Vantage: Scalable and Efficient Fine-Grain Cache Partitioning

Daniel Sanchez and Christos Kozyrakis
Electrical Engineering Department, Stanford University
{sanchezd, kozyraki}@stanford.edu

ABSTRACT

Cache partitioning has a wide range of uses in CMPs, from guaranteeing quality of service and controlled sharing to security-related techniques. However, existing cache partitioning schemes (such as way-partitioning) are limited to coarse-grain allocations, can only support few partitions, and reduce cache associativity, hurting performance. Hence, these techniques can only be applied to CMPs with 2-4 cores, but fail to scale to tens of cores.

We present Vantage, a novel cache partitioning technique that overcomes the limitations of existing schemes: caches can have tens of partitions with sizes specified at cache line granularity, while maintaining high associativity and strong isolation among partitions. Vantage leverages cache arrays with good hashing and associativity, which enable soft-pinning a large portion of cache lines. It enforces capacity allocations by controlling the replacement process. Unlike prior schemes, Vantage provides strict isolation guarantees by partitioning *most* (e.g. 90%) of the cache instead of all of it. Vantage is derived from analytical models, which allow us to provide strong guarantees and bounds on associativity and sizing independent of the number of partitions and their behaviors. It is simple to implement, requiring around 1.5% state overhead and simple changes to the cache controller.

We evaluate Vantage using extensive simulations. On a 32-core system, using 350 multiprogrammed workloads and one partition per core, partitioning the last-level cache with conventional techniques *degrades* throughput for 71% of the workloads versus an unpartitioned cache (by 7% average, 25% maximum degradation), even when using 64-way caches. In contrast, Vantage *improves* throughput for 98% of the workloads, by 8% on average (up to 20%), *using a 4-way cache*.

Categories and Subject Descriptors: B.3.2 [Design Styles]: Cache memories; C.1.4 [Parallel Architectures]

General Terms: Design, Performance

Keywords: Cache partitioning, shared cache, multi-core, QoS

1. INTRODUCTION

As Moore's Law enables chip-multiprocessors (CMPs) with hundreds of threads, controlling capacity allocation in the cache hierarchy to improve performance and provide quality of service (QoS) is becoming a first-order issue. In current designs with 16-128 threads sharing the last-level cache [13, 23], a few threads can use most of its capacity, degrading performance for other applications running in parallel [7, 9, 10].

To address this issue, we can explicitly partition cache capacity across multiple threads. Additionally, cache partitioning has several important uses beyond enforcing isolation and QoS in systems with shared caches. For example, in CMPs with private caches, capacity sharing schemes also need to partition each cache [18]. Several software-controlled memory features like local stores [4, 5] or line pinning [16] can be implemented through partitioning. Architectural proposals such as transactional memory and thread-level speculation [2, 8] use caches to store speculative data, and can use partitioning to avoid having that data evicted by non-speculative accesses. Finally, security schemes can use the isolation provided by partitioning to prevent timing side-channel attacks that exploit the shared cache [17].

A *cache partitioning* solution consists of an *allocation policy* to determine the space allocated per partition (e.g. to improve fairness, favor threads based on priorities, or maximize overall throughput [9, 19, 24]), and a *partitioning scheme* to actually *enforce* those allocations [3, 14, 20, 27]. While allocation policies are often easy to implement and efficient, current partitioning schemes have serious drawbacks. Ideally, a partitioning scheme should allow the cache to hold a large number of arbitrarily fine-grain partitions (e.g. hundreds of partitions of tens or hundreds of lines each). It should maintain strong isolation among partitions, and strictly enforce capacity allocations, without reducing cache associativity. Dynamically creating, deleting or resizing partitions should be quick and efficient. Finally, partitioning should require minimal changes to cache designs and add small state and logic overheads.

Unfortunately, existing schemes for allocation enforcement fail to meet these properties. Way-partitioning [3] is limited to few coarse-grain partitions (at most, as many partitions as ways) and drastically reduces the associativity of each partition. Other schemes partition the cache by sets instead of ways, either in hardware [20] or software [14], maintaining associativity. However, these methods also lead to coarse-grain partitions, require costly changes to cache arrays and expensive data copying or flushing when partitions are resized, and often do not work with shared address spaces. Finally, proposals such as decay-based replacement [26] or PIPP [27] modify the replacement policy to provide some control over allocations. However, they lack strict control and guarantees over partition sizes and interference, preclude the use of a specific replacement policy within each partition, and are often co-designed to work with a specific allocation policy.

In this paper we present *Vantage*, a scheme to enforce capacity allocations in a partitioned cache that addresses the shortcomings of prior proposals. Unlike other schemes that provide strict guarantees, Vantage does not restrict line placement depending on its partition, thus *maintaining high associativity on the partitioned cache* and enabling a large number of fine-grain partitions with capacities defined at line granularities. Partitions can be dynamically resized, created and removed efficiently. Vantage enforces capacity alloca-

Scheme	Scalable & fine-grain	Maintains associativity	Efficient resizing	Strict sizes & isolation	Indep. of repl. policy	Hardware cost	Partitions whole cache
Way-partitioning [3, 20]	No	No	Yes	Yes	Yes	Low	Yes
Set-partitioning [20, 25]	No	Yes	No	Yes	Yes	High	Yes
Page coloring [14]	No	Yes	No	Yes	Yes	None (SW)	Yes
Ins/repl policy-based [10, 26, 27]	Sometimes	Sometimes	Yes	No	No	Low	Yes
Vantage	Yes	Yes	Yes	Yes	Yes	Low	No (most)

Table 1: Classification of partitioning schemes.

tions by controlling the replacement process, but it still requires a replacement policy (e.g. LRU) to rank lines within each partition.

Vantage is *derived from statistical analysis, not empirical observation*. It works with highly-associative cache designs, like skew-associative caches [22] or zcaches [21], which have good analytical properties, namely, they provide a high-quality set of replacement candidates independently of the workload's access pattern. Hence, they enable us to effectively soft-pin a large portion of the lines in the cache through the replacement policy. Vantage provides strong guarantees on partition sizes and isolation by *partitioning most of the cache, not all of it*. Partitions can slightly outgrow their target allocations, but they borrow space from a small unpartitioned region of the cache, not from other partitions. Hence, Vantage eliminates destructive interference between partitions. Sizes are maintained *by matching the average rates at which lines enter and leave each partition*. We prove that by controlling partition sizes this way, the amount of cache space that has to be left unpartitioned for Vantage to work well is both small (e.g. around 5-15% in a 4-way zcache) and *independent of the number of partitions or their sizes*. Therefore, Vantage is *scalable*. Vantage also works with conventional set-associative caches, although with slightly reduced performance and weaker guarantees.

While these conceptual techniques provide strong guarantees, implementing them directly would be complex. We propose a practical design that relies on *negative feedback to control partition sizes in a way that maintains the guarantees of the analytical models without their complexity*. Our design just requires adding few bits to each tag (e.g. 6 bits to support 32 partitions) and simple modifications to the cache controller, which only needs to track about 256 bits of state per partition, and a few narrow adders and comparators for its control logic. On an 8 MB last-level cache with 32 partitions, Vantage adds a 1.5% state overhead overall.

We evaluate Vantage by simulating a large variety of multiprogrammed workloads on both 4-core and 32-core CMPs. We compare it to way-partitioning and PIPP using utility-based cache partitioning (UCP) [19] as the allocation policy. Vantage significantly improves the performance of UCP on the 4-core system (up to 40%), but results are most striking on the 32-core system: while using either way-partitioning or PIPP to partition a 64-way cache almost always degrades performance due to the large loss of associativity, Vantage is able to deliver similar performance improvements as in the 4-core system, maintaining 32 fine-grain, highly-associative partitions *using a 4-way cache* (i.e. 16 times fewer ways). Additional simulation results show that Vantage achieves the benefits and bounds predicted by the analytical models.

2. BACKGROUND

Partitioning requires an *allocation policy* to decide the number and sizes of partitions, and a *partitioning scheme* to *enforce* them. In this work we focus on the latter. Table 1 summarizes the differences between current approaches, which we review in this section. Broadly, there are two approaches to partition a cache:

Strict partitioning by restricting line placement: Schemes with strict partitioning guarantees rely on restricting the locations where a line can be placed depending on its partition. Way-partitioning or column caching [3] divides the cache by ways, restricting fills from each partition to its assigned subset of ways. Way-partitioning is simple, but has several problems: partitions are coarsely sized (in multiples of way size), the number of partitions is limited by the number of ways, and the associativity of each partition is proportional to its way count, imposing a trade-off between isolation and partition performance. For way-partitioning to work well, the number of ways should be significantly larger than the number of partitions, so this scheme does not scale to large partition counts.

To avoid losing associativity, the cache can be partitioned by sets instead of ways, as proposed by one flavor of reconfigurable caches [20] and molecular caches [25]. However, these approaches require configurable decoders or a significant redesign of cache arrays, and must do scrubbing, i.e. flushing or moving data when resizing partitions. Most importantly, this scheme will only work when we have fully disjoint address spaces, which is not true in most cases. Even different applications operating on separate address spaces share library code and OS code and data. A different approach to partition through placement restriction is to leverage virtual memory, using page coloring to constrain the physical pages of a process to map to a portion of the cache sets [14]. While this scheme does not require hardware support, it is limited to coarse-grain partition sizes (multiples of page size$\times$cache ways), precludes the use of superpages, does not work on caches that are indexed using hashing (common in modern processors [23]), and repartitioning requires costly recoloring (i.e. copying) of physical pages, so it must be done infrequently [14].

Soft partitioning by controlling insertion and/or replacement: Alternatively, a cache can be partitioned approximately by modifying the allocation or replacement policies. These schemes avoid some of the issues of restricting line placement, but provide only limited control over partition sizes and inter-partition interference. They are useful for partitioning policies that can work with approximate partitioning, but not for uses that require stricter guarantees. In selective cache allocations [10] each partition is assigned a probability p, and incoming lines from that partition are inserted with probability p or discarded (self-replaced) with probability $1 - p$. In decay-based replacement policies, lines from different partitions age at different rates; adjusting the rates provides some control over partition sizes [26]. Promotion-insertion pseudo-partitioning (PIPP) [27] assigns each partition a different insertion position in the LRU chain and slowly promotes lines on hits (e.g. promoting $\simeq 1$ position per hit instead of moving the line to the head of the LRU chain). With an additional mechanism to restrict cache pollution of thrashing applications, PIPP approximately attains the desired partition sizes. PIPP is co-designed to work with UCP as the allocation policy, and may not work correctly with other policies. Finally, as we will see in Section 6, PIPP's partitioning scheme does not scale with the number of partitions.

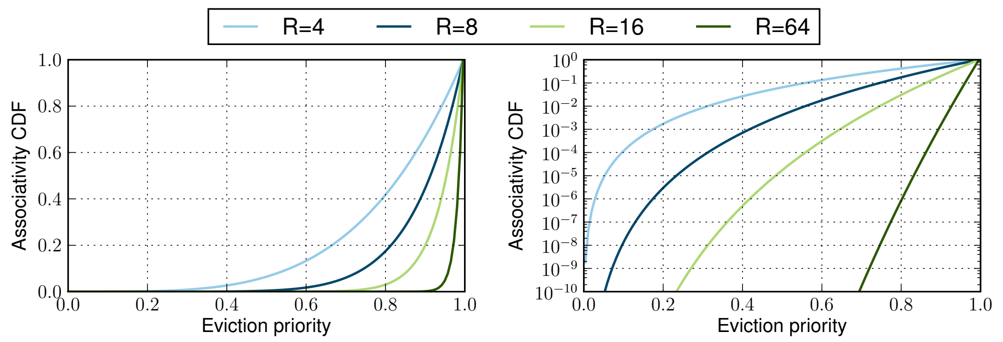


Figure 1: Associativity CDFs under the uniformity assumption ($F_A(x) = x^R, x \in [0, 1]$) for $R = 4, 8, 16, 64$ replacement candidates, in linear and logarithmic scales.

3. VANTAGE TECHNIQUES

3.1 Overview

Vantage is a partitioning scheme for caches with high associativity and good hashing, such as skew-associative caches [22] and zcaches [21]. These caches provide high, predictable associativity regardless of the workload (see Section 3.2), and thus can keep a large portion of the lines effectively pinned in the cache.

Vantage does not physically restrict line placement: lines from all partitions share the cache. It enforces partition sizes at replacement time. On each replacement, Vantage needs to evict one line from a set of replacement candidates. In a partitioned cache, this set may include good candidates from other partitions (i.e. lines that the owning partition would have to evict anyway). To strictly enforce partition sizes, we should always evict a candidate from the same partition as the incoming line. However, this does not scale with the number of partitions, as the portion of candidates from that specific partition will be increasingly small with more partitions. For example, a 16-way set-associative cache has 16 replacement candidates to choose from when unpartitioned, but only 2 when it is evenly divided in 8 partitions. The core idea behind Vantage is to relax this restriction, imposing only that the rates of insertions and evictions from each partition *match on average*. Since Vantage dynamically adjusts how to select candidates based on the insertion rate of each partition, we call this technique *churn-based management* (Section 3.4).

Unfortunately, churn-based management alone has several drawbacks: it allows interference across partitions (as choosing a candidate from another partition means taking space away from that partition and giving it to the one that caused the miss), makes it hard to provide strong guarantees on partition sizes, and requires a complex controller. To solve these issues, we partition *most* of the cache rather than all of it. We divide cache space into a managed region and a small unmanaged region (e.g. 15% of the cache), and partition only the managed region. Partitions can slightly outgrow their target allocations, borrowing space from the unmanaged region instead of from each other. This *managed-unmanaged region division* (Section 3.3) solves all interference issues, allows for a simple controller design, and significantly increases the associativity on the managed region.

Vantage's control scheme is derived from statistical analysis rather than empirical observation. It achieves *provable, strong guarantees*, namely, it eliminates inter-partition interference, provides precise control of partition sizes, and maintains high partition associativities, regardless of the number of partitions or the workload.

3.2 Caches with High Associativity

Vantage relies on caches with good hashing and high associativity, such as skew-associative caches [22] and zcaches [21]. Skew-associative caches index each way with a different hash function, spreading out conflicts. ZCaches enhance skew-associative caches with a replacement process that obtains an arbitrarily large number of candidates with a low number of ways. These caches exhibit two very useful properties for partitioning: they can restrict evictions to a specific portion of lines by simply controlling the replacement policy, and provide high associativity independently of the workload's access pattern [21].

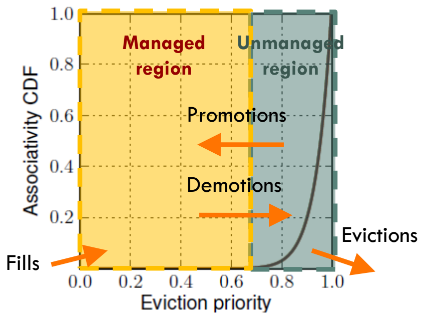
We follow the analytical framework for associativity described in [21], which we summarize here: A cache consists of an *array*, which implements associative lookups and gives a list of *replacement candidates* on each eviction, and a *replacement policy*, which defines a global rank of the lines in the cache. Each line is given an uniformly distributed *eviction priority* $e \in [0, 1]$. On a replacement, the cache controller always evicts the candidate with the highest eviction priority from the ones given by the array. The *associativity distribution* is the probability distribution of the eviction priorities of *evicted* lines. Intuitively, the more skewed towards 1.0 the distribution is, the higher the associativity of the cache.

For skew-associative caches and zcaches, the set of replacement candidates examined on every eviction is statistically very close to an uniform random selection of lines [21]. Hence, the associativity distribution can be derived analytically: if the array gives R uniformly distributed replacement candidates, the cumulative distribution function (CDF) of the associativity distribution is [21]:

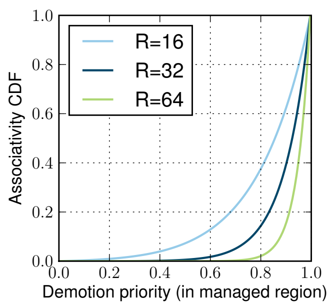
$$F_A(x) = Prob(A \leq x) = x^R, x \in [0, 1] \qquad (1)$$

Fig. 1 plots the associativity distribution for different values of R. *Associativity depends on the number of replacement candidates, not the number of ways.* The good randomization properties of skew-associative caches and zcaches allow them to match this distribution in practice, independently of the workload's access pattern or the replacement policy used [21]. Fig. 1 shows that the probability of evicting lines with a low eviction priority quickly becomes negligible. For example, with $R = 64$, the probability of evicting a line with eviction priority $e < 0.8$ is $F_A(0.8) = 10^{-6}$. Hence, by simply controlling how lines are ranked, we can guarantee that they will be kept in the cache with a very high probability. Unfortunately, this does not apply to set-associative caches, which tend to perform worse than the uniform candidates case [21].

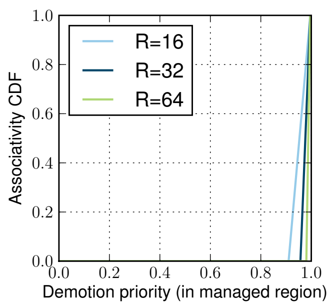
Vantage assumes that the underlying cache design meets the uniformity assumption $F_A(x)$. We will use zcaches in our evaluation,



(a) Managed-unmanaged region division for a $R = 16$ replacement candidates cache, and flows between regions.



(b) Associativity CDF in the managed region when doing *exactly* one demotion per eviction.



(c) Associativity CDF in the managed region when doing one demotion per eviction *on average*.

Figure 2: Managed-unmanaged region division: setup, flows and associativity in the managed region (assuming 30% of the cache is unmanaged).

since they are a cheap way of meeting this constraint. For example, a 4-way zcache can easily provide $R = 16$ or $R = 52$ candidates per replacement [21]. ZCaches, like skew-associative caches, break the concept of a set, so they cannot use replacement policies that rely on set ordering. Nevertheless, most replacement policies can be cheaply implemented, e.g. LRU can be implemented with 8-bit coarse-grain timestamps [21]. Vantage is not limited to zcaches and skew-associative caches, however. In Section 6 we show that Vantage can be used with hashed set-associative caches, although at higher cost (more ways) and with a slight loss of performance and analytical guarantees.

Assumptions: For the rest of this section, we make two assumptions in our analysis. First, we assume that the replacement candidates on each eviction are independent and uniformly distributed. Although this is not strictly the case, it is close enough [21] that our models are accurate in practice, as we will see in Section 6. Second, we assume that, on each replacement, we know the eviction priority of every candidate, as given by the replacement policy. While tracking eviction priorities would be very expensive in practice, Section 4 shows that we can achieve similar results with a much simpler scheme.

3.3 Managed-Unmanaged Region Division

We divide the cache in two logical regions: a *managed* and an *unmanaged* region. This division is done by simply tagging each line as either managed or unmanaged, and region sizes are set by controlling the flow of lines between the two regions. A *base replacement policy* (e.g. LRU) ranks lines as in an undivided cache, oblivious to the existence of the two regions. On an eviction, lines in the unmanaged region are always prioritized for eviction over managed lines. The unmanaged region is sized so that it captures most evictions, making evictions in the managed region negligible.

Fig. 2a illustrates this setup. It shows the associativity distribution of a cache with $R = 16$ candidates, divided in the managed and unmanaged regions, and the flows of lines between the two. To make evictions in the managed region negligible ($\simeq 10^{-3}$ probability), the unmanaged region is sized to 30% of the cache. Caches with $R > 16$ will require a smaller unmanaged region. Incoming lines are *inserted* in the managed region, eventually *demoted* to the unmanaged region, and either *evicted* from there, or *promoted* if they get a hit. Promotions and demotions do not physically move the line, just change its tag.

In a sense, the unmanaged region acts as a victim cache for the managed region. Evicting a line requires that it be demoted first (saving for the rare cases where we do not find a candidate from the unmanaged region). To keep the sizes of both regions constant, we would have to demote one line on each replacement and promotion. We denote the fraction of the cache devoted to the managed and unmanaged regions by m and u, respectively (e.g. in Fig. 2a, $m = 0.7$ and $u = 0.3$). Ignoring the flow of promotions (which is typically small compared to the evictions), if we demote exactly one line on each replacement, the associativity distribution *for demotions* inside the managed region is:

$$F_M(x) \cong \sum_{i=1}^{R-1} B(i,R)F_{A_i}(x) \qquad (2)$$

where $B(i,R) = \binom{R}{i}(1-u)^i u^{R-i}$ is the probability that i of the R replacement candidates are in the managed region (a binomial distribution), and $F_{A_i}(x) = x^i$ is the nominal associativity distribution with i replacement candidates[1]. Fig. 2b plots this distribution for various values of R.

To maintain the sizes of the two regions under control, however, it is not necessary to demote exactly one candidate per eviction. It suffices to demote one *on average*. For example, some evictions might not yield any candidates with high eviction priority from the managed region, while others might find two or more. By allowing demotions to work on the average case rather than being affected by the worst case, associativity increases significantly. In this case the controller only needs to select a threshold value, which we call the *aperture* (A), over which it will demote every candidate that it finds. For example, if $A = 0.05$, it will demote every candidate that is on the top 5% of eviction priorities (i.e. $e \geq 0.95$). Since, on average, $R \cdot m$ of the candidates are from the managed region, maintaining the sizes requires an aperture $A = \frac{1}{R \cdot m}$. The associativity distribution in the managed region is uniform $\sim U[1-A, 1]$, so the CDF is:

$$F_M(x) = \begin{cases} 0 & \text{if} \quad x < 1-A \\ \frac{x-(1-A)}{A} & \text{if} \quad 1-A \leq x \leq 1 \\ 1 & \text{if} \quad x > 1 \end{cases} \qquad (3)$$

Fig. 2c shows the associativity distributions for several values of R. By comparing Fig. 2b and Fig. 2c, we clearly see that demoting on the average significantly improves associativity. For example, with

[1]This formula is approximate, because we ignore the cases $i = 0$ (no replacement candidates are from the managed region, hence none can be demoted), and $i = R$ (all the candidates are from the managed region, so we need to evict from the managed region rather than demote). Both cases have a negligible probability.

$R = 16$ candidates, demoting on the average only demotes lines with eviction priority $e > 0.9$. Meanwhile, when demoting always one line per eviction, 60% of the demotions will happen to lines with $e < 0.9$.

Overall, using the unmanaged region has several advantages. First, it enables the controller to work on the average in the managed region, increasing associativity. Second, once we partition the managed region, partitions will borrow space from it instead of from each other, eliminating inter-partition interference. Third, it will make it practical to implement a Vantage controller (Section 4). While a portion of the cache must remain unpartitioned, this is typically a small percentage, e.g. 5-15% with $R = 52$ candidates (Section 4).

3.4 Churn-based Management

We now logically partition the managed *region*[2]. We have *P partitions* of *target sizes* $T_1, ..., T_P$, so that $\sum_{i=1}^{P} T_i = m$ (i.e. partition sizes are expressed as a fraction of the total cache size). These target sizes are given to Vantage by the allocation policy (e.g. UCP or software mechanisms). Partitions have *actual sizes* $S_1, ..., S_P$, and insertion rates, which we call *churns*, $C_1, ..., C_P$ (a partition's churn is measured in insertions per unit of time). Churn-based management keeps the actual size of each partition close to its target size by matching its demotion rate with its churn. It achieves this by controlling how demotions are done. Instead of having one aperture for the managed region, there is one aperture *per partition*, A_i. On each replacement, all the candidates below their partitions' apertures are demoted. Unlike way-partitioning, which achieves isolation by always evicting a line from the inserting partition, Vantage allows a partition's incoming line to demote others' lines. Vantage embraces interference and uses it to its advantage.

We now describe how churn-based management works on different cases. As in Section 3.3, we ignore the flow of promotions to simplify the analysis. Promotions are rare compared to insertions, hence we treat them as a small modeling error, addressed when implementing the controller (Section 4).

Partitions with similar behavior: The simplest case happens when partitions have both the same sizes (S_i) and churns (C_i). In this case, keeping all apertures equal, $A_i = \frac{1}{R \cdot m}$, will maintain their sizes. This is independent of how the base replacement policy ranks candidates, as we are demoting from the bottom A_i portion from each partition. Furthermore, the aperture is independent from the number of partitions: *Vantage retains the same associativity as if the cache was unpartitioned.*

Partitions with different behaviors: When partitions have different sizes and/or churns, apertures need to accommodate for this. A partition with a higher churn than the average will need a larger aperture, as we need to demote its lines at a higher frequency; and a partition that is smaller in size than the average will also need a larger aperture, because replacement candidates from that partition will be found more rarely.

Overall, partitions with a larger churn and/or a smaller size than the average will have a larger aperture, and partitions with a smaller churn and/or a larger size than the average will have a smaller aperture. For example, consider a case with 4 equally sized partitions ($S_1 = S_2 = S_3 = S_4$), where the first partition has twice the churn as the others ($C_1 = 2C_2$, $C_2 = C_3 = C_4$). The cache examines $R = 16$ replacement candidates per eviction, and the managed region takes $m = 62.5\%$ of the cache. On each replacement, $R \cdot m = 16 \cdot 0.625 = 10$ candidates are in the managed region on average. To maintain the partitions' sizes, on average, for every 5 demotions, 2 should be done from partition 1, and 1 demotion from each of partitions 2, 3 and 4. Every 5 demotions, Vantage gets $5 \cdot 10 = 50$ candidates from the managed region on average, $50/4 = 12.5$ candidates per partition since they are equally sized. Therefore, the apertures need to be $A_1 = 2/12.5 = 16\%$ for partition 1, and $A_2 = A_3 = A_4 = 1/12.5 = 8\%$ for the other partitions. Hence, partitions with disparate churns or sizes cause associativity to be unevenly distributed.

In general, when we have partitions with different sizes S_i and churns C_i, we can derive the aperture of each partition. Out of the $R \cdot m$ replacement candidates per demotion that fall in the managed region, a fraction $\frac{S_i}{\sum_{k=1}^{P} S_k}$ are from partition i, and we need to demote lines at a fractional rate of $\frac{C_i}{\sum_{k=1}^{P} C_k}$ in this partition. Therefore,

$$A_i = \frac{C_i}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{S_i} \frac{1}{R \cdot m} \quad (4)$$

Stability: Depending on the sizes and churns of the partitions, simply adjusting their apertures may not be enough to maintain their sizes. Even if we are willing to sacrifice associativity by allowing the aperture to reach up to 1.0 (demoting every candidate from this partition), a partition with a large C_i/S_i ratio may require a larger aperture. Since it is undesirable to completely sacrifice associativity to maintain partition sizes, we set a *maximum aperture* A_{max}. If using Equation 4 yields an aperture larger than A_{max}, we have three options. First, we can do nothing and let the partition grow beyond its target allocation, borrowing space from the unmanaged region. Second, we can allow low-churn/size $\rightarrow$ high-churn/size partition interference by inserting its lines in the unmanaged region (throttling its churn). Third, we can allow high-churn/size $\rightarrow$ low-churn/size partition interference by reducing the size of one or more low-churn partitions and allocating that space to the high-churn partition until its aperture is lower than A_{max}.

Doing nothing could lead, in principle, to borrowing too much space from the unmanaged region, making it too small and leading to frequent forced evictions from the managed region, breaking our scheme. However, this is not the case if we allow for some extra slack when sizing the unmanaged region. Consider what happens when several partitions cannot match their minimum sizes. Specifically, partitions $1, ..., Q$ ($Q < P$) have very small sizes (e.g. 1 line each) and high churns. Each partition will grow until it is large enough that its C_i/S_i ratio can be handled with aperture A_{max}. This *minimum stable size* is:

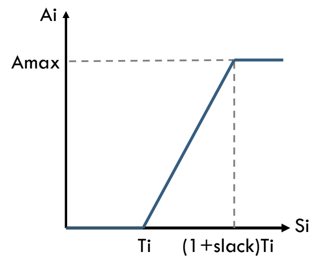
$$MSS_j = \frac{C_j}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m}, \forall j \in \{1, ..., Q\} \quad (5)$$

(obtained from Equation 4, with $S_j = MSS_j$ and $A_j = A_{max}$). Additionally, in the worst case, all other partitions ($Q + 1, ..., P$) have zero churn, so $\sum_{k=1}^{P} C_k = \sum_{k=1}^{Q} C_k$. In this case, the total space borrowed from the unmanaged region is:

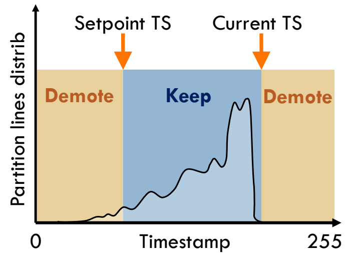
$$\sum_{j=1}^{Q} MSS_j = \frac{\sum_{j=1}^{Q} C_j}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m} = \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m} \quad (6)$$

and assuming $\sum_{k=1}^{P} S_k \cong m$, $\sum_{j=1}^{Q} MSS_j \cong {}^{1}/_{(A_{max}R)}$. For the exact derivation, $\sum_{k=1}^{P} S_k = \sum_{k=1}^{P} T_k + \sum_{j=1}^{Q} MSS_j$, and the target sizes achieve $\sum_{k=1}^{P} T_k = m$. By substituting on the previous equation, $\sum_{j=1}^{Q} MSS_j = {}^{1}/_{(A_{max}R - 1/m)}$. For any reasonable values of A_{max}, R and m, $A_{max}R \gg 1/m$, and therefore

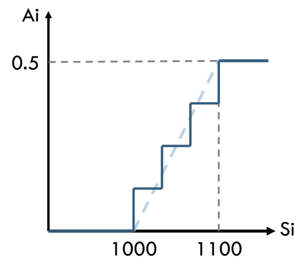
[2]Please note the distinction between regions and partitions: Vantage keeps two *regions*, managed and unmanaged, and divides the managed region in *partitions*.



(a) Linear transfer function used in feedback-based aperture control.



(b) Setpoint-based demotions selects candidates below setpoint (in modulo arithmetic).



Si range (lines)	Dems per 256 candidates
1000-1033	32
1034-1066	64
1067-1100	96
1101+	128

(c) 4-entry demotion thresholds lookup table for a 1000-line partition with 10% slack.

Figure 3: Feedback-based aperture control and setpoint-based demotions.

$\sum_{j=1}^{Q} MSS_j \cong 1/(A_{max}R)$ is a fine approximation. Hence, sizing the unmanaged region with an extra $1/(A_{max}R)$ of the cache guarantees that the scheme maintains the desired number of evictions from the managed region, *regardless of the number of partitions*! For example, if the cache has $R = 52$ candidates, with $A_{max} = 0.4$, we need to assign an extra $1/0.4{\cdot}52 = 4.8\%$ to the unmanaged region. Given that this is an acceptable size, we let partitions outgrow their allocations, disallowing inter-partition interference.

Transient behavior: So far, we have analyzed what happens in a *steady-state* situation. However, partitions may be suddenly *resized*. A partition that is suddenly downsized will need some time to reach its new target size (its aperture will be A_{max} during this period). Similarly, a partition that is suddenly upsized will take some time to acquire capacity (and will have an aperture of 0 until it reaches it). If we are reassigning space and upsized partitions gain capacity faster than downsized partitions lose it, the managed region may temporarily grow larger than it should be. In our evaluation, re-partitioning is infrequent and this is a minor issue. However, Vantage applications that resize partitions at high frequency should control the upsizing and downsizing of partitions progressively and in multiple steps.

Since partitions are cheap, some applications (e.g. local stores [4, 5]) might want to have a variable number of partitions, creating and deleting partitions dynamically. Deleting an existing partition simply requires setting its target size to 0, and its aperture to 1.0. When most or all of its lines have been demoted, the partition identifier can be reused for a new partition.

4. VANTAGE CACHE CONTROLLER

Vantage implements partitioning through the replacement process, so only the cache controller needs to be modified. Specifically, the controller is given the target sizes of each partition and the partition ID of each cache access. Partition sizes are set by an external resource allocation policy (such as UCP), and partition IDs depend on the specific application. In our evaluation, we have one partition per thread, but other schemes may have other assignments, e.g. local stores [4, 5] may partition by address range, TM and TLS [2, 8] would have extra partitions to hold speculative data, etc. Vantage tags each line with its partition ID, and, on each replacement, performs evictions from the unmanaged region and demotions from the managed region, as described in Section 3. However, implementing a controller simply using the previous analysis is impractical due to several reasons:

1. It is too compute-intensive: Each aperture A_i depends on the sizes and churns of all the other partitions (Equation 4 in Section 3.4), and they need to constantly change to adapt to time-varying behavior. Recomputing these on every replacement would be extremely expensive. Also, we need to estimate the churn (insertions/cycle) of each partition, which is not trivial.
2. It is not robust: The prior analysis has two sources of modeling errors. First, replacement candidates are not exactly independent and uniformly distributed (though they are close [21]). Second, the previous analysis ignores promotions, which have no matching demotion[3]. Even if we could perfectly estimate the A_i, these modeling errors would cause partition sizes to drift away from their targets.
3. It requires knowing the eviction priority of every line (in order to know which candidates are below the aperture): This would be extremely expensive to do in practice.

In this section, we address these issues with a practical controller implementation that relies on two techniques: *feedback-based aperture control* enables a simple and robust controller where the required aperture is found using feedback instead of calculating it explicitly, and *setpoint-based demotions* lets us demote lines according to the desired aperture without knowing their eviction priorities.

4.1 Feedback-based Aperture Control

Deriving the aperture of each partition is possible by using negative feedback alone. Once again, we let partitions slightly outgrow their target allocations, borrowing from the unmanaged region, and adjust their apertures based on how much they outgrow them. Specifically, we derive each aperture A_i as a function of S_i, as shown in Fig. 3a:

$$A_i(S_i) = \begin{cases} 0 & \text{if} \quad S_i \le T_i \\ \frac{A_{max}}{slack}\frac{S_i - T_i}{T_i} & \text{if} \quad T_i < S_i \le (1+slack)T_i \\ A_{max} & \text{if} \quad S_i > (1+slack)T_i \end{cases} \tag{7}$$

where T_i is the partition's target size, and $slack$ is the fraction of the target size at which the aperture reaches A_{max} and tapers off. This is a classic application of negative feedback: an increase in size causes an increase in aperture, attenuating the size increase. The system is stable: partitions can reach and exceed a size of $(1 + slack)T_i$, in which case A_{max} aperture is applied, and the

[3]One could argue that promotions are not bounded, so they may affect the strong guarantees derived in Section 3. Addressing this issue completely just requires to do one demotion per promotion on average, but we observe that in practice, promotions are rare compared to evictions, so demoting on evictions is enough for Vantage to work well.

dynamics of the system follow what was discussed in the previous section (i.e. the partition will reach a minimum stable size MSS_i). This linear transfer function is simple, works well in practice, and the extra space requirements are small and easily derived: in the linear region, $\Delta S_i = S_i - T_i = slack \cdot S_i \frac{A_i}{A_{max}}$. Using Equation 4 (Section 3.4), we get:

$$\Delta S_i = \frac{slack}{A_{max}} S_i \frac{C_i \sum_{k=1}^{P} S_k}{S_i \sum_{k=1}^{P} C_k} \frac{1}{R \cdot m} = \frac{slack}{A_{max}} \frac{C_i}{\sum_{k=1}^{P} C_k} \frac{1}{R} \tag{8}$$

Therefore, the aggregate outgrow for all partitions in steady-state is:

$$\sum_{i=1}^{P} \Delta S_i = \frac{slack}{A_{max} \cdot R} \tag{9}$$

We will need to account for this when sizing the unmanaged region. This is relatively small, e.g. with $R = 52$ candidates, $slack = 0.1$ and $A_{max} = 0.4$, $\sum_{i=1}^{P} \Delta S_i = 0.48\%$ of the cache size. This also reveals the trade-off in selecting the slack: with a larger slack, apertures will deviate less from their desired value due to instantaneous size variations, but it requires a larger unmanaged region, as partitions will outgrow their target sizes by a larger amount. We will see how to size the unmanaged region in Section 4.3.

4.2 Setpoint-based Demotions

Setpoint-based demotions is a scheme to perform demotions without tracking eviction priorities. We first explain it with a concrete replacement policy, then generalize it to other policies.

We use coarse-timestamp LRU [21] as the base replacement policy. Each partition has a *current timestamp* counter that is incremented every k_i accesses, and accessed lines are tagged with the current timestamp value. We choose 8-bit timestamps with $k_i = 1/16$ of the partition's size, which guarantees that wrap-arounds are rare. To perform demotions, we choose a *setpoint timestamp*, and all the candidates that are below it (in modulo 256 arithmetic) are demoted if the partition is exceeding its target size. We adjust the setpoint every c candidates seen from each partition in the following fashion: we have a counter for candidates seen from this partition, and a counter for the number of demoted candidates, d_i. Every time that the candidates counter reaches c, if $d_i > c \cdot A_i$ (i.e. $d_i/c > A_i$), the partition's setpoint is incremented, and if $d_i < c \cdot A_i$, it is decremented. Both counters are then reset. Additionally, we increase the setpoint every time the timestamp is increased (i.e. every k_i accesses), so that the distance between both remains constant.

Fig. 3b illustrates this scheme. Adjusting the setpoint allows us to track the aperture indirectly, without profiling the distribution of timestamps in the partition. In our controller, we find that $c = 256$ candidates is a sensible value. Since c is constant and, in our evaluation, target allocations are varied sparingly (every 5 million cycles), we do not even need to explicitly compute the desired aperture from the size (as in Equation 7). Instead, we use a small 8-entry *demotion thresholds lookup table* that gives the d_i threshold for different size ranges. Fig. 3c shows a concrete example of this lookup table, where we have a partition with $T_i = 1000$ lines, and a 10% slack. For example, if when we reach $c = 256$ candidates from this partition, its size is anywhere between 1034 and 1066 lines, having more/less than 64 demotions in this interval will cause the setpoint to be incremented/decremented. This table is filled at resize time, and used every c candidates seen.

This scheme is also extensible to other policies beyond coarse-timestamp LRU. For example, in LFU we would choose a setpoint

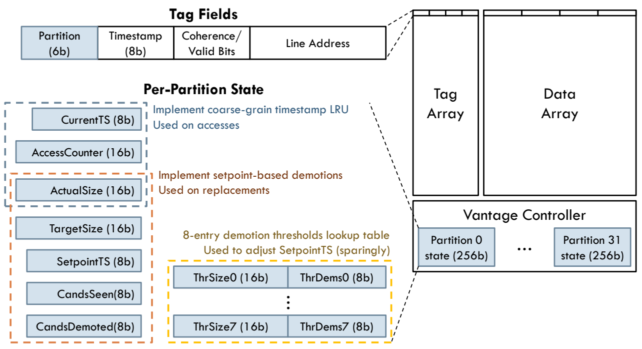


Figure 4: State required to implement Vantage: tag fields and per-partition registers. Additional state over an unpartitioned baseline is shown in blue. Each field or register shows its size in bits.

access frequency, and RRIP [12] can use a setpoint re-reference prediction value, as we will see in Section 6.

4.3 Putting it all Together

Now that we have seen the necessary techniques, we describe the implementation of the Vantage controller in detail.

State: Fig. 4 shows the state required by Vantage:

- Tag state: Each line needs to be tagged with its partition ID, and we need an extra ID for the unmanaged region. For example, with $P = 32$ partitions, we need 33 identifiers, or 6 bits per tag. If tags are nominally 64 bits, and cache lines are 64 bytes, this is a 1.01% increase in cache state. Note that each tag also has an 8-bit timestamp field to implement the LRU replacement policy, as in the baseline zcache.
- Per-partition state: For each partition, the controller needs to keep track of the registers detailed in Fig. 4. We explain how each of these registers is used below. Each register is labeled as either 8 or 16-bit, but 16-bit registers, which track sizes or quantities relative to size, assume a cache with 2^{16} lines. We assume that each of these registers is kept in partition-indexed register files. With $32K$ lines per bank, this amounts to 256 bits per partition. For 32 partitions and 4 banks (for an 8 MB cache), this represents 4 KBytes, less than a 0.5% state overhead.

Hits: On each hit, the controller writes the partition's *CurrentTS* into the tag's *Timestamp* field and increases the partition's *AccessCounter*. This counter is used to drive the timestamp registers forward: when *AccessCounter* reaches *ActualSize*/16, the counter is reset and both timestamp registers, *CurrentTS* and *SetpointTS*, are increased. This scheme is similar to the basic coarse-grained timestamp LRU replacement policy [21], except that the timestamp and access counter are per partition. Additionally, if the tag's *Partition* field indicates that the line was in the unmanaged region, this is a promotion, so *ActualSize* is increased and *Partition* is written when updating the *Timestamp* field.

Misses: On each miss, the controller examines the replacement candidates and performs one demotion on average, chooses the candidate to evict, and inserts the incoming line:

- All candidates are checked for demotion: a candidate from partition *p* is demoted when both *ActualSize[p]* > *TargetSize[p]* (i.e. the partition is over its target size) and the candidate's *Timestamp* field is not in between *SetpointTS[p]* and *CurrentTS[p]* (as shown in Fig. 3b), which requires two comparisons to decide. If the candidate is demoted, the tag's *Partition* field is changed to the unmanaged region, its *Timestamp* field is updated to the

Cores	32 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency
L2 cache	8 MB NUCA , 4 banks, 2 MB per bank, shared, non-inclusive, MESI directory coherence, 4-cycle average L1-to-L2-bank latency, 8-cycle L2 bank latency
MCU	4 memory controllers, 200 cycles zero-load latency, 32 GB/s peak memory BW

Table 2: Main characteristics of the large-scale CMP. Latencies assume a 32 nm process at 2GHz.

Insensitive (n)	perlbench, bwaves, gamess, gromacs, namd, gobmk, dealII, povray, calculix, hmmer, sjeng, h264ref, tonto, wrf
Cache-friendly (f)	bzip2, gcc, zeusmp, cactusADM, leslie3d, astar
Cache-fitting (t)	soplex, lbm, omnetpp, sphinx3, xalancbmk
Thrashing/streaming (s)	mcf, milc, GemsFDTD, libquantum

Table 3: Classification of SPEC CPU2006 workloads.

unmanaged region's timestamp, *ActualSize[p]* is decreased, and *CandsDemoted[p]* is increased. Regardless of whether the candidate is demoted or not, *CandsSeen[p]* is increased.

- The controller evicts the candidate from the unmanaged region with the oldest timestamp. If all candidates come from the managed region, it chooses one of the demoted candidates arbitrarily, and if no lines are selected for demotion, it chooses among all the candidates. Note that if the unmanaged region is sized correctly, the common case is to find candidates from it.
- The incoming line is inserted into the cache as usual, with its *Timestamp* field set to its partition's *CurrentTS* register, and its *ActualSize* is increased. As in a hit, *AccessCounter* is increased and the timestamps are increased if it reaches *ActualSize*/16.

Additionally, to implement the setpoint adjustment scheme from Section 4.2, partition *p*'s setpoint is adjusted when *CandsSeen[p]* crosses 0. At this point, the controller has seen 256 candidates from *p* since the last time it crossed 0 (since the counter is 8 bits), and has demoted *CandsDemoted[p]* of them. The controller finds the first entry *K* in the 8-entry demotion thresholds lookup table (as in Fig. 3b) so that the partition's threshold size, *ThrSize[K][p]*, is lower than its current size, *ActualSize[p]*. It then compares *CandsDemoted[p]* with the demotion threshold, *ThrDems[K][p]*. If the demoted candidates exceed the threshold, *SetpointTS[p]* is decreased, while if they are below the threshold, the setpoint is increased. Finally, *CandsDemoted[p]* is reset. Note that this happens sparingly, e.g. if the cache examines 64 replacement candidates per miss, the controller does one setpoint adjustment each $256/64 = 4$ misses on average, independently of the number of partitions.

Implementation costs: The controller requires counter updates and comparisons on either 8 or 16-bit registers, so a few narrow adders and comparators suffice to implement it. Operation on hits is simple and does not add to the critical path. On misses, demotion checks are the main overhead versus an unpartitioned cache, as the controller needs to decide whether to demote every candidate it sees, and each demotion check requires a few comparisons and counter updates. When a W-way zcache is used (typically $W = 4$ ways), replacements are done over multiple cycles, with the cache array returning at most W candidates per cycle. Therefore, a narrow pipeline suffices for demotions (i.e. we only need logic that can check $W = 4$ candidates per cycle). When using wider caches (e.g. a 16-way set-associative cache), the controller can implement demotion checks over multiple cycles, because the replacement process is not on the critical path [21]. Finally, note that, while all candidates are checked for demotion, only one on average is demoted per miss. Unlike other partitioning schemes, Vantage does not need to implement set ordering or LRU chains or pseudo-random number generation [10, 19, 27].

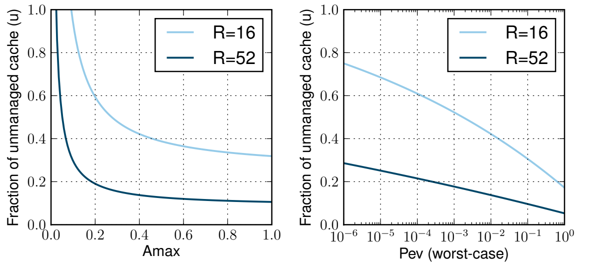
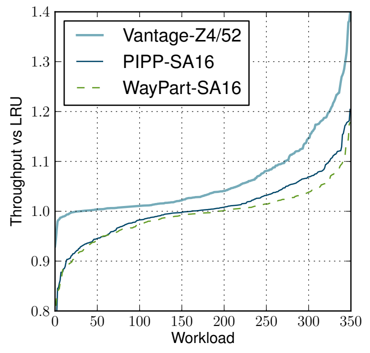


Figure 5: Fraction of the cache dedicated to the unmanaged region, with $slack = 0.1$ and $R = 16, 52$ candidates, both (a) as a function of A_{max}, with $P_{ev} = 10^{-2}$, and (b) as a function of P_{ev}, with $A_{max} = 0.4$.

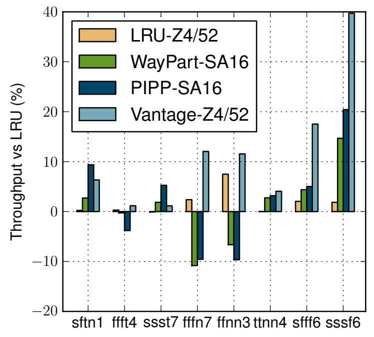
Sizing the unmanaged region: We finally have all the information needed to size the unmanaged region. First, from Equation 1 (Section 3.2), to have a worst-case probability of a forced eviction from the managed region P_{ev}, we need $F_A(m) = F_A(1-u) = P_{ev} = (1-u)^R$. Hence, at least we need $u \geq 1 - \sqrt[R]{P_{ev}}$. Additionally, we need to reserve ${}^{1}/_{(A_{max}R)}$ to allow high-churn/small-sized partitions to grow to their minimum stable sizes, and ${}^{slack}/_{(A_{max}R)}$ for feedback-based aperture control. Sizing $u = 1 - \sqrt[R]{P_{ev}} + {}^{(1+slack)}/_{(A_{max}R)}$ accounts for all these effects. Fig. 5 shows the fraction of the cache that needs to be unmanaged when varying both A_{max} and P_{ev}, for a 10% slack and $R = 16$ or 52 candidates. For example, with 52 candidates, having $A_{max} = 0.4$ requires 13% of the cache to be unmanaged for $P_{ev} = 10^{-2}$, while going down to $P_{ev} = 10^{-4}$ would require 21% to be unmanaged. Different applications will have different requirements for P_{ev}. For example, $P_{ev} \simeq 10^{-2}$ may suffice for applications that only require approximate partitioning, while applications with strong partitioning and isolation requirements may need $P_{ev} \simeq 10^{-4}$ or lower.

5. EXPERIMENTAL METHODOLOGY

Modeled systems: We perform microarchitectural, execution-driven simulation using an x86-64 simulator based on Pin [15], and model both small and large-scale CMPs. Our large-scale design has 32 in-order, single-threaded x86 cores modeled after Atom [6]. The system has private L1s and a shared 8MB, 4-bank L2, where the different partitioning schemes are implemented. Table 2 shows the details of the system. On a high-performance 32nm process, this CMP requires about 220 mm^2 and has a TDP of around 90W at 2GHz. Our small-scale design is similar, but has 4 cores, a 2MB L2 (1 bank) and 4GB/s of memory bandwidth.



(a) Results over all 350 workloads



(b) Throughput for selected workloads

Figure 6: Throughput improvements over an unpartitioned 16-way set-associative L2 with LRU obtained with different partitioning schemes on the 4-core configuration.

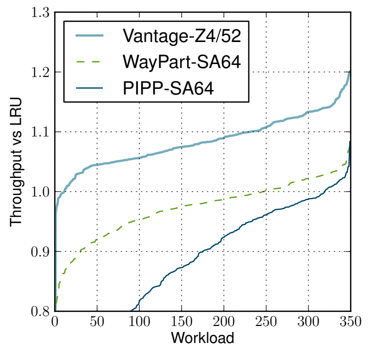


Figure 7: Throughput improvements over a 64-way set-associative L2 with LRU on the 32-core configuration.

Partitioning schemes: We compare Vantage against way-partitioning and PIPP. Way-partitioning uses LRU, and its replacement process is implemented as in [19]. PIPP is implemented as described in [27] ($p_{prom} = 3/4$, stream detection with $\theta_m \geq 12.5\%$, 1 way per streaming application and $p_{stream} = 1/128$).

Allocation policy: We use utility-based cache partitioning (UCP) to determine space allocation among partitions [19]. UCP uses auxiliary cache monitors to estimate how well each core uses cache capacity, and allocates more capacity to the threads that benefit from it the most. Each core has a small utility monitor based on dynamic set sampling (UMON-DSS) with 64 sets. Partition sizes are found with the Lookahead algorithm [19]. UCP repartitions the cache every 5 million cycles. When used with Vantage, UMONs are configured with the same number of ways as way-partitioning and PIPP are using, but since Vantage can partition at line granularity instead of at way granularity, we linearly interpolate the miss rate curves given by UMON, getting 256-point curves, and use them to drive the Lookahead algorithm.

Workloads: We use multiprogrammed SPEC CPU2006 application mixes, and follow the methodology of prior cache partitioning studies [19, 27]. Each application in the mix is fast-forwarded for 20 billion instructions, and the mix is simulated until all applications have executed 200 million instructions. We report aggregate throughput ($\sum IPC_i$), where each application's IPC is measured on its first 200 million instructions. Other studies also report metrics that give insight on fairness, such as weighted speedup or harmonic mean of weighted speedups [19, 27]. Due to lack of space, and because UCP attempts to maximize throughput, we report throughput only. We have checked these metrics and they do not offer additional insights. Fairness is mostly an issue of the *allocation policy*, i.e. UCP.

The 29 SPEC programs are divided in four categories, following a classification similar to the one in [11]. We first run each application alone, using cache sizes from 64KB to 8MB. Applications with less than 5 L2 misses per kilo-instruction (MPKI) are classified as *insensitive*; from the remaining ones, applications that gradually benefit from increased cache size are classified as *cache-friendly*; those where misses decrease abruptly with size when getting close to cache capacity (over 1MB) are classified as *cache-fitting*, and the ones where additional capacity does not yield any benefit are marked as *thrashing/streaming*. Table 3 shows this classification. There are 35 possible combinations (with repetitions) of these four categories, each of which forms a class. In the 4-core mixes, we have 10 mixes per class, with each application being randomly selected from the ones in its category, yielding 350 workloads. The 32-core mixes have 8 randomly chosen workloads per category, and again 10 mixes per class, for another 350 workloads.

6. EVALUATION

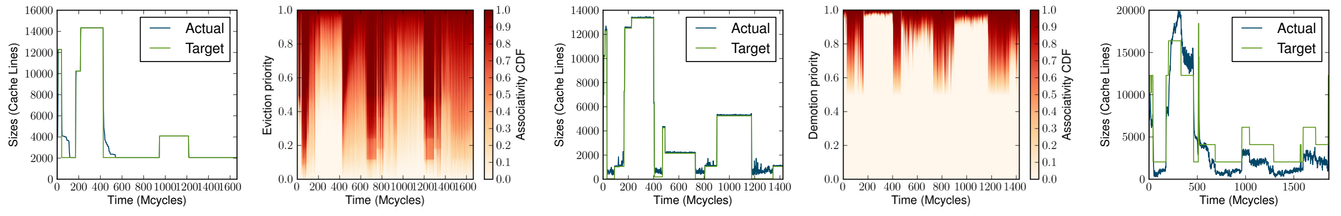
We first compare Vantage against other partitioning schemes using utility-based cache partitioning. We then present a series of experiments focused on Vantage, showing how to configure it, its sensitivity to configuration parameters, and confirm that the assumptions made in the analysis are met in practice.

6.1 Comparison of Partitioning Schemes

Small-scale configuration: Fig. 6a summarizes the performance results across the 350 workload mixes on the simulated 4-core system. Each line shows the throughput ($\sum IPC_i$) of a different scheme, normalized to a 16-way set-associative cache using LRU. For each line, workloads (the x-axis) are sorted according to the improvement achieved. All caches use simple H_3 hashing [1, 21], since it improves performance in most cases. Way-partitioning and PIPP use a 16-way set-associative cache, while Vantage uses a 4-way zcache with 52 replacement candidates (Z4/52), with a $u = 5\%$ unmanaged region, $A_{max} = 0.5$ and $slack = 10\%$. Although zcaches have a lower hit latency [21], we simulate the same hit latency for all designs (which is unfair to Vantage, but lets us isolate the improvements due to partitioning).

Fig. 6a shows that, overall, Vantage provides much larger improvements than either way-partitioning or PIPP: a 6.2% geometric mean on average and up to 40%. While Vantage slightly decreases performance for only 4% of the workloads, when using either way-partitioning or PIPP, around 45% of the workloads show worse throughput, often significantly (up to 22% worse for way-partitioning, and 29% worse for PIPP). These workloads already share the cache efficiently with LRU, and partitioning hurts performance by decreasing associativity. Indeed, when using 64-way set-associative caches, way-partitioning and PIPP improve performance for most workloads. This shows the importance of maintaining high associativity, which Vantage achieves.

Fig. 6b compares the throughput of selected workload mixes. Each bar represents throughput improvements of a specific configuration, and there is an additional configuration per set, an un-



(a) Way-partitioning (b) Vantage (c) PIPP

Figure 8: Comparison of way-partitioning, Vantage and PIPP for a specific partition in a 4-core mix. Plots show target partition size (as set by UCP) and actual size for the three schemes. We also show heat maps of the measured associativity CDF on this partition for way-partitioning and Vantage.

partitioned Z4/52 zcache, to determine how much the higher associativity of the zcache is helping Vantage. As we can see, most of the benefits are due to Vantage, not the zcache, though they are complementary. We have selected these workloads to illustrate several points. First, we observe that PIPP sometimes shows significantly different behavior from way-partitioning and Vantage, sometimes outperforming both (`sftn1`), and sometimes doing considerably worse (`ffft4`). PIPP does not use LRU, and performance differences do not necessarily come from partitioning. Nevertheless, both way-partitioning and Vantage can benefit from another replacement policy, as we will see in Section 6.2. Between way-partitioning and Vantage, Vantage achieves higher performance in all except 3 of the 350 workloads. In these rare cases (e.g. `ssst7`), way-partitioning has a slight edge as Vantage cannot partition the whole cache, which affects some mixes, especially those with cache-fitting applications where the miss rate curve decreases abruptly. Way-partitioning and PIPP, however, do significantly worse on associativity-sensitive workloads, such as `fffn7` and `ffnn3`. We can see that, in these cases, the highly-associative zcache has a more noticeable effect in improving Vantage's performance. Finally, mixes `ttnn4`, `sfff6` and `sssf6` illustrate typical behavior of workloads that benefit more from partitioning than from high associativity: both way-partitioning and PIPP improve performance, with PIPP having a slight edge over way-partitioning, while Vantage provides significantly higher throughput.

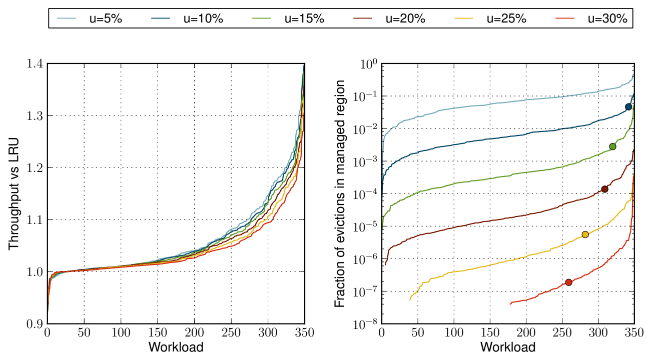
Large-scale configuration: Fig. 7 shows the throughput improvements of different partitioning schemes for the 32-core system, in the same fashion as Fig. 6a. In this configuration, the baseline, way-partitioning and PIPP configurations use a 64-way cache, while Vantage uses *the same* Z4/52 zcache and configuration of the 4-core experiments. Results showcase the scalability of Vantage: while way-partitioning and PIPP degrade performance for most workloads, even with their highly-associative caches, Vantage continues to provide significant improvements on most workloads (8.0% geometric mean and up to 20%) with the same configuration as the 4-core system. While low associativity is again the culprit with way-partitioning, PIPP has much more severe slowdowns (up to $3\times$) because its approach of assigning an insertion position equal to the number of allocated ways causes very low insertion positions with many partitions, leading to high contention at the lower end of the LRU chain and hard to evict dead lines at the higher end.

Partition sizes and associativity: Fig. 8 shows, for each partitioning scheme, the target and actual partition sizes as a function of time for a specific partition and workload mix in the 4-core system. As we can see, way-partitioning and Vantage closely track the target size, while PIPP only approximates it. More importantly, in Vantage the partition is never under its target allocation, while in PIPP the target is often not met (e.g. in some intervals the target size is 2048 lines, but the partition has less than 100). We also observe that with way-partitioning, when the target size is suddenly decreased, reaching the new target allocation can take a significant amount of time (100 Mcycles). This happens because the applications that now own the reallocated ways need to access all the sets and evict all of this partition's lines in those ways. In contrast, Vantage adapts much more quickly, both because of the better location randomization of zcaches and because it works on global, not per-set, allocations. Finally, at times UCP gives a negligible allocation to this partition (128 lines in Vantage, 2048 lines, i.e. 1 way in way-partitioning/PIPP). Vantage cannot keep the partition size that small, so it grows to its minimum stable size, which hovers around 400-700 lines. In this cache, the worst-case minimum stable size is $1/(A_{max}R) = 1/0.5 \cdot 52 = 3.8\%$, i.e. 1260 lines, but replacements caused by other partitions help this partition stay smaller.

Fig. 8 also shows the time-varying behavior of the associativity distributions on way-partitioning and Vantage using heat maps. For each million cycles, we plot the portion of eviction/demotions that happen to lines below a given eviction/demotion priority (i.e. the empirical associativity CDFs). For a given point in time (x-axis), the higher in the y-axis the heat map starts becoming darker, the more skewed the demotion/eviction priorities are towards 1.0, and the higher the associativity. Vantage achieves much higher associativity than way-partitioning: when the partition is large (7 ways at 200-400 Mcycles), way-partitioning gets acceptable associativity, but when given one way, evictions have almost uniformly distributed eviction priorities in $[0, 1]$, and even worse at times (e.g. 700-800 Mcycles). In contrast, Vantage maintains a very high associativity when given a large allocation (at 200-400 Mcycles, the aperture hovers around 3%) because the churn/size ratio is low. Even when given a minimal allocation, demoted lines are uniformly distributed in $[0.5, 1]$, by virtue of the maximum aperture, giving acceptable worst-case associativity.

6.2 Vantage Evaluation

Sensitivity analysis: Fig. 9a shows the performance of Vantage on the 4-core workloads when the size of the unmanaged region changes from 5% to 30% in a Z4/52 zcache. Differences are relatively small, and a size of 5% delivers the highest throughput. Fig. 9b shows what portion of evictions happen from the managed region (because no candidates are from the unmanaged region). For $u = 5\%$, on most workloads 1% to 10% of the evictions come from the managed region. By having a smaller unmanaged region, Vantage can partition a larger portion of the cache, but this slightly degrades isolation. UCP is not very sensitive to strict isolation or partition size control, but benefits from having more space to parti-



(a) Throughput improvement (b) Evictions in managed region

Figure 9: Throughput and fraction of evictions in the managed region when varying the size of the unmanaged region, on a Z4/52 cache with $A_{max} = 0.5$ and $slack = 0.1$.

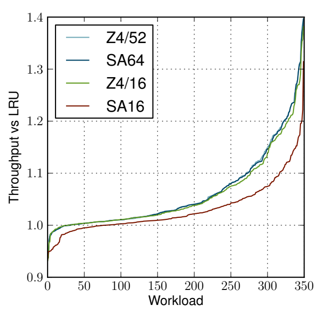


Figure 10: Throughput improvements of Vantage on the 4-core system, using different cache designs.

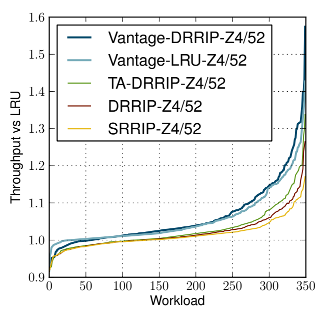


Figure 11: Throughput improvements on the 4-core system using RRIP variants and Vantage.

tion, so 5% works best. Other applications may need better isolation, which would require a larger unmanaged region. We have also studied the sensitivity of Vantage to the maximum aperture, A_{max}, and the $slack$ needed for feedback-based aperture control. With UCP, Vantage is largely insensitive to these parameters: ranges of $5-70\%$ for A_{max} and $slack > 2\%$ work well.

Comparison with analytical models: In Fig. 9b, we have included a round marker at the point where each line crosses the worst-case eviction priority P_{ev}, as predicted by our models (Section 4.3). Most workloads achieve probabilities below the predicted worst-case. For those that exceed it, we have determined that frequent transients are the main culprit: these workloads have fast time-varying behavior, UCP continuously changes target sizes, and the size of the unmanaged region shrinks during transients, increasing evictions. Nevertheless, Fig. 9b shows that we can make evictions in the managed region arbitrarily rare by increasing the size of the unmanaged region, achieving strong isolation guarantees.

We also simulated Vantage in two unrealistic configurations to test that our assumptions hold: first, using feedback-based aperture control but with perfect knowledge of the apertures instead of using setpoint-based demotions, and second, using a *random candidates* cache, an unrealistic cache design that gives truly independent and uniformly distributed candidates. Both design points perform exactly as the practical implementation of Vantage. These results show that our simple controller provides the benefits predicted by the analytical models.

Set-associative and low-associativity caches: Fig. 10 compares Vantage on different cache designs on the 4-core system: our original Z4/52 zcache; a Z4/16 zcache, and 64 and 16-way set-associative caches. Vantage is tuned in each case: the 16-way set-associative and Z4/16 caches use an unmanaged region $u = 10\%$, while the 64-way set-associative and Z4/52 caches use $u = 5\%$. All use $A_{max} = 0.5$ and $slack = 0.1$. As we can see, Vantage works well on set-associative caches and degrades gracefully with lower associativity: the 64-way set-associative cache and Z4/52 zcache achieve practically the same performance, followed very closely by the Z4/16 design, and the 16-way set-associative does sensibly worse, although still significantly better than either way-partitioning or PIPP with a 16-way cache (Fig. 6a). These results show that, although Vantage works best and provides stronger isolation with zcaches, it is practical to use with traditional set-associative caches.

Comparison with alternative replacement policies: We have used LRU so far because the partitioning schemes we compare Vantage with are based on LRU. However, much prior work has improved on LRU, both in performance and implementation cost. The RRIP family of replacement policies [12] is one such example. They include scan-resistant SRRIP, thrash-resistant BRRIP, and scan and thrash-resistant DRRIP, which uses set dueling to choose between SRRIP and BRRIP dynamically. Additionally, TA-DRRIP enhances performance in shared caches by using TADIP's thread-aware set dueling mechanism on DRRIP [11]. These policies do not require set ordering, so they are trivially applicable to zcaches and Vantage. Fig. 11 compares the performance achieved by using these policies with two variants of Vantage, one using LRU and other using DRRIP. All RRIP variants use a 3-bit re-reference prediction value (RRPV) in each tag instead of 8-bit LRU timestamps. In Vantage-DRRIP, we have a per-partition setpoint RRPV instead of a setpoint LRU timestamp, and do not age lines from partitions below their target size, but otherwise the scheme works as in [12]. Additionally, for Vantage-DRRIP to work, we have to (1) modify UCP's UMON-DSS mechanism to work with RRIP instead of LRU, and (2) provide a way to decide between SRRIP and BRRIP. To achieve this, UMON-DSS is modified to maintain RRIP chains instead of LRU chains (i.e. lines are ordered by their RRPVs), and one half of the UMON sets use SRRIP, while the other half use BRRIP. Each time partitions are resized, the best of the two policies is chosen for each partition and used in the next interval. Because the decision of whether to use SRRIP or BRRIP is done per partition, Vantage-DRRIP is automatically thread-aware.

Fig. 11 shows that Vantage-LRU outperforms all RRIP variants, and Vantage-DRRIP further outperforms Vantage-LRU, although by a small amount: the geometric means over all benchmarks are 2.5% for TA-DRRIP, 6.2% for Vantage-LRU, and 6.8% for Vantage-DRRIP. We can extract three conclusions from these experiments. First, Vantage can be easily modified to work with alternative replacement policies. Second, Vantage is still beneficial when using a better replacement policy. Moreover, partitioning has several other uses beyond improving miss rates, as explained in Section 1. Finally, we note that these results are preliminary, as there may be better ways than using UMON to decide partition sizes and choosing the replacement policy. We defer a more detailed exploration of these issues to future work.

7. CONCLUSIONS

We have presented Vantage, a scalable and efficient scheme for fine-grained cache partitioning. Vantage works by matching the insertion (churn) and demotion rates of each partition, thus keeping their sizes approximately constant. It partitions most of the cache, and uses the unmanaged region to eliminate inter-partition interference and achieve a simple implementation. Vantage is derived from analytical models, which allow it to provide different degrees of isolation by varying the size of the unmanaged region: a small unmanaged region (5%) suffices to provide moderate isolation, while a larger region (20%) can provide strong isolation and eliminate inter-partition interference. Thus, Vantage satisfies the needs of applications with different isolation requirements, all while maintaining a good associativity per partition regardless of the number of partitions. Under UCP, Vantage outperforms existing partitioning schemes on small-scale CMPs, but most importantly, it continues to deliver the same benefits on CMPs with tens of threads, where previous schemes fail to scale.

8. ACKNOWLEDGEMENTS

We sincerely thank Woongki Baek, Asaf Cidon, Christina Delimitrou, Jacob Leverich, David Lo, Tomer London, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. Daniel Sanchez was supported by a Hewlett-Packard Stanford School of Engineering Fellowship.

9. REFERENCES

[1] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proc. of the 9th annual ACM Symposium on Theory of Computing*, 1977.

[2] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006.

[3] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proc. of the 37th annual Design Automation Conf.*, 2000.

[4] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. of the 37th annual Design Automation Conf.*, 2000.

[5] H. Cook, K. Asanović, and D. A. Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. Technical report, EECS Department, U. of California, Berkeley, 2009.

[6] G. Gerosa et al. A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile PCs in 45nm hi-K metal gate CMOS. In *IEEE Intl. Solid-State Circuits Conf.*, 2008.

[7] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. *ACM SIGARCH Computer Architecture News*, 35(1), 2007.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture*. 2004.

[9] L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of the 15th intl. conf. on Parallel Architectures and Compilation Techniques*, 2006.

[10] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proc. of the 18th annual intl. conf. on Supercomputing*, 2004.

[11] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008.

[12] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010.

[13] N. Kurd et al. Westmere: A family of 32nm IA processors. In *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[14] 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 *Proc. of the 14th IEEE intl. symp. on High Performance Computer Architecture*, 2008.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2005.

[16] V. Nagarajan and R. Gupta. ECMon: exposing cache events for monitoring. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.

[17] C. Percival. Cache missing for fun and profit. *BSDCan*, 2005.

[18] M. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. In *Proc. of the 10th intl. symp. on High Performance Computer Architecture*, 2009.

[19] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006.

[20] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[21] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010.

[22] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[23] J. Shin et al. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *Intl. Solid-State Circuits Conf.*, 2010.

[24] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc of the 8th IEEE intl. symp. on High Performance Computer Architecture*, 2002.

[25] K. Varadarajan, S. 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 *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006.

[26] C. Wu and M. Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. In *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, 2008.

[27] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.