Scalable and Efficient Fine-Grained Cache Partitioning with Vantage

The Vantage cache-partitioning technique enables configurability and quality-of-service guarantees in large-scale chip multiprocessors with shared caches. Caches can have hundreds of partitions with sizes specified at cache line granularity, while maintaining high associativity and strict isolation among partitions.

Daniel Sanchez
Christos Kozyrakis
Stanford University

Shared caches are pervasive in chip multiprocessors (CMPs). In particular, CMPs almost always feature a large, fully shared last-level cache (LLC) to mitigate the high latency, high energy, and limited bandwidth of main memory. A shared LLC has several advantages over multiple, private LLCs: it increases cache utilization, accelerates intercore communication (which happens through the cache instead of main memory), and reduces the cost of coherence (because only non-fully-shared caches must be kept coherent). Unfortunately, these advantages come at a significant cost. When multiple applications share the CMP, they suffer from interference in shared caches. This causes large performance variations, precluding quality-of-service (QoS) guarantees, and can degrade cache utilization, hurting overall throughput. Interference can cause performance variations of over 3× in systems with few cores,[1] and is a growing concern due to the increasing number of cores per chip and the emergence of cloud computing.

We can eliminate interference by using cache partitioning to divide the cache explicitly among competing workloads. A cache-partitioning solution has two components: an allocation policy that decides each partition's size to achieve a specific objective (maximizing throughput, improving fairness, meeting QoS requirements, and so on), and a partitioning scheme that enforces those sizes. Although allocation policies are generally simple and efficient,[1-3] current partitioning schemes have serious drawbacks. In this work, we focus on the partitioning scheme.

Ideally, a partitioning scheme should satisfy several desirable properties. First, it should be scalable and fine-grained, capable of maintaining many fine-grained partitions (for instance, hundreds of partitions of tens or hundreds of lines each). Second, it should maintain strict isolation among partitions, with no reduction of cache performance (that is, without hurting associativity or replacement policy performance). Third, it should be dynamic, allowing quick creation, deletion, and resizing of partitions. Finally, it should be simple to implement.

Unfortunately, prior partitioning schemes fail to meet these properties. Several schemes

Published by the IEEE Computer Society

Related Work on Cache Partitioning

Partitioning requires an allocation policy to decide the number and size of partitions, and a partitioning scheme to enforce them. This article focuses on the latter. There are generally two approaches for partitioning a cache: *strict partitioning* by restricting line placement and *soft partitioning* by controlling the insertion and/or replacement policies.

Strict partitioning

Schemes with strict guarantees rely on restricting the locations where a line can reside depending on its partition. Way partitioning divides the cache by ways, restricting insertions from each partition to its assigned subset of ways.[1] This scheme is simple but has several problems: partitions are coarsely sized (in multiples of way size), and partition associativity is proportional to way count, sacrificing performance for isolation. Way partitioning needs significantly more ways than partitions to work well, so it is not scalable.

To avoid losing associativity, reconfigurable caches[2] and molecular caches[3] partition the cache by sets instead of ways. However, these approaches require significant changes to cache arrays, and they must flush or move data when resizing partitions. Most importantly, they only work with fully disjoint address spaces, which are not common in chip multiprocessors (CMPs), because even processes with separate address spaces share library and operating system code and data.

Finally, virtual memory and page coloring can constrain the physical pages of a process to map to specific cache sets.[4] Although this scheme doesn't require hardware support, it is limited to coarse-grained partitions, it is incompatible with superpages and caches indexed using hashing (common in modern processors), and repartitioning requires costly recoloring (copying) of physical pages and thus must be done infrequently.[4]

Soft partitioning

Other schemes partition a cache approximately by modifying the insertion or replacement policies. These schemes avoid some of the issues of restricting line placement, but they provide limited control over partition sizes and interference. They're useful when approximate partitioning is sufficient, but not when strict guarantees are required.

In decay-based replacement policies, lines from different partitions age at different rates; adjusting these rates provides some control over partition sizes.[5] Promotion-insertion pseudo-partitioning (PIPP) assigns each partition a different insertion position in the least recently used (LRU) chain and slowly promotes lines on hits (for example, promoting one position per hit rather than moving the line to the head of the LRU chain).[6] With an additional mechanism to restrict pollution of thrashing applications, PIPP approximately attains the desired partition sizes. PIPP is co-designed to work with utility-based cache partitioning as the allocation policy and, due to the limited size of LRU chains, is not scalable.

References

1. D. Chiou et al., ''Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches,'' *Proc. 37th Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 416-419.
2. P. Ranganathan, S. Adve, and N.P. Jouppi, ''Reconfigurable Caches and Their Application to Media Processing,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM, 2000, pp. 214-224.
3. K. Varadarajan et al., ''Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2006, pp. 433-442.
4. J. Lin et al., ''Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems,'' *Proc. 14th Int'l Symp. High-Performance Computer Architecture* (HPCA 08), IEEE CS, 2008, pp. 367-378.
5. C.-J. Wu and M. Martonosi, ''A Comparison of Capacity Management Schemes for Shared CMP Caches,'' *Proc. 7th Ann. Workshop Duplicating, Deconstructing, and Debunking* (WDDD 08), IEEE CS, 2008; http://www.princeton.edu/~carolewu/WDDD08-CJW.pdf.
6. Y. Xie and G.H. Loh, ''PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture* (ISCA 09), ACM, 2009, pp. 174-183.

partition the cache strictly by restricting line placement. For example, *way partitioning* assigns a subset of the ways to each partition. However, these schemes are limited to few coarse-grained partitions, and partitioning often hurts performance (for instance, by degrading associativity). Other schemes modify the replacement policy to provide some control over partition sizes, but this partitioning is approximate and often hurts replacement policy performance. Most importantly, current partitioning schemes aren't scalable or fine-grained (for more information, see the ''Related Work on Cache Partitioning'' sidebar). Shared LLCs are already used in commercial large-scale CMPs with hundreds of threads and cores,[4,5] and are included in thousand-core research prototypes,[6] thus stressing the need for scalable partitioning.

In our paper presented at the 38th Annual International Symposium on Computer

Architecture (ISCA 2011),[7] we introduced Vantage, a partitioning scheme that overcomes the drawbacks of prior techniques. Vantage can maintain hundreds of partitions defined at cache line granularity, provides strict isolation among partitions, maintains high cache performance, and is simple to implement, working with conventional cache arrays and requiring minimal overhead. Unlike other techniques, Vantage is designed and fully characterized by accurate analytical models. Vantage doesn't physically restrict line placement, side-stepping the problems of previous strict partitioning techniques, and it leverages its analytical models to provide strict guarantees on partition sizes and interference independently of workload behavior. Thanks to these features, Vantage enables performance isolation and QoS in current and future large-scale CMPs, and can be used for several other purposes, such as cache-pinning critical data or implementing flexible local stores through application-controlled partitions.

Vantage overview

Vantage achieves its strict analytical guarantees by using caches with high associativity and good hashing, such as skew-associative caches[8] and zcaches.[9] These caches remove the influence of the workload's access pattern from cache performance, and can be characterized accurately via workload-independent analytical models. These caches have a surprising property: they can avoid evictions to a large subset of the lines with very high probability. For example, we can select an arbitrary 90 percent of cache lines, and constrain practically all evictions to the remaining 10 percent. Vantage exploits this property by reserving a small portion of the cache, called the *unmanaged region*, leaving it unpartitioned, and partitioning the remaining portion (for example, 90 percent), which is effectively pinned to the cache. So that the unmanaged region's size can be maintained, lines are first inserted into their partition, eventually *demoted* to the unmanaged region, and evicted from there. The unmanaged region is still used, acting as a victim cache for the partitions, but it isn't partitioned.

Vantage controls partition sizes in a scalable fashion. On each miss, Vantage evicts one of the replacement candidates from the unmanaged region and inserts the incoming line into its partition. Additionally, to keep partition sizes constant, Vantage would need to perform a demotion from the same partition as the incoming line. However, this approach is not scalable: with many partitions (for instance, 100 partitions and 64 candidates), the current set of candidates most likely doesn't have a line from the inserting partition. Nevertheless, the set of replacement candidates often includes good candidates from other partitions (that is, lines that the owning partition would have to demote anyway). To be scalable, Vantage relaxes this restriction. It allows partitions to grow slightly over their target sizes (borrowing space from the unmanaged region, not from one another), and controls sizes by matching the rate at which it demotes lines from each partition to match that partition's insertion rate. Using analytical models, we show that this approach controls partition sizes accurately and maintains high associativity in all cases.

Finally, the analytical models use several metrics that are expensive to compute in practice. To simplify the design, we use negative feedback to derive these quantities cheaply and implicitly, achieving an implementation with minimal overhead that maintains analytical guarantees.

Vantage techniques

Vantage addresses the drawbacks of prior techniques by leveraging efficient highly associative caches that provide analytical guarantees, dividing the cache into a managed and an unmanaged region, and logically partitioning the managed region through churn-based management.

Efficient highly associative caches

Vantage achieves strict analytical guarantees with highly associative caches, such as skew-associative caches[8] and zcaches.[9] Skew-associative caches index each way with a different hash function, spreading out conflicts. Zcaches enhance skew associativity with a replacement process that obtains an arbitrarily large number of candidates with only a few ways. We focus on zcaches because they achieve high associativity far

more cheaply (for example, 64-way associativity with the area, latency, and hit energy of a 4-way cache, and miss energy similar to a 64-way cache).

Zcaches provide high associativity by increasing the number of replacement candidates, but they keep the number of positions where a block can be located as low as the number of ways. To evaluate this approach, we developed a framework to compare associativity across cache designs independently of the replacement policy.[9] We realize that, conceptually, all replacement policies simply rank lines globally by how much they would like to evict them. By convention, we express this rank through a uniformly distributed number in the range [0, 1], the eviction priority e. For example, with a least recently used (LRU) policy, the most recently used line in the entire cache would have an eviction priority of $e = 0.0$, the least recently used line would have $e = 1.0$, and every other line in the cache would be somewhere in between. On each miss, the cache array retrieves a given number of replacement candidates and evicts the one with the highest eviction priority. We then characterize associativity as a probability distribution: the distribution of the eviction priorities of evicted lines. Intuitively, the higher the associativity, the higher these eviction priorities are, and the more skewed the distribution is toward 1.0. This model decouples replacement policy performance from cache associativity, letting us study associativity in isolation.

In zcaches, the set of replacement candidates examined on every eviction is statistically very close to a uniform random selection of lines, independent of the workload's access pattern or the replacement policy.[9] Hence, their associativity distribution can be derived analytically. If the array gives R uniformly distributed replacement candidates, its cumulative distribution function (CDF) is[9]

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

Figure 1 plots this distribution for different values of R. Note that associativity depends on the number of replacement candidates, not the number of ways. Moreover, Figure 1 shows that the probability of evicting lines with low eviction priority quickly becomes negligible. For example, with $R = 64$, the probability of evicting a line with $e < 0.8$ is $F_A(0.8) = 10^{-6}$. Hence, by controlling how lines are ranked, we can guarantee that they won't be evicted with very high probability. This doesn't apply to set-associative caches, which perform significantly worse.[9]

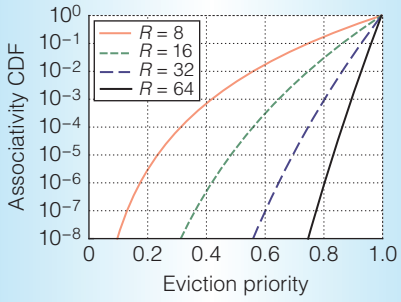


Figure 1. Associativity cumulative distribution function (CDF) ($F_A(x) = x^R$, $x \in [0, 1]$) for $R = 8$, 16, 32, and 64 uniformly distributed replacement candidates, in a semilogarithmic scale. With a large number of candidates, the probability of evicting lines with a low eviction priority quickly becomes negligible.

We used the analytical guarantees of zcaches to design Vantage. We can also use Vantage with set-associative arrays, with some performance degradation and no analytical guarantees.[7] Zcaches, like skew-associative caches, have no sets, so they cannot use replacement policies that rely on set ordering. Nevertheless, most replacement policies can be implemented efficiently. For example, an LRU policy could be implemented with 8-bit coarse-grained time stamps.[9] To keep Vantage independent of the replacement policy, we design it assuming that we know every candidate's eviction priority. Although tracking eviction priorities would be expensive in practice, we will show that it is unnecessary if we adapt Vantage to the specific replacement policy used.

Managed-unmanaged region division

Vantage divides the cache into two logical regions—a managed region and an unmanaged region—by simply tagging each line as either managed or unmanaged. On an

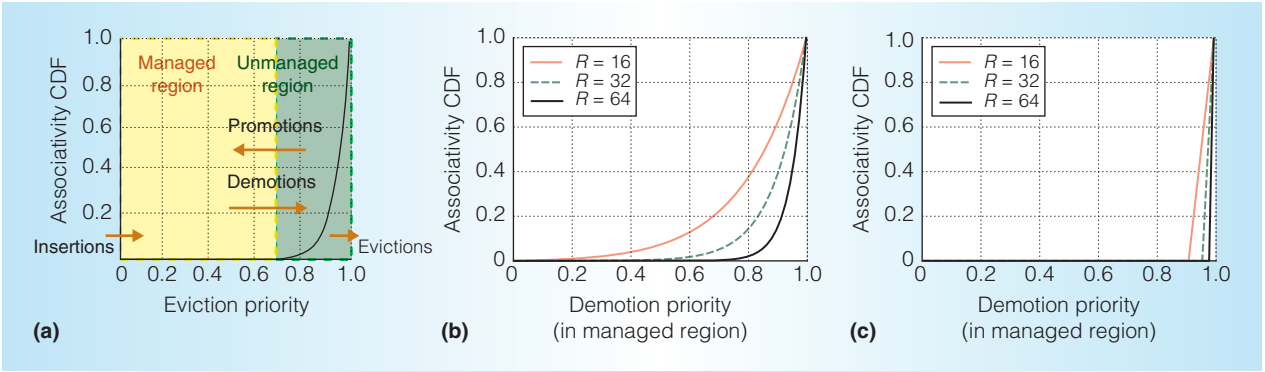


Figure 2. Managed-unmanaged region division (assuming 30 percent of the cache is unmanaged): managed-unmanaged region division for a cache with $R = 16$ replacement candidates, and flows between regions (a); associativity CDF in the managed region when doing exactly one demotion per eviction (b); and associativity CDF in the managed region when doing one demotion per eviction on average (c).

eviction, Vantage always prioritizes unmanaged lines for eviction over managed lines. The unmanaged region is sized to capture most evictions, making evictions in the managed region negligible. Vantage maintains region sizes by controlling the flow of lines between both regions.

Figure 2a illustrates this setup. It shows the associativity distribution of a cache with $R = 16$ candidates, divided into the managed and unmanaged regions, and the flow of lines between the two. To make evictions in the managed region negligible ($\simeq 10^{-3}$ probability), the unmanaged region is sized to 30 percent of the cache. Caches with $R > 16$ require a smaller unmanaged region. Incoming lines are inserted into the managed region, eventually demoted to the unmanaged region, and either evicted from there or promoted if they get a hit. Promotions and demotions don't move the line; they just change its tag. The unmanaged region acts as a victim cache for the managed region. Evicting a line requires that it be demoted first (except for the rare case when all candidates come from the managed region).

In the rest of the discussion, we ignore the flow of promotions to simplify the analysis. Promotions are rare compared to evictions, so we treat them as a small modeling error, and we address this error when implementing the controller. Thus, we can perform demotions at replacement time only—that is, together with evictions.

To keep the sizes of both regions constant, Vantage would have to demote one line on each replacement. However, to maintain region sizes under control, it suffices to demote one candidate per eviction on average. For example, some evictions might not yield any high-priority candidates from the managed region, whereas others might find two or more. To implement this scheme, Vantage selects a threshold value, which we call *aperture* (A), and demotes every candidate over that threshold. For example, if $A = 0.05$, Vantage demotes every candidate that is in the top 5 percent of eviction priorities ($e \geq 1 - A = 0.95$). We denote the fractional sizes of the managed and unmanaged regions as m and u (for example, in Figure 2a, $m = 0.7$ and $u = 0.3$). On average, $R \cdot m$ of the candidates are from the managed region, so maintaining the sizes requires an aperture of $A = \frac{1}{R \cdot m}$.

We can derive the associativity distribution for demotions much like we derive the associativity distribution for evictions (see the full paper for the derivations[7]). Figure 2 shows the distribution for demotions, both when one candidate is always demoted per eviction (Figure 2b) and when one candidate is demoted on average (Figure 2c). Demoting one candidate on average significantly improves associativity. For example, with $R = 16$ candidates, only lines with eviction priority $e > 0.9$ are demoted. Meanwhile, when always demoting one line per

eviction, 60 percent of the demotions occur to lines with $e < 0.9$.

Churn-based management

Vantage logically partitions the managed region. We have P partitions of target sizes T_1, ..., T_P, so that $\sum_{i=1}^{P} T_i = m$ (that is, partition sizes are expressed as a fraction of total cache size). The allocation policy (for example, utility-based partitioning) sets these target sizes. 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 each partition's actual size close to its target by matching its demotion rate with its churn. Rather than 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 other partitions' lines.

Apertures depend on the sizes and churns of all partitions. Intuitively, a partition with an above-average churn needs a larger aperture, because its lines must be demoted at a higher frequency; a partition with a below-average size also needs a larger aperture, because replacement candidates from that partition will be found more rarely. In general, out of the $R \cdot m$ candidates per demotion that fall into the managed region, a fraction, $\frac{S_i}{\sum_{k=1}^{P} S_k}$, are from partition i, and we need to demote its lines at a fractional rate of $\frac{C_i}{\sum_{k=1}^{P} C_k}$. Therefore,

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

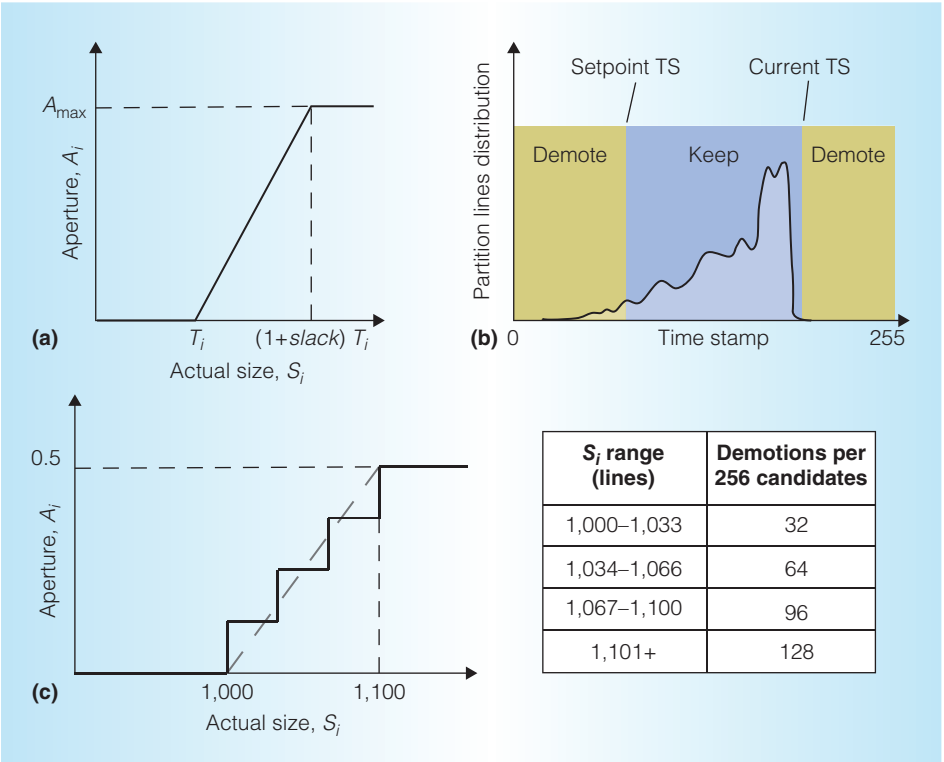
If a partition has a large C_i/S_i ratio (for example, a one-line partition with very frequent misses), adjusting its aperture might not be enough to maintain its size, even if we are willing to sacrifice associativity by setting the aperture to 1.0 (demoting every candidate from this partition). Because completely sacrificing associativity is undesirable, we set a maximum aperture $A_{\max}$ (for instance, $A_{\max} = 0.4$ yields an associativity similar to a 16-candidate cache). If a partition requires $A_i > A_{\max}$, we set $A_i = A_{\max}$ and let it grow over its target. At first glance, this could lead to borrowing too much space from the unmanaged region, making it too small and leading to frequent forced evictions from the managed region. However, using analytical models, we find that the maximum amount of space that all partitions can borrow from the unmanaged region is $\frac{1}{A_{\max} R}$ of the cache (see the full paper for the derivation[7]).

Intuitively, this happens for two reasons. First, a partition with aperture $A_{\max}$ never grows beyond $\frac{1}{A_{\max} R}$, even in the worst case (that is, if all misses come from this partition). Second, when multiple partitions need $A_i > A_{\max}$, they help one another demote lines, so the sum of the growths over their targets never exceeds $\frac{1}{A_{\max} R}$. This quantity is small and independent of the number of partitions. Therefore, sizing the unmanaged region with an extra $\frac{1}{A_{\max} R}$ guarantees negligible managed-region evictions, even in the worst case. For example, if the cache has $R = 52$ candidates, with $A_{\max} = 0.4$, we need to assign an extra $\frac{1}{0.4 \times 52} = 4.8$ percent to the unmanaged region.

Vantage cache controller

In theory, we could implement Vantage using the previous analysis alone. However, three reasons make this impractical. First, obtaining the apertures using Equation 1 is too computationally intensive and requires estimating the churns. Second, our analysis makes two approximations: replacement candidates are not exactly independent and uniformly distributed (although they are close[9]), and we have ignored promotions, which have no matching demotion. Without correction, these modeling errors would cause partition sizes to slowly drift away from their targets. Third, although eviction priorities are a useful conceptual tool, tracking them would be expensive.

We address these issues with two techniques. First, *feedback-based aperture control* enables a simple and robust controller that finds apertures implicitly using negative



S_i range (lines)	Demotions per 256 candidates
1,000–1,033	32
1,034–1,066	64
1,067–1,100	96
1,101+	128

Figure 3. Feedback-based aperture control and setpoint-based demotions: linear transfer function used in feedback-based aperture control (a); setpoint-based demotions, in which candidates are selected below the setpoint (in modulo arithmetic) (b); and four-entry demotion thresholds lookup table for 1,000-line partition with 10 percent slack (c). (TS: time stamp.)

feedback instead of calculating them explicitly. Second, *setpoint-based demotions* lets us demote the lines below the aperture without knowing their eviction priorities. Using these techniques, we design a controller with minimal overhead and complexity that still maintains analytical guarantees.

Feedback-based aperture control

We can derive apertures cheaply and implicitly using negative feedback. We let partitions slightly outgrow their target sizes, borrowing space from the unmanaged region, and we adjust their apertures according to how much they outgrow them. Specifically, we derive each aperture A_i as a linear transfer function of actual size S_i, shown in Figure 3a. Partitions below their target T_i have a zero aperture. Over T_i, the aperture increases linearly, until S_i reaches $(1 + slack)\,T_i$, at which point A_{max} is used.

This is a classic application of negative feedback: an increase in size causes a larger aperture, attenuating the size increase. The *slack* parameter modulates the feedback loop: a larger *slack* reduces the effect of instantaneous size variations, causing more stable apertures, but requires a larger unmanaged region, as partitions outgrow their target sizes further. Using analytical models, we find that partitions borrow an additional $\frac{slack}{A_{max} R}$ fraction of the cache from the unmanaged region. This is fairly small; for example, with $R = 52$, $slack = 0.1$, and $A_{max} = 0.4$, this fraction is 0.48 percent of the cache.

Setpoint-based demotions

Using setpoint-based demotions lets us demote the lines below the aperture without tracking eviction priorities. This requires taking the replacement policy into account. Here, we describe the implementation of setpoint-based demotions for an LRU policy, but the scheme is extensible to other policies, such as least frequently used (LFU) and re-reference interval prediction (RRIP).[7]

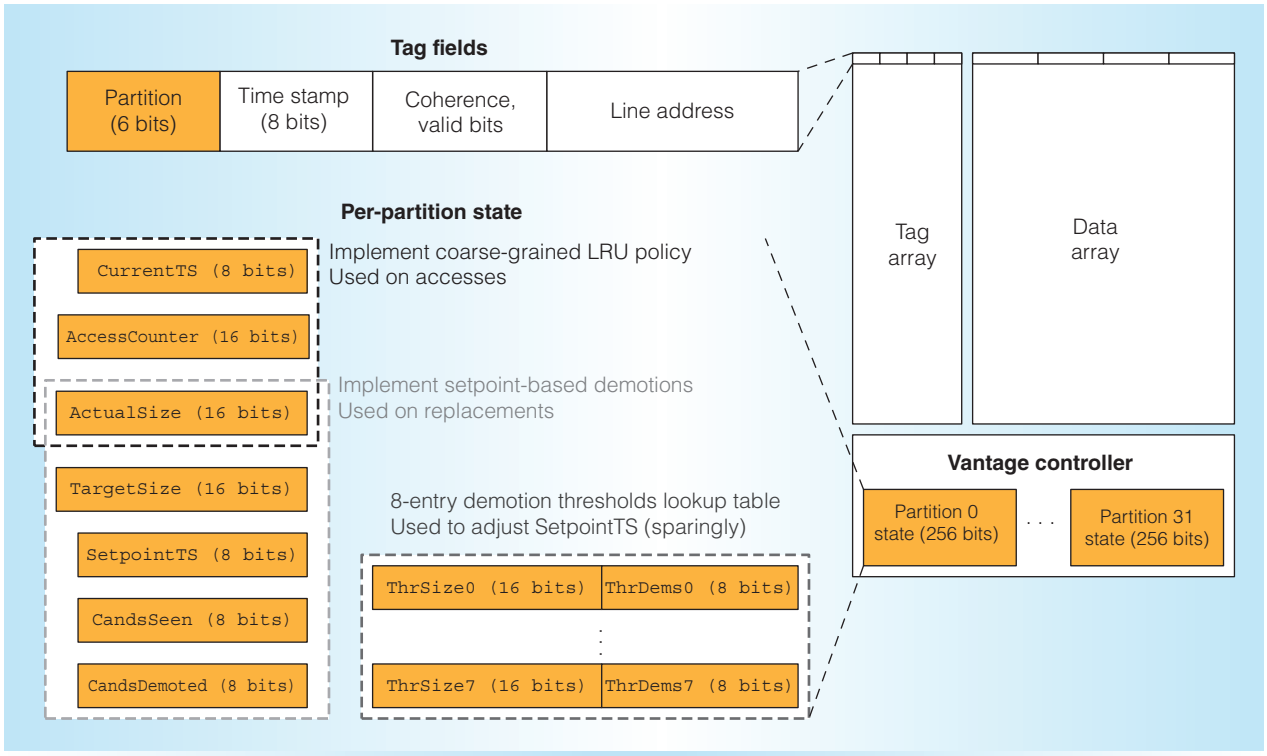


Figure 4. State required to implement Vantage, including tag fields and per-partition registers. Additional state over an unpartitioned baseline is shown in shaded boxes. Each field or register shows its size in bits.

The per-partition replacement policy implements an LRU policy using coarse-grained time stamps.[9] Each partition has a *current time stamp* counter that is incremented every k_i accesses, and accessed lines are tagged with the current time stamp value. We choose 8-bit time stamps with k_i = 1/16 of the partition size, making wraparounds rare.

To perform demotions, we choose a *setpoint time stamp*, and all candidates below it (in modulo 256 arithmetic) are demoted if the partition exceeds its target size, as Figure 3b shows. Vantage adjusts the setpoint by counting both the candidates seen (`CandsSeen`) and demoted (`CandsDemoted`) for each partition. When the controller has seen a reasonably large number of candidates from a partition, it adjusts the setpoint to keep that `CandsDemoted`/`CandsSeen` ratio as close to the aperture as possible. Therefore, the setpoint is increased if the ratio is over the aperture (covering fewer lines), and decreased if below. Both counters are then reset. In our experiments, we've found that performing this procedure every 256 candidates seen works well. Additionally, we increase the setpoint every time the current time stamp is increased (that is, every k_i accesses), so the distance between both counters remains constant.

Finally, we can completely avoid computing apertures, simplifying the implementation even further. We use a small eight-entry demotion thresholds lookup table that stores the `CandsDemoted` thresholds for different size ranges. Figure 3c shows a concrete example for a partition with T_i = 1,000 lines and *slack* = 10 percent. For example, when 256 candidates from this partition have been seen, if its size is anywhere between 1,034 and 1,066 lines, and more or less than 64 candidates have been demoted, the setpoint is incremented or decremented, respectively. This per-partition table is filled at resize time, and is used every 256 candidates seen.

Putting it all together

With these techniques, Vantage can be implemented with minimal state and logic overheads. Figure 4 shows the state required.

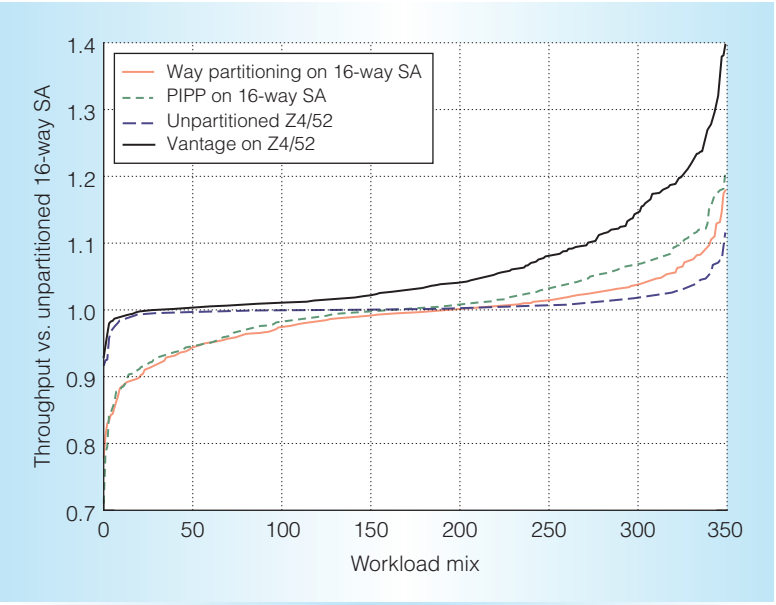


Figure 5. Throughput improvements over an unpartitioned 16-way set-associative (SA) Level-2 (L2) cache with a least recently used (LRU) policy, on the four-core configuration. Whereas way partitioning and promotion-insertion pseudo-partitioning (PIPP) often degrade throughput, Vantage improves throughput significantly for practically all workloads using a 4-way zcache with 52 replacement candidates (Z4/52).

Each line must be tagged with its partition ID, adding a few bits per tag. Additionally, Vantage must maintain a few per-partition registers to implement the replacement policy and setpoint-based demotions. These registers require about 256 bits per partition. Dividing the 8-Mbyte, four-bank LLC that we evaluate in 32 partitions requires 96 Kbytes for the per-tag ID, and 4 Kbytes for the per-partition state, a 1.1 percent overhead.

Logic overhead in Vantage is also minimal. The algorithm requires counter updates and comparisons on either 8- or 16-bit registers, so a few narrow adders and comparators suffice to implement it. Vantage runs on misses, so hits are unaffected, and its logic is off the critical path.

Comparison of partitioning schemes

We simulate four- and 32-core CMPs with private L1 caches and a shared L2 cache. We compare Vantage, way partitioning,[10] and promotion-insertion pseudo-partitioning (PIPP),[11] with utility-based cache partitioning[2] (UCP) as the partitioning policy. UCP allocates more space to the cores that can use it better, with the goal of maximizing system throughput. Therefore, we use throughput as our evaluation metric. We use 350 multiprogrammed mixes of four and 32 applications each, using SPEC CPU2006 benchmarks. The complete description of our methodology is available in the full paper.[7]

Small-scale configuration

Figure 5 compares performance across the 350 workload mixes on the four-core system. Each line shows the throughput ($\sum IPC_i$) of a different scheme, normalized to a 16-way set-associative cache using an LRU policy. For each line, workloads (x-axis) are sorted by the improvement achieved. Way partitioning and PIPP use a hashed 16-way set-associative cache, whereas Vantage uses a 4-way zcache with 52 replacement candidates (Z4/52), with a $u = 5$ percent unmanaged region, $A_{max} = 0.5$, and $slack = 0.1$.

Figure 5 shows that, although way partitioning and PIPP can improve throughput, they also degrade it, often significantly, for about 45 percent of the workloads. These workloads already share the cache efficiently with the LRU policy, and partitioning hurts performance by decreasing associativity. When using a 64-way set-associative cache instead (not shown), this degradation practically disappears. Vantage uses a more associative (but cheaper) zcache, which already improves throughput for most workloads, as Figure 5 shows. However, most of the throughput improvements come from Vantage, which achieves a 6.2 percent geometric mean and up to 40 percent speedups, and improves performance for practically all workloads because it maintains high associativity.

Large-scale configuration

Figure 6 shows the throughput improvements of different partitioning schemes for the 32-core system in the same fashion as Figure 5. The baseline, way partitioning, and PIPP schemes use a 64-way cache, whereas Vantage uses the same Z4/52 zcache and configuration as in the four-core experiments. Results showcase Vantage's scalability. Whereas way partitioning and PIPP degrade performance for most workloads,

even with their barely implementable 64-way caches, Vantage still provides significant improvements on most workloads (an 8.0 percent geometric mean and up to 20 percent) with the same 4-way cache as in the four-core system. (For more details regarding why Vantage outperforms previous partitioning schemes, see the ''Partition Sizes and Associativity'' sidebar.)

We have presented Vantage, a cache-partitioning scheme that addresses the drawbacks of prior techniques. Vantage addresses the scalability issues of other techniques, where implementing more than a few partitions significantly degrades performance. With 100-core CMPs that share an LLC already on the market, Vantage addresses a fundamental problem. Vantage's analytically driven design methodology is also worth noting. Architects typically rely on empirical observation and intuition to optimize the common-case behavior. Analytical models are derived a posteriori, if ever, and are often approximate. Instead, we begin with a component that is accurately characterized by analytical models (zcache) and use them to drive the design. This leads to a solution that is fully characterized by analytical models, so it not only performs better in the common case but also can make strict guarantees in all scenarios, which is crucial to providing performance isolation. We hope that the design methodology and insights of Vantage inspire architects to design future systems that can provide strict QoS, system-level isolation, predictability, and efficient use of resources. MICRO

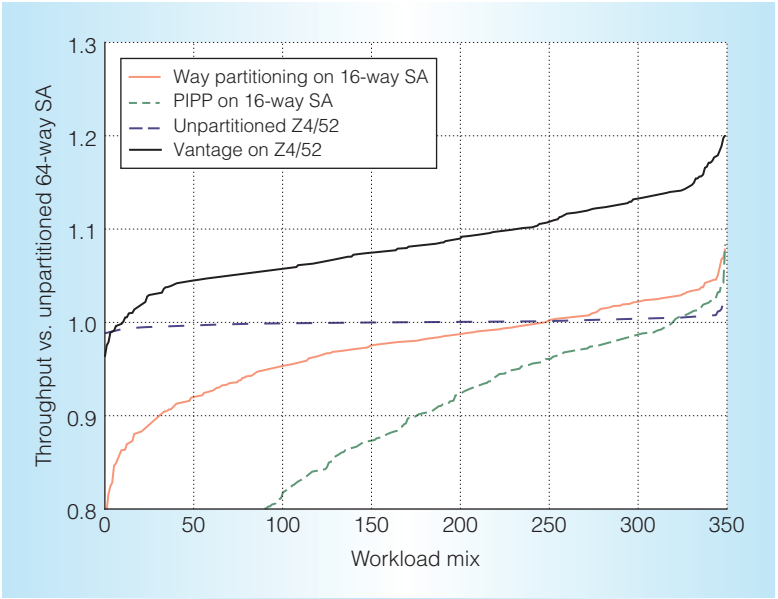


Figure 6. Throughput improvements over an unpartitioned 64-way set-associative L2 with an LRU policy, on the 32-core configuration. Way partitioning and PIPP degrade throughput for most workloads, despite the higher associativity, whereas Vantage improves performance using the same associativity as in the four-core system.

Acknowledgments

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

References

1. L.R. Hsu et al., ''Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource,'' *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), ACM, 2006, pp. 13-22.
2. M.K. Qureshi and Y.N. Patt, ''Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2006, pp. 423-432.
3. G.E. Suh, S. Devadas, and L. Rudolph, ''A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,'' *Proc. 8th Int'l Symp. High-Performance Computer Architecture* (HPCA 08), IEEE CS, 2002, pp. 117-128.
4. J.L. Shin et al., ''A 40nm 16-Core 128-Thread CMT SPARC SoC Processor,'' *Proc. IEEE Int'l Solid-State Circuits Conf.* (ISSCC 10), IEEE Press, pp. 98-99.
5. ''TILE-Gx 3000 Series Overview,'' Tilera, 2011; http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx%203000%20Series%20Brief.pdf.
6. J.H. Kelm et al., ''Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture* (ISCA 09), ACM, 2009, pp. 140-151.

Partition Sizes and Associativity

For each partitioning scheme that we compared, Figure A shows the target and actual partition sizes as a function of time for a specific partition and execution in the four-core system. We can make several observations. First, Vantage works with fine-grained allocations, whereas way partitioning and promotion-insertion pseudo-partitioning (PIPP) work with coarse-grained allocations (one way, 2,048 lines). Second, way partitioning and Vantage closely track target size, whereas PIPP only approximates it. Third, way partitioning has far longer transients: when target allocations change, reaching the new allocations can take a long time (100 Mcycles). This happens because the new owner of the reallocated ways must access all their sets and evict the previous owner's lines. In contrast, Vantage has far faster transients, because it works on global, not per-set, allocations. Finally, in Vantage, utility-based cache partitioning (UCP) sometimes gives a very small target allocation (128 lines). Vantage can't keep the partition that small, so it grows to its minimum stable size, which hovers at around 400 to 700 lines. In this cache, the worst-case minimum stable size is $\frac{1}{A_{\max} R} = \frac{1}{0.5 \times 52} = 3.8$ percent (1,260 lines), but replacements caused by other partitions help keep this partition smaller.

Figure B shows the time-varying behavior of the associativity distributions on way partitioning and Vantage using heat maps. For each million cycles, we plot the empirical associativity cumulative distribution functions (CDFs)—that is, the fraction of evictions or demotions that happen to lines below a given eviction or demotion priority. Vantage achieves far higher associativity than way partitioning. With a large allocation (seven ways at 200 to 400 Mcycles), way partitioning achieves acceptable associativity. However, when given one way, evictions have almost uniformly distributed priorities in [0, 1], and even worse at times (for instance, 700 to 800 Mcycles). In contrast, Vantage maintains very high associativity with a large allocation (at 200 to 400 Mcycles, the aperture hovers around 3 percent) 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 an acceptable worst-case associativity.

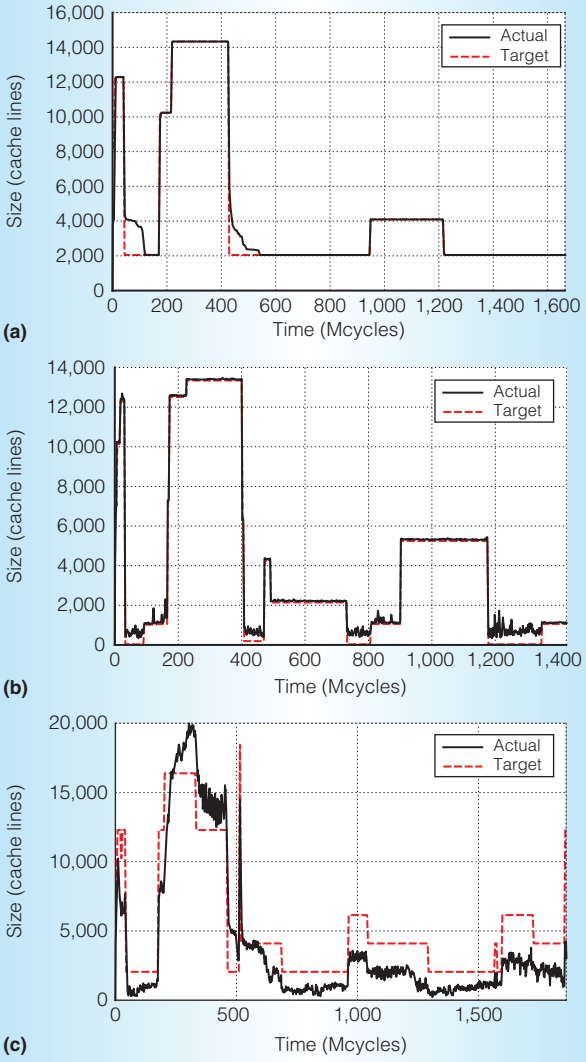


Figure A. Comparison of way partitioning (a), Vantage (b), and promotion-insertion pseudo-partitioning (PIPP) (c) for a specific partition in a four-core mix. Plots show target partition size (as set by utility-based cache partitioning) and actual size for the three schemes.

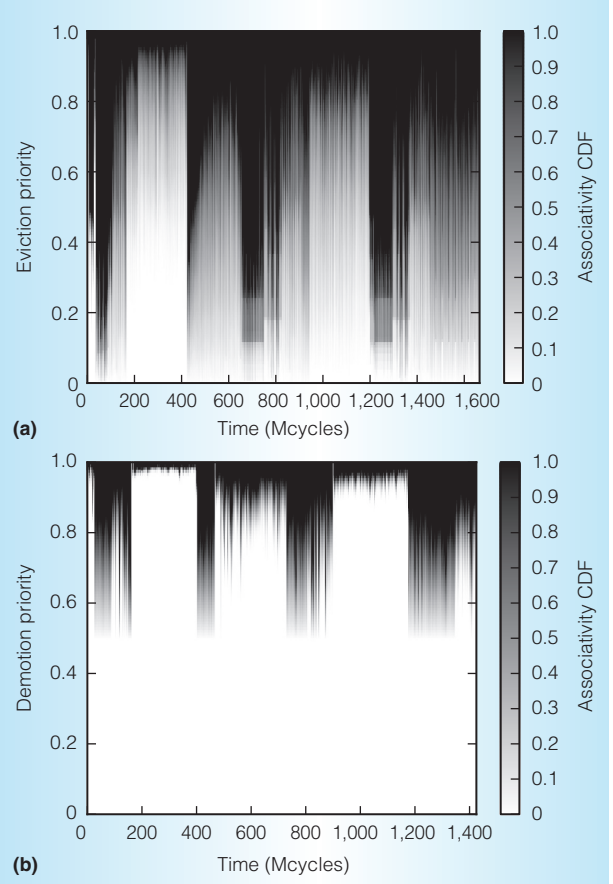


Figure B. Heat maps of the measured associativity cumulative distribution function on the partition from Figure A for way partitioning (a) and Vantage (b). For a given point in time (*x*-axis), the higher in the *y*-axis that the heat map starts becoming darker, the more skewed the demotion or eviction priorities are toward 1.0, and the higher the associativity is.

7. D. Sanchez and C. Kozyrakis, ''Vantage: Scalable and Efficient Fine-Grain Cache Partitioning,'' *Proc. 38th Ann. Int'l Symp. Computer Architecture* (ISCA 11), ACM, 2011, pp. 57-68.
8. A. Seznec, ''A Case for Two-Way Skewed-Associative Caches,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), ACM, 1993, pp. 169-178.
9. D. Sanchez and C. Kozyrakis, ''The ZCache: Decoupling Ways and Associativity,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2010, pp. 187-198.
10. D. Chiou et al., ''Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches,'' *Proc. 37th Design Automation Conf.* (DAC 00), ACM, 2000, pp. 416-419.
11. Y. Xie and G.H. Loh, ''PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture* (ISCA 09), ACM, 2009, pp. 174-183.

Daniel Sanchez is a PhD student in the Department of Electrical Engineering at Stanford University. His research focuses on large-scale chip multiprocessors, specifically on scalable and dynamic fine-grained runtimes and schedulers, hardware support for scheduling, scalable and efficient memory hierarchies, and architectures with quality-of-service guarantees. Sanchez has an MS in electrical engineering from Stanford University. He is a student member of IEEE.

Christos Kozyrakis is an associate professor of electrical engineering and computer science and the Willard R. and Inez Kerr Bell faculty scholar at Stanford University. His research interests include energy-efficient data centers, architecture and runtime environments for large-scale chip multiprocessors, hardware and software techniques for transactional memory, and security systems. Kozyrakis has a PhD in computer science from the University of California, Berkeley. He is a senior member of IEEE and ACM.

Direct comments and questions about this article to Daniel Sanchez, Gates Hall, 353 Serra Mall, Room 354, Stanford, CA 94305; sanchezd@stanford.edu.