target size and the whole cache can be managed at fine-granularity. Precise sizing allows an allocation policy to precisely control cache capacity resource so that it can make more efficient assignments of limited on-chip cache capacity resource.
3) *High Associativity*: This refers to that each partition should have high associativity even on a very large-scale CMP (i.e., having hundreds or thousands of partitions). High associativity improves the utility of cache capacity and can yield significant performance improvement for associativity-sensitive applications.

The objective of this work is to design a cache partitioning enforcement scheme that maintains all three above mentioned properties.

B. Placement- and Replacement-based Partitioning

Broadly, cache partitioning enforcement schemes can be categorized into two groups: placement-based partitioning and replacement-based partitioning. Placement-based partitioning schemes [11, 13, 16] partition a cache by placing the cache lines from different partitions into disjoint physical cache regions. For example, way-partitioning or column caching [11] statically assigns physical cache ways to each partition. Although way-partitioning is straightforward to implement, it cannot support fine-grained partitioning, and cache associativity reduces rapidly as the number of partitions increases. To address these problems, reconfigurable caches [12] and molecular caches [13] are proposed to partition caches by sets instead of ways. Page Coloring [16] is another placement-based partitioning that maps the physical pages of different applications to different cache sets. However, all these placement-based partitioning schemes have a common problem: there is a large overhead to resize partitions, i.e., cache data has to be flushed or moved when a partition changes its size.

Replacement-based partitioning schemes [4, 9, 10] partition a cache by adjusting the line eviction rate of each partition at replacement. For instance, Cache Quota Violation Prohibition (CQVP [4]) sets the quota for each partition and always chooses the cache lines from the partition that exceeds its quota to evict. Similarly, Probabilistic Shared-cache Management (PriSM [10]) controls the partitioning by adjusting the eviction probability of each partition based on its insertion rate and size deviation from its target. Vantage [9] stabilizes the size of each partition via controlling its "aperture", where a larger "aperture" incurs a higher eviction rate. In general, if a cache controller evicts lines from a partition at a rate that is higher (lower) than its insertion rate, the size of the partition will shrink (expand). Since partition sizes are changing in the process of replacement, they can be controlled at the granularity of a single line and there is little cost for resizing. However, the existing replacement-based partitioning schemes either suffer from associativity degradation as the number of partitions increases [4, 10], or cannot precisely partition the whole cache [9, 10]. In this

paper, we present a replacement-based partitioning scheme that achieves both high associativity and precise sizing.

III. Partitioning-induced Associativity Loss

A. Cache Model and Definitions

To facilitate the analysis of our proposed design throughout this paper, we use a similar cache model described in [9, 17] but extend it with the notion of cache futility ranking. A cache consists of the following three components.

- *Cache Array*: This implements associative lookups and provides a list of replacement candidates on each eviction.
- *Futility Ranking*: This maintains a strict total order of the uselessness of cache lines within each partition.
- *Replacement Policy*: This identifies the victim from the list of replacement candidates based on their futility and partitioning requirements.

The cache array could be a common set-associative cache, skew-associative cache [18] or zcache [17]. It provides a list of replacement candidates on each eviction. The *futility* of a cache line is used to assess how useless keeping this line in the cache would be. For a partitioned cache, the uselessness of cache lines within each partition is strictly ordered by a specific futility ranking scheme. For example, in LRU, LFU and OPT [14] futility ranking schemes, cache lines are ranked by the time of their last accesses, their access frequencies, and the time to their next references, respectively. A cache line with a higher rank is less useful. To make the rest of the analysis independent of cache size, we define a line's *futility* to be its rank normalized to $[0,1]$, i.e., for a cache line ranked as the rth place in a partition with a size of M lines, its *futility* is $f = r/M$, $f \in [0,1]$. Given the total number of cache lines is very large, we assume the futility of cache lines is *continuously* and uniformly distributed over the range of $[0,1]$ in the following analysis.

Conventionally, cache associativity is considered as the number of ways for a set associative cache. A set associative cache with a larger number of ways has a better ability to evict lines that are less useful. Previous work [17] has generalized the concept of associativity based on the insight that associativity does not rely on the number of allowed block locations but instead, on the number of replacement candidates available on an eviction. In this paper, in order to compare associativity across different partitioning schemes regardless of cache organizations, futility ranking schemes, and actual workloads, we follow the universal associativity concept descried in [17]. The cache *associativity* is considered as *the ability of a cache to evict a useless line on a replacement*, i.e., the ability to retain useful lines in the cache. *The less useful the evicted lines are, the higher the associativity.*

B. Partitioning-induced Associativity Loss

Generally, cache associativity is subject to two factors: (1) the "quality" (i.e., the futility) of replacement candidates that a cache can provide and (2) the ability of a cache to evict a "high-quality" line (i.e., the line with large futility) from the list of replacement candidates. There are two common ways to improve the "quality" of replacement candidates. One is to use good hash functions to index the cache [19, 20] in order to spread out accesses and, thus, avoid the worst case that the "quality" of all replacement candidates are low (i.e., the futility of all candidates is small). The other is to increase the number of replacement candidates, e.g., add more ways or use zcache [17]. The more replacement candidates a cache can provide, the higher chance for the cache to include a "high-quality" line in its candidate list. However, improving the "quality" of replacement candidates does not necessarily increase associativity if the replacement policy cannot choose a "high-quality" candidate to evict. In a non-partitioning cache, the replacement policy is always able to choose the least useful candidate (i.e., the candidate with best "quality") to evict. However, this is not true when a cache needs to be partitioned.

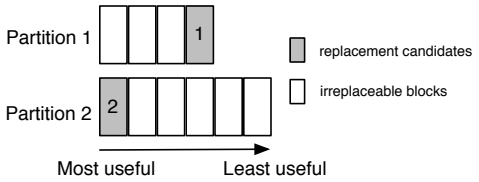


Figure 1: Associativity and sizing dilemma in a replacement-based partitioning scheme

In a shared cache with replacement-based partitioning, a replacement policy needs to perform two roles: to improve the associativity within a partition and to maintain the size ratios among partitions. The first role requires the replacement policy to evict the most useless of candidates as possible, which is the same requirement for the replacement policy in a non-partitioning context. The second role requires the replacement policy to prioritize cache replacement candidates from the oversized partitions over those from undersized ones in victim selection, which is not addressed in a non-partitioned cache. These two criteria often conflict with each other.

For a list of replacement candidates provided on an eviction, the least useful candidate may not come from an oversized partition. For example, as is shown in Figure 1, assume there is a cache with ten cache lines in total. The cache needs to be partitioned equally, i.e., each partition should have five cache lines. However, the current sizes of the two partitions are four and six, respectively. Assume an incoming line that belongs to the second partition invokes the replacement process, and the cache array provides two replacement candidates: one is the least useful line in the first partition, and the other is the most useful line in the second partition. In this scenario, a dilemma arises. On the one hand, from the perspective of improving associativity, the first candidate should be evicted since it is less useful. However, this would lead to the sizes of both partitions straying even

Table I: Notations used in this paper

Term	Meaning
N	Total number of partitions
S_i	Fraction of cache space for Partition i, $\sum_{i=1}^{N} S_i = 1$
I_i	Fraction of total misses/insertions from Partition i , $\sum_{i=1}^{N} I_i = 1$
E_i	Fraction of total evictions from Partition i, $\sum_{i=1}^{N} E_i = 1$
N_i^A	Actual number of cache lines of Partition i
N_i^T	Target number of cache lines of Partition i
R	Number of replacement candidates on an eviction

farther away from their target sizes. On the other hand, from the perspective of enforcing the desired size ratio, the second candidate should be evicted since it belongs to the oversized partition. However, evicting the most useful cache line would inevitably hurt the associativity of the second partition. Therefore, how to enforce the partitioning while still largely maintaining the associativity of each partition becomes a challenging problem.

C. Partitioning-First Scheme

To further illustrate the previously described associativity degradation problem, we analyze a partitioning-first scheme.

Some of the common terms used throughout this paper are defined below (also listed in Table I). N is the total number of partitions. S_i is the fraction of total cache space for Partition i. I_i and E_i represent the fractions of total insertions and evictions that belong to Partition i, respectively. R is the number of candidates provided for each replacement. N_i^A and N_i^T denote the actual and target number of cache lines for Partition i, respectively.

Algorithm 1 represents a Partitioning-First (PF) scheme which focuses on enforcing partitioning rather than improving associativity. This approach always evict lines from the partition that exceeds its target size most. The algorithm has two steps: Partition Selection (PS) and Victim Identification (VI). In the PS step, the algorithm chooses the partition whose current actual size (N_i^A) most exceeds its target size (N_i^T) among all the candidates' partitions. Then in the VI step, it evicts the line with the largest futility among the candidates belonging to the partition chosen by the PS step.

This algorithm first does its best effort to ensure the size of a partition as close to its target as possible in the PS step, and then in the VI step, it aims to achieve best possible associativity by evicting the most useless cache line from a *reduced* list of candidates.

Figure 2 compares the PF scheme with different number of partitions (N = 1, 2, 4, 8, 16, 32) in a 16-way set associative cache. The cache is equally partitioned with the size of $512kB$ for each partition. Each workload for an N-partition configuration is constructed by duplicating a SPEC CPU2006 benchmark N times. To ignore any undesirable properties of a particular futility ranking scheme (e.g., increasing associativity decreases performance), we use the OPT [14] scheme to rank the futility of each line. More detailed information about system configuration is present in

Algorithm 1 Partitioning-First Scheme

Require: The ith candidate is from partition p_i and has a futility of f_i, $1 \leq i \leq R$, $1 \leq p_i \leq N$, $0 \leq f_i \leq 1$

Step 1: Partition Selection (PS)

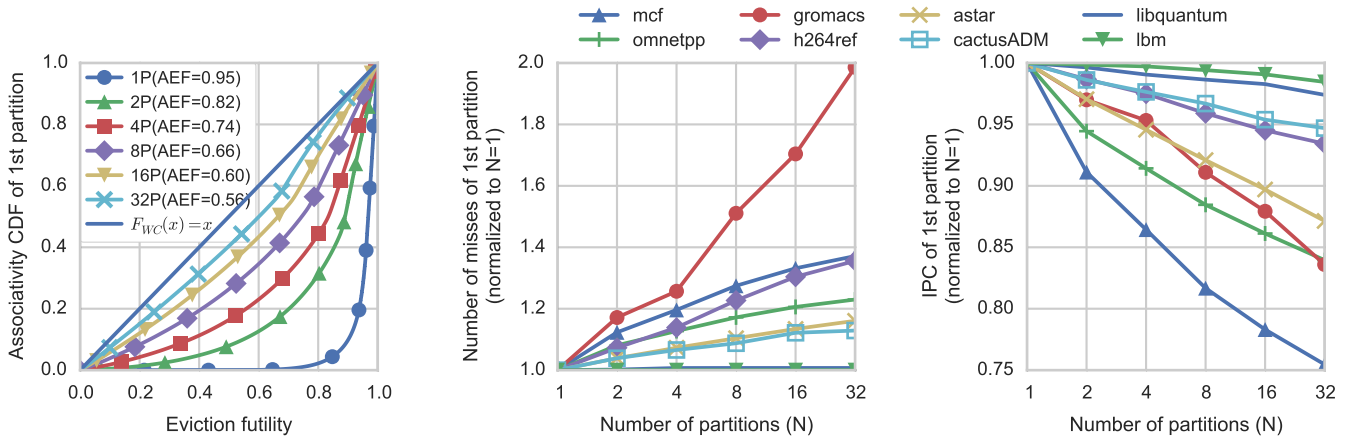
```
max_over ← −∞
chosen_partiton ← none
for i ← 1 to R do
    if max_over < (N^A_{p_i} − N^T_{p_i}) then
        max_over ← (N^A_{p_i} − N^T_{p_i})
        chosen_partition ← p_i
```

Step 2: Victim Identification (VI)

```
max_futility ← −∞
chosen_victim ← none
for i ← 1 to R do
    if p_i = chosen_partition then
        if max_futility < f_i then
            max_futility ← f_i
            chosen_victim ← i
return chosen_victim
```

Section VII. We only report the result of the first partition in each workload. The result for the rest of partitions in the workload are similar since they are homogeneous, i.e., having the same size and occupied by the same benchmark.

To assess the quality of associativity, we use an *Associativity Distribution* as defined in [17], which is the probability distribution of evicted lines' futility. For a fully-associative cache, it always chooses to evict the line with the largest futility, $f_{evict}^{FA} = 1.0$. In general, *for a partitioning scheme, the more skewed the distribution of a partition is towards* $f_{evict} = 1.0$*, the higher the associativity*. Figure 2a shows cumulative associativity distributions of the first partition for the workloads constructed by the *mcf* benchmark. From the figure, we can see that, when there is only one partition, its associativity is high, i.e., its average eviction futility (AEF) reaches 0.95. However, as the number of partitions increases, its associativity decreases (i.e., the associativity CDF curve is skewed farther away from $f_{evict} = 1.0$). This is because, in the PF scheme, as the number of partitions increases, the list of replacement candidates available for the VI step is shortened by the PS step. As the number of replacement candidates decreases, the probability of this reduced candidate list including a high-futility line becomes lower, which leads to a lower average eviction futility and, thus, worse associativity. In the worst case ($N \gg R$), there is always only one cache line in the candidate list that belongs to the chosen partition. The VI step has no option other than to evict this line regardless of its futility (this is very similar to a direct-mapped cache that can only provide one replacement candidate on an eviction). In such case, the futility of evicted lines becomes random, and thus the associativity CDF curve becomes a straight diagonal line $F_{WC}(x) = x$ ($AEF = 0.5$). This results in the futility ranking becoming irrelevant as cache lines with different futility have the same probability of being evicted. As shown in Figure 2a, when the number of partitions (N) reaches 32, the associativity CDF ($AEF = 0.56$) of the first partition is very close to the worst case. Figure 2b and 2c show the number of misses and IPCs, respectively, of the first



(a) Associativity CDF for *mcf*

(b) Number of misses of 8 benchmarks

(c) IPCs of 8 benchmarks with PF scheme

Figure 2: Comparisons of the PF scheme in a cache with different number of partitions ($N = 1, 2, 4, 8, 16, 32$)

application in each workload under the PF scheme. All the results are normalized to the results of the workloads with $N = 1$. From the figures, we can see that, due to the degradation of associativity as the number of partitions increases, the number of misses of each application increases and its IPC decreases correspondingly. Different applications have different sensitivities to associativity, e.g., associativity degradation has negligible effect on *lbm*'s performance since it has a very low rate of cache reuses but *mcf* has more than 37% increase in cache misses and correspondingly 24% drop in IPC decrease as the number of partitions goes from 1 to 32. Hence, the PF scheme is not scalable for large-scale CMPs due to its associativity degradation.

IV. Futility Scaling

A. Overview

The basic idea of *Futility Scaling* (FS) is to control the size of each partition by scaling the futility of its cache lines. FS works as follows. Each partition has a scaling factor of α_i. On each eviction, the futility of a replacement candidate (f_{cand}) belonging to Partition i will be scaled by α_i, i.e., $f_{cand}^{scaled} = f_{cand} \times \alpha_i$ and the candidate with the largest scaled futility will be evicted. By scaling up/down the futility of lines in a partition, cache lines belonging to this partition will be evaluated as less/more useful in the view of the whole cache. Since FS always chooses the least useful replacement candidate (i.e., the candidate with the largest scaled futility) to evict, the more useless lines a partition has, the fewer number of lines belonging to this partition will be kept in the cache. Therefore, by increasing or decreasing the scaling factor of a partition, FS can shrink or expand its size correspondingly.

In the rest of this section, we study the associativity and sizing properties of the FS scheme with two partitions in an analytical framework proposed in [17]. The properties of FS in a more general scenario (i.e., more than two partitions) are summarized at the end of the section. The framework is based on two assumptions below.

1) (*Uniformity Assumption*) *On each eviction, the replacement candidates are independent and uniformly distributed.*
   While this assumption is not strictly true, it is statistically close enough in a practical cache indexed by good random hash functions [17].
2) *On each eviction, the exact futility of each candidate is given.*
   Although tracking exact futility would be very expensive, we present a practical feedback-based futility scaling design without knowledge of exact futility in Section IV.

Note that even though the quantitative results derived from this analytical framework are not accurate for real caches due to the above two assumptions, the qualitative conclusions drawn from our analysis provide some guiding principles in designing FS, especially for a real cache with good hash indexing, e.g., set associative cache with H3 hashing, skewed associative cache and zcache [17].

B. Calculation of Scaling factors

Assume a cache with R replacement candidates has two partitions with insertion rates of I_1 and I_2, and target size fractions of S_1 and S_2, respectively. Without losing generality, we assume that $I_1 < S_1$ and thus $I_2 > S_2$. Naturally (i.e., without scaling), a partition's size fraction tends to be proportional to its insertion rate. In order to reduce the size of Partition 2 from I_2 to S_2, we scale up the futility of cache lines belonging to Partition 2 by a factor of α_2 ($\alpha_2 > 1$) and keep Partition 1 unscaled, i.e., $\alpha_1 = 1$.

Based on the analytical framework and the condition for a stable partitioning (i.e., the insertion rate of a partition equals its eviction rate, $I_i = E_i$), we can obtain the analytical form

of α_2 with the following [1]:

$$\alpha_2 = \frac{S_2}{\sqrt[(R-1)]{\frac{I_1}{S_1}} - S_1} \quad (1)$$

Figure 3 shows the calculated scaling factors of Partition 2 (α_2) with different insertion rates ($I_2 = 0.6$, 0.7, 0.8, 0.9) and size fractions ($S_2 = 0.2$, 0.25, 0.3, 0.35, 0.4). The number of replacement candidates R is 16. From the figure (and also from Equation (1)), we can see that, for Partition 2, as its insertion rate I_2 increases and its size fraction S_2 decreases, its scaling factor α_2 becomes larger. In general, in order to constrain a partition with high insertion rate I to a small size S, FS needs to scale up its cache lines' futility more (i.e., with a large scaling factor).

From Equation (1), we can see that, when $I_1 < S_1^R$, there is no valid α (i.e., α becomes negative), which indicates that there is no way to enforce such partitioning. This partitioning bound is not just for FS but for all replacement-based partitioning schemes. This is because the minimum eviction rate of Partition 1 ($E_{min,1}$) is S_1^R since on an eviction, the probability of all R candidates belonging to Partition 1 is S_1^R, in which case a line from Partition 1 has to be evicted. When the insertion rate of Partition 1 is always smaller than its eviction rate (i.e., $I_1 < S_1^R = E_{min,1} \leq E_1$), the size of Partition 1 will eventually shrink and the desired partitioning cannot be enforced. In practice, when $R = 16$, Partition 1 with a very small insertion rate, e.g., $I_1 = 0.01$, can occupy about $\sqrt[16]{0.01} \approx 75\%$ of the total cache size, which is reasonably good in most scenarios.

C. Associativity

Figure 4 compares the associativity CDF of the FS and PF schemes with $S_1/S_2 = 9/1$, $6/4$ when $I_1/I_2 = 1$. For all the experiments in the rest of this section, the results are collected from a trace-driven simulator running two *mcf* threads on a $2MB$ *"random candidates"* cache [17] (i.e., the cache follows the *Uniformity Assumption*) with the number of candidates $R = 16$ and the insertion rate of each partition is controlled by adjusting the speed of the trace feeding (i.e., the probability of next insertion that belongs to Partition i is equal to the pre-configured insertion rate I_i). As can be seen from the figure, for the PF scheme, as the partition's size decreases, its associativity reduces, e.g., the average eviction futility (AEF) of Partition 2 decreases from 0.86 to 0.63 when its size reduces from 0.4 to 0.1. This is because, as the partition size shrinks, the number of replacement candidates belonging to this partition becomes smaller on each eviction, which reduces the probability to evict high-futility cache lines at the VI step of the PS scheme (as mentioned above in Section III-C). For the FS scheme, the associativity of Partition 1 always stays the same, while the associativity of Partition 2 reduces as its size decreases. For example, when the size fraction of Partition 2 reduces from 0.4 to 0.1, its scaling factor (α_2) increases from 1.031 to 1.624 and correspondingly its AEF decreases from 0.94 to 0.81. This is because the associativity of a partition in FS depends on its scaling factor. When a partition has a larger scaling factor α, a cache line of this partition with original futility of x is evaluated as less useful (i.e., higher scaled futility) compared to the lines with the same original futility of x in other partitions that have smaller scaling factors, and correspondingly has a higher probability to be evicted. Therefore, the AEF of this partition becomes lower and thus its associativity is degraded. When a partition's line futility is unscaled ($\alpha = 1$), its associativity will be fully preserved (i.e., the same associativity as a private cache with the same size). Note that the associativity degradation in FS is *not* caused by the increasing number of partitions, but rather determined by its scaling factor and thus its I/S ratio. If all the partitions have the same I/S ratio (i.e., $I_i/S_i = 1$), they will have the same scaling factor (i.e., $\alpha_i = 1$) and the associativity of all partitions will be fully preserved regardless of the number of partitions, i.e., the same as a non-partitioned cache. Overall, for both Partition 1 and 2, FS maintains much better associativity than PF.

D. Sizing

Figure 5 compares the cumulative distributions of Partition 1's size deviation from its target with the FS and PF schemes. The data are sampled at every eviction. The insertion rates of Partition 1 are $I_1/I_2 = 9/1$, $5/5$ and the cache is equally partitioned ($S_1/S_2 = 1$). From the figure, we can see that PF has the near ideal sizing property that the actual size of a partition is close to its target size (i.e., its mean absolute deviation (MAD) is smaller than 1). Statistically, FS also enforces the partitioning properly, i.e., the average size of each partition equals its target. The actual size of each partition in FS can be temporally deviated from its target (i.e., having more or less cache lines). The temporal deviation is smaller if its curve is more skewed towards $x = 0$. On an eviction, the probability that the actual size of Partition 1 increments by one line is $I_1 \times (1 - I_1)$, i.e., the probability of the insertion belonging to Partition 1 (i.e., I_1) multiplies the probability of the eviction not from Partition 1 (i.e., $1 - E_1 = 1 - I_1$, where $E_1 = I_1$ at a steady state). Similarly, the probability that the size of Partition 1 decrements by one line is also $I_1 \times (1 - I_1)$. Therefore, the size of Partition 1 with a smaller $I_1 \times (1 - I_1)$ will have less possibility to be deviated away from its target and, hence, its size deviation distribution curve will be more skewed towards $x = 0$, which is better. The size deviation distribution for Partition 1 becomes worst when $I_1 = 0.5$, where $I_1 \times (1 - I_1)$ is the largest, i.e., 0.25. From this figure, we can see that, even in the worst case, the deviation is relatively small, i.e., its MAD is smaller than 68, which is less than 0.5% deviation for a partition of $1MB$ (i.e., $16K$ lines with each line of $64B$).

[1] A brief derivation of this analytical expression is provided in a technical report [21].

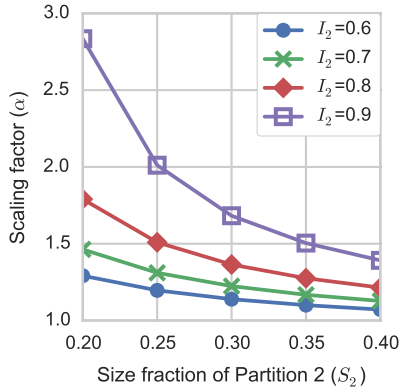


Figure 3: Scaling factors with different insertion rates and size fractions

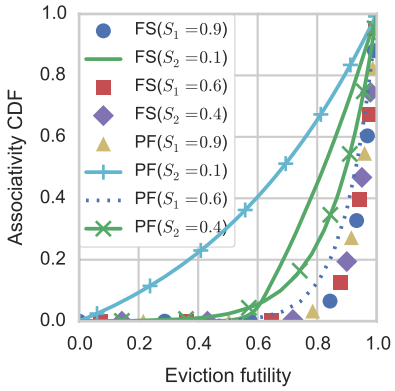


Figure 4: Associativity CDF of FS and PF with different size ratios

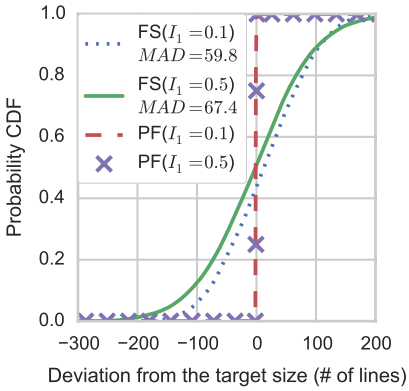


Figure 5: Size deviation of FS and PF with different insertion rates

E. Summary

Compared to the PF scheme, FS trades some small temporal size deviations for preserving high associativity. When facing the associativity and sizing dilemma described in Figure 1, FS may evict the least useful line in the first partition (as long as its scaled futility is larger than the scaled futility of the most useful line in the second partition) and thereby preserve the associativity. Although the actual size of each partition may be deviated further away from its target temporally, FS is still able to maintain the partition's actual size statistically close to its target, as explained above in this section.

The scaling factors of FS with more than two partitions can be derived in a similar way [21]. Due to space limitations, the results are not presented here. However, our experimental results show that FS with more than two partitions share similar qualitative properties on both associativity and sizing as FS with two partitions, which we summarize as follows.

1) *(Associativity)*: FS can maintain high cache associativity regardless of the number of partitions. In FS, a partition with a higher I/S ratio will have a larger scaling factor α and, thus, worse associativity. However, this associativity degradation is *not determined by the number of partitions but by the partition's I/S ratio*.
2) *(Sizing):* FS can enforce the size of a partition statistically very close to its target size. The partition with a higher $I_i \times (1 - I_i)$ will have a higher probability to be temporally deviated away from its target size. However, even in the worst case (i.e., $I_i = 0.5$), the deviation is still reasonably small.

V. Feedback-based Futility Scaling

As shown in the previous section, the Futility Scaling (FS) scheme has good properties on both associativity and sizing. However, it is not practical to implement it by using only the analysis above for the following two reasons. For one, it is not robust. The prior analysis is based on the *Uniformity Assumption* which is not exactly true in a real cache (though it is very close in a cache with good hash indexing [17]). Moreover, the exact insertion rate of each partition is hard to be predicted accurately as an application's behavior changes over time. Those imprecisions could cause the actual size of a partition shifted away from its target size. Second, it requires knowledge of the exact futility of every line. This would incur a large implementation overhead in practice.

In this section, we present a practical FS implementation to address these problems. Futility is estimated by a simple coarse-grain timestamp-based LRU [17], and the required scaling factor of each partition is dynamically adjusted through a feedback mechanism.

A. Feedback-based Scaling Factor Adjustment

We use a coarse-grain timestamp-based LRU (proposed in [17]) as the base futility ranking scheme for each partition. Each partition has an 8-bit counter for its current timestamp. An incoming cache line is tagged with the current timestamp of its partition and a partition's current timestamp counter is incremented by one every K accesses ($K = 1/16$ of this partition's size).

In a coarse-grain timestamp-based LRU, the timestamp of a cache line having a longer distance from the current timestamp of its partition indicates this line is less recently used. We estimate the futility of a cache line by the distance between its timestamp and the current timestamp of its partition. The timestamp-based futility (f_{ts}) of a cache line belonging to Partition i tagged with timestamp of x is calculated as $f_{ts} = (CurrentTS_i + 256 - x)\%256$, which is just an unsigned 8-bit subtraction in hardware.

We use a feedback-based approach to adjust the scaling factor of each partition dynamically. In general, enlarging or reducing the scaling factor of a partition will increases or decreases its eviction rate and shrinking or expanding its size correspondingly. Algorithm 2 describes how the scaling factor of each partition should be adjusted.

Algorithm 2 Feedback-based scaling factor adjustment

Require: N_i^E: number of evictions. N_i^I: number of insertions. N_i^A: actual number of cache lines occupied by Partition i. N_i^T: target number of cache lines assigned to Partition i. l: interval length. $\Delta\alpha$: changing ratio.

For each partition

```
if N_i^E ≥ l or N_i^I ≥ l then
    if N_i^I ≥ N_i^E and N_i^A > N_i^T then
        α_i ← α_i * Δα
    else if N_i^I ≤ N_i^E and N_i^A < N_i^T then
        α_i ← α_i ÷ Δα
    N_i^I ← 0
    N_i^E ← 0
```

The scaling factor of each partition is adjusted every l insertions or evictions (whichever is achieved first) for the partition in the following fashion. For each partition, every time that the insertion or eviction counter reaches l (i.e., the interval length), if the partition is oversized (i.e., current actual partition size N_i^A is greater than its target size N_i^T, $N_i^A > N_i^T$) and has a tendency to grow (i.e., number of insertions N_i^I is greater than number of evictions N_i^E in the last interval, $N_i^I \geq N_i^E$), the scaling factor α_i of this partition will be scaled up by a factor of $\Delta\alpha$. Similarly, If $N_i^A < N_i^T$ and $N_i^I \leq N_i^E$, the scaling factor α_i of Partition i will be scaled down by $\Delta\alpha$. At the end of each interval, both N_i^E and N_i^I are then reset to zero. The interval length (l) is determined by both the number of insertions N_i^I and evictions N_i^E of a partition so that the scaling factor adjustment process can respond promptly to both the increasing (i.e., N_i^I reaches l first) and decreasing (i.e., N_i^E reaches l first) of a partition's size. By checking the tendency of size changing in the last interval (i.e., $N_i^I \leq N_i^E$ or $N_i^I \geq N_i^E$), FS controller can avoid over-scaling line futility of a partition in the transient period of resizing, e.g., if a partition has a tendency to shrink its size, FS controller will stop increasing the scaling factor of this partition even if its current actual size is still above its target size.

In our FS implementation, we find that the interval length $l = 16$ is a sensible value. For hardware implementation efficiency, we set $\Delta\alpha$ to 2 so that the scaling factor will always be the power of two and the multiplication of a timestamp-based futility and a scaling factor can be done by a simple bit-shift operation in hardware.

B. Implementation

The feedback-based FS scheme uses a coarse-grain timestamp-based LRU as the underlying futility ranking scheme, which incurs around 1.5% storage overhead of the total cache (as described in [9]). Besides the cost of implementing a coarse-grain timestamp-based LRU, the FS cache controller only needs additional five registers per partition: 16-bit $ActualSize$ and $TargetSize$, 4-bit $InsertionCounter$ and $EvictionCounter$, and a 3-bit $ScalingShiftWidth$. The $TargetSize$ registers are set by an external allocation policy. We explain how to update the rest of registers in the following.

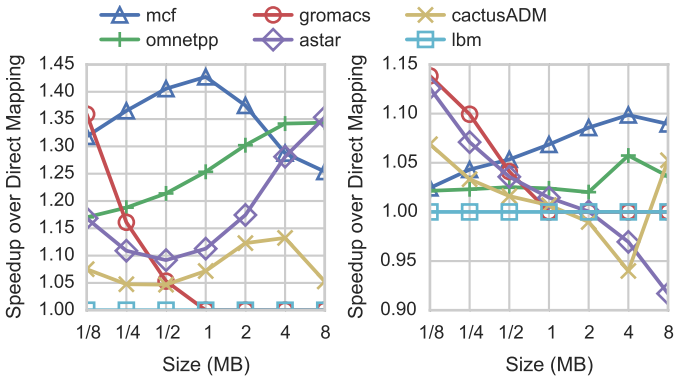
On a hit, the timestamp of the accessed cache line will be updated to the current timestamp of this partition obtained from the coarse-grained timestamp-based LRU procedure [9].

On a miss, the controller calculates the scaled futility for all the candidates, chooses the candidate with the largest scaled futility to evict, and inserts the incoming line.

- The timestamp-based futility for a cache line belonging to Partition i and tagged with timestamp x is calculated by one 8-bit hardware operation as $f_{ts} = (CurrentTS_i + 256 - x)\%256$. The scaled futility ($f_{ts}^{scaled}$) of a candidate from Partition i is obtained through left shifting f_{ts} by $ScalingShiftWidth_i$ bits, i.e., $f_{ts}^{scaled} = f_{ts} << ScalingShiftWidth_i$. Then the candidate with the largest scaled futility will be evicted. The $EvictionCouter_i$ and $ActualSize_i$ of the evicted line's partition increments and decrements by one, respectively.
- The incoming line is inserted with the current timestamp of this partition obtained from the coarse-grained timestamp LRU procedure [9]. Both $InsertionCounter_i$ and $ActualSize_i$ of the incoming line's partition increments by one.

Additionally, to implement the scaling factor adjustment scheme described in Algorithm 2, the scaling factor of Partition i is adjusted when $InsertionCounter_i$ or $EvictionCounter_i$ crosses 0. If $AcutalSize_i > TargetSize_i$ and $InsertionCounter_i = 0$ (This means $InsertionCounter_i$ reaches $l = 16$ first and thus there are more insertions than evictions in the last interval), $ScalingShiftWidth_i$ increments by one. Similarly, if $AcutalSize_i < TargetSize_i$ and $EvictionCounter_i = 0$, $ScalingShiftWidth_i$ decrements one. Otherwise, $ScalingShiftWidth$ stays the same. A $ScalingShiftWidth$ register is a 3-bit saturation counter (i.e., the range of its value is from 0 to 7) so that the timestamp-based futility f_{ts} can be scaled up by a factor of $2^7 = 128$ at most. After each adjustment, both $InsertionCounter_i$ and $EvictionCounter_i$ are reset to zero.

The FS controller only needs a few narrow adders and comparators to implement required counter updates and comparisons. Operations on hits are all about coarse-grain timestamp-based LRU updates, which are fairly simple without increasing the critical path [9]. On misses, for the cache with R replacement candidates, the controller needs to calculate the timestamp-based futility of each candidate (i.e., R subtraction operations), scale them (i.e., R shift operations) and then identify the candidate with the largest scaled futility (i.e., $R-1$ comparisons). So, in total, there are $3R-1$ simple operations that can be easily paralleled and pipelined in hardware. Those operations for the replacement process can be implemented over multiple cycles since they are off the critical path.



(a) OPT (b) LRU

Figure 6: Speedups of 6 benchmarks on fully-associative versus direct-mapped caches with different sizes

VI. Discussion

While FS can increase associativity, the impact of this increase in associativity on a real application's performance largely depends on the sensitivity of the application to associativity. The associativity sensitivity of an application could be reflected by the speedup of the application running on a fully-associative cache versus a direct-mapped cache. The higher speedup an application has, the more sensitive to associativity an application is. An application's associativity sensitivity is subject to two factors. One is the access pattern of an application itself. For example, if the access pattern of an application is streaming, improving cache associativity will have no effect on its performance. Figure 6a shows the speedups of 6 SPEC CPU2006 benchmarks running on fully-associative caches versus direct-mapped caches with the OPT ranking scheme. The cache sizes vary from $128KB$ to $8MB$. As can be seen in the figure, different applications have different sensitivities to associativity. With full associativity, *mcf* achieves speedups of at least 25% on all sizes of caches, while *lbm* has little improvement. An application's associativity sensitivity also varies with cache sizes. For example, gromacs has more than 35% speedup on a $128KB$ cache but negligible speedup when the cache size is over $1MB$. The second factor that affects an application's associativity sensitivity is a futility ranking scheme. An ideal futility ranking scheme based on OPT ranks a cache line with high re-reference potential as less futile. However, a practical futility ranking scheme may not always achieve this effect. Figure 6b shows the speedups of 6 benchmarks on fully-associative caches versus direct-mapped caches with LRU ranking scheme. As shown in the figure, compared to OPT, the associativity sensitivity of those benchmarks with the LRU has been dramatically reduced. For example, *mcf* only achieves 10% speedup at most with LRU (compared to at least 25% speedup with OPT) on fully-associative caches versus direct-mapped caches. In some scenarios, when cache lines with high re-reference potential are mistakenly ranked as useless, it could cause ill-effects for high-associativity caches, i.e., *increasing associativity could hurt performance*. As shown in Figure 6b, cactusADM loses 6% performance on a $4MB$ fully-associative cache compared to its performance on a direct-mapped cache with the same size.

Our FS partitioning scheme is conceptually independent of a futility ranking scheme. In this paper, we use a practical coarse-grain timestamp-based LRU ranking scheme to demonstrate the feasibility of our feedback-based FS design. We also report the results of the FS scheme working with an ideal OPT ranking scheme to show the performance headroom that can be obtained from the increased associativity in FS if a potential better ranking scheme is available.

VII. Methodology

A. System Configuration

Table II: System Configuration

Cores	2 GHz in-order, x86-64 ISA, 32 cores
L1 $s	split I/D, private, 32KB, 4-way set associative
	1-cycle latency, 64B line
L2 $	16-way set associative, non-inclusive, unified, shared
	8-cycle access latency, 64B line
	8 MB NUCA, 4 banks, 4-cycle average L1-to-L2 latency
	Futility Ranking: coarse-grain timestamp-based LRU/ OPT
MCU	200 cycles zero-load latency, 32 GB/s peak memory BW

Our simulator models a shared last-level (L2) cache, a mesh network and an off-chip memory. The simulator is fed with L2 access traces collected from Sniper simulator [22] that models an in-order core, on-chip L1 caches and a perfect L2 cache (i.e., no L2 misses). During the trace-driven simulation, network and memory access latency will be fed back into trace timing and, thus, delay future L2 cache accesses accordingly. The system is configured with $32KB$ private split I/D L1s for each core and an $8MB$ unified shared 16-way set associative L2 cache with an XOR-based indexing [19]. The detailed configuration of the system is shown in Table II.

B. Schemes

We compare five cache partitioning schemes with both coarse-grain timestamp-based LRU and the ideal OPT ranking schemes.

PF: This scheme first chooses the most oversized partition among all the candidates' partitions and then evicts the candidate with the largest futility from the chosen partition, as described in Algorithm 1.

PriSM [10]: This scheme first selects a partition in accordance to the pre-computed eviction probability distribution and then evicts the least useful replacement candidate belonging to the selected partition.

Vantage [9]: This scheme controls the size of each partition by adjusting its "aperture". In our experiment, Vantage is configured in the same way as it in [9] on a 16-way set associative cache, i.e., an unmanaged region $u = 10\%$, a maximum aperture $A_{max} = 0.5$ and $slack = 0.1$.

FullAssoc: This scheme is referred to as the PF scheme on a fully-associative cache. It always evicts the least useful cache line from the partition that exceeds its target size most. FullAssoc scheme is an ideal partitioning scheme that provides exact partitioning and full associativity for each partition.

Feedback-based FS: This is our proposed scheme that controls the size of each partition by scaling its line futility. In our experiment, we set the changing ratio $\Delta\alpha = 2$ and the interval length $l = 16$ by default.

C. Workloads

We mix multiprogrammed SPEC CPU 2006 benchmarks with reference input for our evaluation. For each benchmark, we use SimPoint [23] to select a representative region of 250M instructions. The simulation runs until 250M instructions have been executed for each thread.

VIII. Evaluations

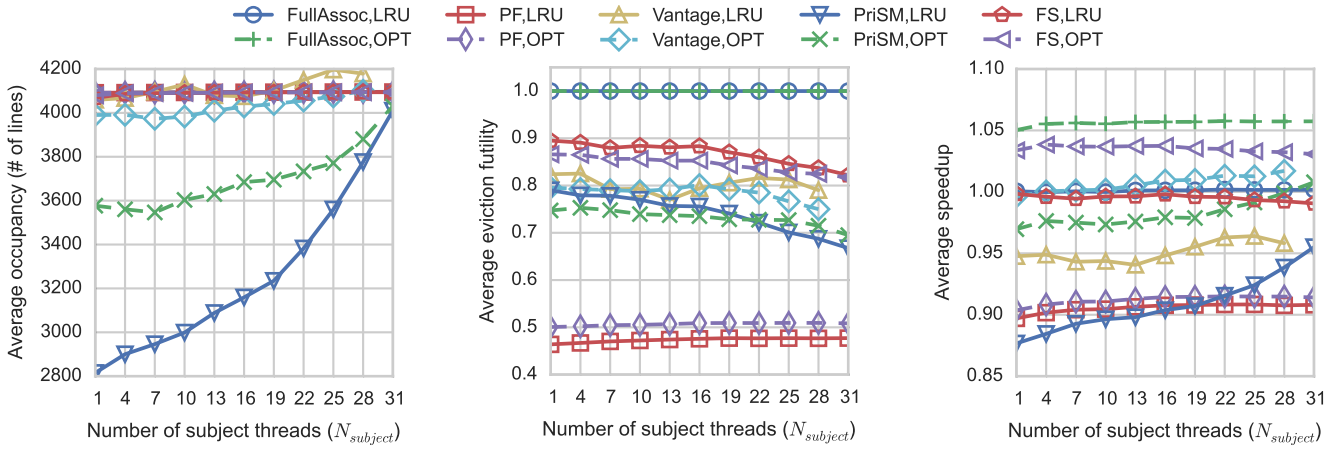
The proposed FS scheme is first compared with four other partitioning schemes (i.e., PF, Vantage [9], PriSM [10] and FullAssoc) in a QoS enabled environment. We then focus on the FS scheme itself, showing its sensitivity to two configuration parameters.

A. QoS Performance

We compare the associativity and sizing properties of FS, PF, Vantage, PriSM and FullAssoc on a QoS enabled CMP that provides cache space guarantees for 32 concurrently executing threads. Each workload mix running in this system is constructed by two types of application threads: subject threads that require cache space guarantees and background threads that have no QoS requirement. In our experiment, each subject thread runs an associativity-sensitive benchmark *gromacs* while each background thread runs a memory-intensive benchmark *lbm*. Note that *lbm* has a much higher miss rate than *gromacs* which would more aggressively occupy the cache space if resource sharing is unregulated. The system allocation policy assigns $256KB$ cache capacity (i.e., 4096 cache lines) to each subject thread and divides the rest of cache capacity equally among background threads. To evaluate the efficacy of our proposal, we generate 11 workloads mixes by varying the number of subject threads ($N_{subject}$) in each mix (i.e., $N_{subject}$ is from 1 to 31 in increments of 3), and accordingly the number of background threads in each mix is $32 - N_{subject}$. Vantage is not evaluated under the workload with 31 subject threads (requiring $^{31}/_{32} \approx 97\%$ of total cache space) as it can only manage 90% of cache space.

Figure 7a compares the average occupancy of subject threads with different partitioning schemes in each workload mix. As shown in the figure, FullAssoc, PF and FS can enforce the cache occupancy of each subject thread very close to its target size. In Vantage, cache space is divided into a managed region and an unmanaged region. Vantage can provide strong isolation for partitions in the managed region if all the evictions are only from the unmanaged region. With strong isolation, Vantage can always over-provision cache space to a thread, i.e., guarantee a thread the exact amount of resource assigned by an allocation policy in the managed region and meanwhile, allow the thread to borrow more space from the unmanaged region. However, on a 16-way set associative cache, 10% unmanaged region ($u = 0.1$) is not large enough for Vantage to provide strong isolation, i.e., there is 18.5% probability ($P_{ev} = (1-u)^{16} \approx 0.185$ [9]) that a line in the managed region is forced to be evicted. Due to the relatively weak isolation, Vantage cannot enforce the sizes of the partitions strictly above their targets. As can be seen in the figure, although the average occupancy of partitions in Vantage is relatively close to their targets, it can be at most 3% below its target. Note that Vantage could provide a higher degree of isolation on a cache that provides more replacement candidates (e.g., Z4/52 zcache [9]). In PriSM, the replacement process has two steps: *(1) Partition-Selection*: choose a partition according to the pre-calculated eviction probability distribution and then *(2) Victim-Identification:* choose the victim that belongs to the partition selected by the partition-selection step. However, there is a possibility of "abnormality" that no replacement candidate belonging to a partition identified by the partition-selection step. PriSM is designed to properly enforce the desired partition size only when this "abnormality" is rare. However, in our experiment, due to the large number of partitions ($N = 32$) and the small number of replacement candidates ($R = 16$), the possibility of this "abnormality" becomes very high (more than 70%) and consequently PriSM loses the ability to properly enforce the desired partitioning. As shown in the figure, in PriSM, the average occupancy of subject threads is, on average, 20.9% and 9.9% with LRU and OPT, respectively, below the target among all the workloads.

Figure 7b compares the average eviction futility (AEF) of subject threads with different partitioning schemes in different workloads. As expected, FullAssoc always maintains full associativity (i.e., $AEF = 1$). PF suffers from severe associativity degradation, and its lowest AEF is less than 0.51 across all the workloads. FS provides consistent high associativity, i.e., its AEF is, on average, 0.86/0.84 with LRU/OPT across all the workloads. In Vantage, the lines in a managed region have smaller futility than the ones in an unmanaged region. Due to the high probability of forced evictions from the managed region (i.e., forcing to evict a line with a small futility), the associativity of the partitions in Vantage is slightly degraded, i.e., its AEF is, on average, 0.80/0.79 with LRU/OPT. PriSM partitions a cache in a similar way to PF, i.e., first choosing a partition (*Partition-Selection*) and then evicting from the selected partition (*Victim-Identification*). Therefore, PriSM is supposed to suffer from the associativity degradation the same way as PF when the list of replacement candidates is shortened in the *Victim-Identification* step. However, as shown in the



(a) Average occupancy of subject threads (b) Average eviction futility of subject threads (c) Average speedup of subject threads

Figure 7: Comparisons among five partitioning schemes with 11 workload mixes that require different cache space guarantees

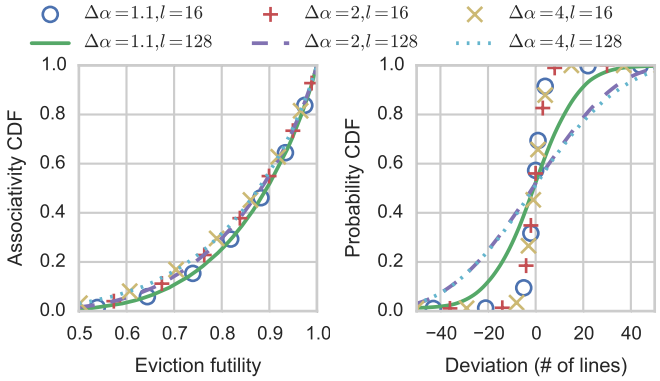
figure, PriSM achieves a better associativity than PF (i.e., its AEFs are above 0.73). This is due to the high occurrences of "abnormality", in which case, PriSM will choose a candidate with large futility regardless of partitioning requirements and thus improve associativity.

Figure 7c compares the average speedups ($\frac{IPC_{share}}{IPC_{alone}}$) of subject threads in each workload. IPC_{alone} refers to the IPC achieved when a subject thread running on a $256KB$ private 16-way set associative cache with LRU ranking. FullAssoc always achieves the best performance across different mixes, i.e., achieve average speedup of 1.0 and 1.05 with LRU and OPT, respectively. Due to the severe associativity degradation, the average speedups with the PF scheme are only 0.90 and 0.91 with LRU and OPT, respectively. In Vantage, the average speedups of subject threads achieve 0.95 and 1.0 with LRU and OPT, respectively. Because of the high occurrences of "abnormality" when the number of partitions is large, PriSM generally has little control of cache resources, and thus its average speedups are, on average, only 0.90 and 0.98 with LRU and OPT, respectively. Owing to its properties of high associativity and precise sizing, FS has steady high speedups of, on average, 0.995 and 1.036 with LRU and OPT, respectively, across all workload mixes.

In summary, with a coarse-grain timestamp-based LRU ranking, FS improves performance over PF, Vantage, and PriSM by up to 11.2%, 6.0%, and 13.7%, respectively, which is only 0.5% less than an ideal FullAssoc scheme. With an ideal OPT ranking, the high associativity provided by FS can further improve the performance, on average, by 4.1%, compared to the PF scheme.

B. Parameter Sensitivity analysis

The feedback-based FS scheme has two configurable parameters: the changing ratio of a scaling factor ($\Delta\alpha$) and the interval length for the scaling factor adjustment procedure (l). Figure 8 compares the associativity CDFs and



(a) Associativity distribution (b) Size deviation distribution

Figure 8: Comparisons of FS with different changing ratios ($\Delta\alpha$) and interval lengths (l)

the size deviation distributions of the first partition in FS with different changing ratios ($\Delta\alpha = 1.1, 2, 4$) and interval lengths ($l = 16, 128$). The experiments are conducted on a two-core system with a 2MB cache running *mcf* on each core. As can be seen in Figure 8a, by increasing $\Delta\alpha$, the associativity of FS is slightly degraded, i.e., AEF drops from 0.864 to 0.839 when $\Delta\alpha$ goes from 1.1 to 4. This is because with a larger $\Delta\alpha$, a partition has a higher chance of having an overlarge scaling factor (i.e., the partition's line futility is over-scaled) and a larger scaling factor can lead to a worse associativity, as we discussed in Section IV-C. Figure 8a also shows that the FS associativity is not sensitive to the changes of the interval length l, i.e., little associativity change when l goes from 16 to 128. From Figure 8b, we can see that, a larger $\Delta\alpha$ leads to a worse size deviation distribution, i.e., the mean absolute deviation (MAD) increases from 20.9 to 29.6 as α goes from 1.1 to 4 at $l = 128$. This is because a larger $\Delta\alpha$ results in larger changes in scaling factors, and consequently larger fluctuations in eviction rates and partition sizes. However, the interval length l can restrict

the effect of $\Delta\alpha$ on the sizing. Smaller l will make all the $\Delta\alpha$ configurations have a better sizing property (i.e., $MAD < 11.8$ at $l = 16$).

In summary, a smaller $\Delta\alpha$ has better associativity and sizing. Therefore, we choose to use a $\Delta\alpha$ as small as possible. Considering the efficiency of hardware implementation, we set $\Delta\alpha$ to 2 in order to convert multiplication operations (for scaling line futility) to bit-shift operations. While the interval length l has little impact on associativity, a smaller interval length can reduce size deviations. Hence, we use a small interval length ($l = 16$) to achieve a precise partitioning.

IX. Conclusion

We have presented Futility Scaling (FS), a novel replacement-based partitioning scheme that can precisely partition the whole cache while still maintain high associativity even with a large number of partitions. By scaling the futility of its cache lines properly, FS precisely controls the size of a partition. We have studied the properties of FS on both associativity and sizing in an analytical framework, and presented a feedback-based implementation of FS that incurs little overhead in practice. Simulation results show that, due to its properties of both precise sizing and high associativity, FS provides significant performance improvement over prior art.

Acknowledgments

We gratefully acknowledge the contributions from Timothy Pinkston in earlier stages of this work and the helpful suggestions from reviewers which have improved the paper. This research was supported, in part, by NSF grant CCF-1321131.

References

[1] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122.

[2] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, ser. PACT '06, Seattle, Washington, USA: ACM, 2006, pp. 13–22.

[3] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th International Symposium on Microarchitecture*, ser. MICRO '39, Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432.

[4] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *Proceedings of the 15th International Conference on Parallel architectures and Compilation Techniques*, ser. PACT '06, Seattle, Washington, USA: ACM, 2006, pp. 2–12.

[5] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '07, San Diego, California, USA: ACM, 2007, pp. 25–36.

[6] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, Washington, DC, USA: IEEE Computer Society, 2007, pp. 343–355.

[7] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith, "Multicore resource management," *Micro, IEEE*, vol. 28, no. 3, pp. 6–16, 2008.

[8] X. Zhou, W. Chen, and W. Zheng, "Cache sharing management for performance fairness in chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 384–393.

[9] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th International Symposium on Computer Architecture*, ser. ISCA '11, San Jose, California, USA: ACM, 2011, pp. 57–68.

[10] R Manikantan, K. Rajan, and R Govindarajan, "Probabilistic shared cache management (prism)," in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12, Portland, Oregon: IEEE Computer Society, 2012, pp. 428–439.

[11] D. Chiou, P. Jain, S. Devadas, and L. Rudolph, "Dynamic cache partitioning via columnization," in *Proceedings of Design Automation Conference*, Citeseer, 2000.

[12] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00, Vancouver, British Columbia, Canada: ACM, 2000, pp. 214–224.

[13] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell, "Molecular caches: a caching structure for dynamic creation of application-specific heterogeneous cache regions," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, Washington, DC, USA: IEEE Computer Society, 2006, pp. 433–442.

[14] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[15] D. Sanchez and C. Kozyrakis, "Scalable and efficient fine-grained cache partitioning with vantage," *Micro, IEEE*, vol. 32, no. 3, pp. 26 –37, 2012.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008, pp. 367–378.

[17] D. Sanchez and C. Kozyrakis, "The zcache: decoupling ways and associativity," in *Proceedings of the 43rd International Symposium on Microarchitecture*, ser. MICRO 43, Washington, DC, USA: IEEE Computer Society, 2010, pp. 187–198.

[18] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93, San Diego, California, USA: ACM, 1993, pp. 169–178.

[19] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through xor-based placement functions," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97, Vienna, Austria: ACM, 1997, pp. 76–83.

[20] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 288–299.

[21] R. Wang and L. Chen, "Futility scaling: high-associativity cache partitioning (extended version)," University of Southern California, Tech. Rep. CENG-2014-07, 2014.

[22] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: ACM, 2011, 52:1–52:12.

[23] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: faster and more flexible program analysis," *Journal of Instruction Level Parallelism*, 2005.