KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores

Nosayba El-Sayed[§*] Anurag Mukkara[§] Po-An Tsai[§] Harshad Kasture[§†] Xiaosong Ma[‡] Daniel Sanchez[§]
[§]*MIT Computer Science and Artificial Intelligence Lab* [‡]*Qatar Computing Research Institute, HBKU* [†]*Oracle Labs*
{nosayba, anuragm, poantsai, harshad, sanchez}@csail.mit.edu, xma@hbku.edu.qa

***Abstract*—Cache partitioning is now available in commercial hardware. In theory, software can leverage cache partitioning to use the last-level cache better and improve performance. In practice, however, current systems implement way-partitioning, which offers a limited number of partitions and often hurts performance. These limitations squander the performance potential of smart cache management.**

We present KPart, a hybrid cache partitioning-sharing technique that sidesteps the limitations of way-partitioning and unlocks significant performance on current systems. KPart first groups applications into clusters, then partitions the cache among these clusters. To build clusters, KPart relies on a novel technique to estimate the performance loss an application suffers when sharing a partition. KPart automatically chooses the number of clusters, balancing the isolation benefits of way-partitioning with its potential performance impact. KPart uses detailed profiling information to make these decisions. This information can be gathered either offline, or online at low overhead using a novel profiling mechanism.

We evaluate KPart in a real system and in simulation. KPart improves throughput by 24% on average (up to 79%) on an Intel Broadwell-D system, whereas prior per-application partitioning policies improve throughput by just 1.7% on average and hurt 30% of workloads. Simulation results show that KPart achieves most of the performance of more advanced partitioning techniques that are not yet available in hardware.

***Keywords*-cache partitioning; multicore architectures; performance isolation; application clustering**

I. Introduction

Cache partitioning lets system software divide cache capacity among threads or processes. Partitioning can be used to improve performance and fairness [21, 30, 39, 53], prioritize critical applications [12], and implement isolation among workloads to provide performance guarantees [28, 34]. After years of research showing its potential, recent multicores from Intel [19] and Cavium [9, 59] now implement cache partitioning on the shared last-level cache (LLC).

The problem is that these recent systems implement way-partitioning, a simple cache partitioning technique that has significant limitations. Way-partitioning divides the few (8 to 32) cache ways among partitions. Therefore, the system can support only a limited number of partitions (as many as ways); these partitions are coarsely sized (in multiples of way size); and small partitions have few ways and thus low associativity, adding cache misses. For these reasons, way-partitioning sacrifices significant performance, limiting its usability in current systems.

[*]Part of the QCRI-MIT Joint Research Program.

Prior work has proposed cache partitioning techniques that address the issues of way-partitioning and can support hundreds of finely-sized partitions without hurting performance [42, 58]. However, they are not yet available in hardware. Until better techniques make it to real systems, we should find how to best use the hardware we have.

In this work we focus on using way-partitioning to improve performance. Prior techniques like the widely-used *utility-based cache partitioning* (UCP) [39] give a partition to each application, and allocate more space to applications that use it better. To do this, UCP relies on per-application miss curves, which capture the misses that the application would incur for each possible partition size.

UCP is effective because it complements the replacement policy. Replacement policies make frequent decisions on what specific cache line to evict on each miss, so even sophisticated policies must be simple and act on local information. By contrast, UCP makes infrequent decisions about overall cache allocation, which lets it use more expensive optimization and global information that captures long-term utility. However, UCP is completely ineffective in current systems, because the performance loss incurred by way-partitioning squanders UCP's benefits—in fact, UCP often *hurts performance* even though its only goal is to improve it.

We solve this problem through two contributions. Our first and main contribution is *KPart*, a hybrid cache partitioning-sharing technique that sidesteps the limitations of way-partitioning (Sec. III). KPart first groups applications into clusters, then partitions the cache among these clusters. To do clustering effectively, KPart relies on a novel algorithm to estimate the performance loss of sharing a cache partition. KPart estimates performance for different numbers of clusters and automatically chooses the best-performing cluster count.

KPart, like prior partitioning policies, requires detailed profiling information (e.g., miss curves) to find high-quality configurations. Unfortunately, commodity systems lack the performance monitoring hardware needed to gather this information online (e.g., cache utility monitors [39]). Without this support, prior real-system partitioning techniques either rely on extensive offline profiling, which precludes online adaptation and requires advance application knowledge, or resort to lower-performance partitioning policies that work with limited information [12, 34, 45].

Our second contribution, DynaWay, tackles this problem by enabling low-overhead online profiling (Sec. IV). DynaWay exploits way-partitioning support to perform profiling.

DynaWay periodically samples different partition sizes across co-running applications and uses basic performance counters to infer the per-application cache demand during workload execution. This information can be used to guide several cache and memory optimizations; in this work we use it to guide KPart's clustering and partitioning decisions.

We implement KPart and DynaWay and evaluate them using a wide range of application mixes running on an Intel Broadwell-D system with way-partitioning (Sec. V). We first evaluate KPart with offline application profiles to study the quality of its clustering and partitioning decisions. Whereas a conventional policy that assigns cache partitions to individual applications improves throughput by just 1.7% on average and hurts performance on 30% of the mixes, KPart improves throughput by 24% on average (up to 79%) and does not hurt performance in any mix. We demonstrate that KPart's cache-aware clustering is key to achieving good performance—simpler clustering techniques yield minimal benefits.

We then evaluate KPart+DynaWay, which relies on online profiling to make dynamic clustering and partitioning decisions, and adapts to applications' time-varying cache demands. KPart+DynaWay matches or outperforms KPart with offline profiling information and does not require a-priori knowledge of application characteristics.

We also use simulation to compare KPart with Vantage [42], a fine-grained partitioning technique that does not yet exist in real hardware. KPart performs almost as well as Vantage+UCP, closing most of the gap with high-performance partitioning when the goal is to improve performance.

Finally, we show that KPart+DynaWay can be extended to handle mixes of latency-critical and batch applications, improving throughput for batch applications without degrading tail latency for a latency-critical application.

In summary, KPart is a simple technique that extracts significant performance from basic way-partitioning hardware. However, KPart is not a full replacement for more advanced partitioning techniques. First, fine-grained partitioning always outperforms partitioning-sharing, as shown by Brock et al. [8]. Second, while KPart improves performance, it does not implement strict isolation, making it inadequate when many applications want strict performance guarantees. KPart is publicly available at http://kpart.csail.mit.edu.

II. Background and Related Work

Cache partitioning enables dividing cache capacity among cores or threads. This section provides background on prior software and hardware partitioning *techniques* to enforce partition sizes, *policies* to determine partition sizes, and current *implementations* of cache partitioning in real systems.

A. Cache partitioning techniques

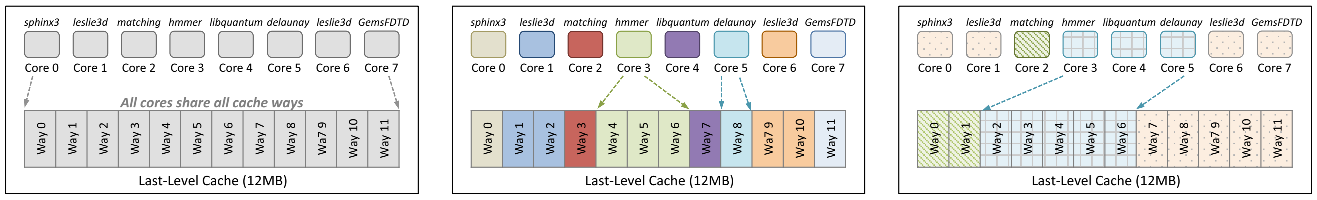
Software techniques rely on *page coloring* to partition the cache along sets, restricting the physical pages used by each partition so they map to specific cache sets [32, 47, 54, 65, 66]. Page coloring needs no hardware support and does not sacrifice associativity, but despite much prior work, it is rarely used. This is due to four main drawbacks. First, page coloring requires heavy modifications to the OS's virtual memory subsystem and precludes the use of other beneficial features, such as superpages. Second, partitions are coarsely sized (in multiples of page size$\times$cache ways), so the number of partitions is limited. This limitation is worse when caches are indexed using hashing, common in LLCs of modern systems [36]. Third, the application's memory footprint imposes a lower bound on partition size. For example, an application that uses half of physical memory must get at least half of the cache. This is unfortunate, because high-footprint applications often use the cache poorly and should use small partitions. Fourth, repartitioning incurs high overheads due to the costly process of recoloring (copying) memory pages.

Why not use page coloring? The above limitations are qualitatively severe, yet one might ask whether they have a major quantitative effect. We perform a simple experiment to answer this question. Consider just one of the virtual memory features that page coloring is incompatible with, superpages. Current OSs use superpages whenever possible to reduce TLB pressure and thus translation overheads. In our baseline system, disabling superpages (by turning off Linux's Transparent Huge Pages [13]) degrades performance by 10% on average on our applications, with individual applications slowing down by up to 33%. Since cache partitioning improves performance by 20-25%, even this single drawback *erases about half of the potential benefits*. We conclude that coloring is at a severe quantitative disadvantage.

Hardware techniques modify the cache to support partitioning. Way-partitioning [1, 2, 11, 40], the most common technique, restricts insertions from each partition to its assigned subset of ways. Though simple, way-partitioning has significant limitations: it supports a small number of coarsely-sized partitions (in multiples of way size), and partition associativity is proportional to its way count, so partitioning can easily degrade performance.

Prior work has proposed alternative hardware techniques that avoid the problems of way-partitioning. Some techniques make the indexing function configurable to partition the cache by sets instead of ways [4, 40, 57] while avoiding the problems of page coloring. Other techniques modify the insertion and replacement policies [35, 42, 58, 62, 64]. Among these, Vantage [42] and Futility Scaling [58] provide strong probabilistic bounds on partition size and isolation among partitions. These techniques support hundreds of partitions and do not degrade performance. However, they are not available in existing hardware, which implements basic way-partitioning.

Hybrid techniques: Finally, Wang et al. [59] recently proposed SWAP, a technique that combines set- and way-partitioning to achieve finer-granularity partitions than are possible with either technique. SWAP and KPart share the



(a) With no partitioning (our baseline), all applications share all ways.

(b) With per-application partitioning, each application is mapped to a different partition.

(c) With KPart, multiple applications can be mapped to a single partition.

Figure 1: Real-system example demonstrating the limitations of conventional UCP in a way-partitioned cache on an 8-core Intel Broadwell processor.

same goal: making way-partitioning useful. However, they take different approaches. SWAP leverages page coloring to achieve finer-grain partitions. SWAP thus inherits the limitations of page coloring: SWAP reduces but does not remove the recoloring problem, requires heavy OS modifications, and does not work with superpages (since a 2 MB x86 superpage covers all cache sets). By contrast, KPart leverages clustering to make good use of coarse-grain partitions. KPart avoids the performance and complexity issues of page coloring, but does not provide isolation among applications in the same cluster.

B. *Cache partitioning policies*

A partitioning policy allocates cache capacity to each partition to satisfy system-level objectives, such as maximizing system throughput [8, 39], improving fairness [32, 37, 60], or guaranteeing quality of service [34, 67]. A key aspect of partitioning policies is how they map applications to partitions. Most policies map each core or application to a different partition [37, 39], and some use multiple partitions per core or application [5, 38]. By contrast, KPart clusters applications so that these policies work well even when hardware supports a limited number of partitions.

The most popular partitioning policy for optimizing *system throughput*, which is the focus of this work, is Utility-based Cache Partitioning (UCP) [39]. UCP decides partition sizes using per-core miss curves. These curves capture the misses that the core would incur for each possible partition size. UCP proposes simple hardware monitors to estimate miss curves, though other proposals profile them offline or infrequently in software [55]. UCP proposes the *Lookahead algorithm*, a quadratic-time algorithm that finds the partition sizes that seek to maximize the expected total number of cache hits (i.e., the cache utility). Prior work has extended online miss curve profiling and cache partitioning to non-uniform cache architectures (NUCAs) [4, 26, 31, 56] and to perform thread mapping [6, 25]. In place of the Lookahead algorithm, prior work has also used dynamic programming to find the optimal partition sizes [8, 37, 50].

KPart adopts UCP's Lookahead algorithm, but allocates cache ways across *groups of applications*. In addition to performing cache-aware application clustering, KPart improves UCP's decisions by *(i)* using instructions per cycle (IPC) speedup, a more direct performance metric, as the utility objective, and *(ii)* accounting for bandwidth contention.

C. *Cache partitioning in real systems*

Recent processors from Intel [19] and Cavium [9, 59] feature hardware support for cache partitioning, but are limited in two aspects. First, they implement way-partitioning, hurting performance when many partitions are used. Second, they do not have hardware miss curve monitors, which limits the information available to dynamic partitioning policies.

These handicaps have so far limited the range of policies these systems can support. While Cook et al. [12], Heracles [34], and Dirigent [67] exploit this hardware support, they seek to prioritize or isolate a single critical application. Therefore, they use two partitions only (one for the critical application and another for all other processes), and their partitioning algorithm is driven by the performance needs of the critical application rather than by overall efficiency. By contrast, techniques that optimize for overall efficiency, like UCP, assign cache partitions on a per-application basis and fare poorly on real hardware with coarse-grained way-partitioning (see motivating example in the next section).

Finally, concurrently with our work, Selfa et al. [45] have recently proposed to leverage way-partitioning to improve system fairness. Like KPart, Selfa et al. group applications into clusters, then partition the cache among clusters. Because current systems lack hardware miss curve monitors, Selfa et al. rely on k-means to cluster applications based on a single hardware performance counter. This simplistic clustering can degrade system throughput significantly unless clusters use overlapping cache partitions, and even then, it does not improve system throughput [45]. By contrast, KPart groups applications using richer profiling information and significantly improves system throughput. Our evaluation shows that clustering based on detailed profiling information is essential to achieving high-performance configurations.

III. KPart Design

KPart is a hybrid cache partitioning-sharing technique. It avoids the granularity and associativity pitfalls of way-partitioning by grouping applications into a few clusters, then partitioning the cache among these clusters. While reducing the number of partitions by grouping applications is a straightforward concept, finding the right set of clusters is not.

Motivating example: Fig. 1 showcases the need for KPart using a concrete example. We run a mix of applications on an Intel Broadwell-D processor where eight cores share a

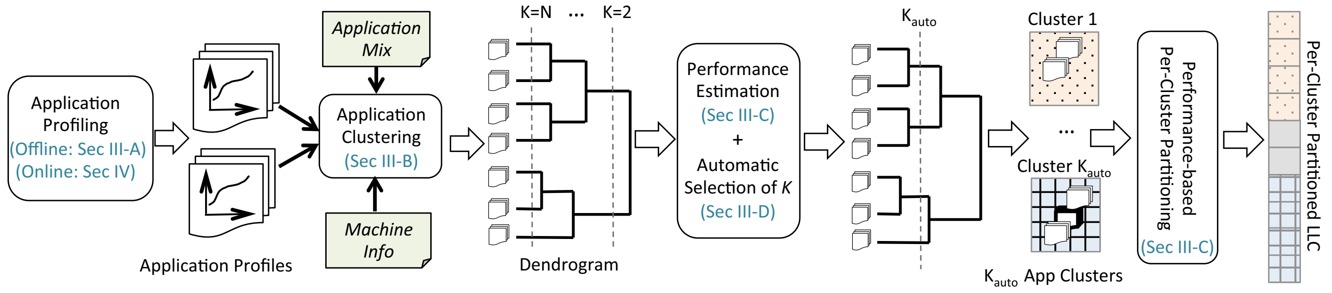


Figure 2: KPart overview. KPart uses hierarchical clustering to group applications, estimates the performance of each number of clusters under partitioning, chooses the number of clusters that yields the best estimated performance, then assigns cache partitions to clusters.

12 MB, 12-way LLC. Fig. 1a shows the eight-application mix we use running in the baseline, unpartitioned configuration, where all applications share the whole LLC. Fig. 1b shows how a conventional per-application partitioning policy, UCP, divides cache capacity. In this machine, each partition must receive at least one way. Therefore, all applications but `hmmer` receive one or two ways, and suffer from poor associativity. As a result, UCP ends up *degrading throughput* (measured as weighted speedup) over the unpartitioned baseline by 3.8%.

Fig. 1c shows how KPart sidesteps way-partitioning's limitations. Using application cache profiles, KPart groups this set of applications into three clusters of one, three, and four applications each. It then partitions the cache among clusters. Applications within a cluster share the cluster's partition. That way, each partition is larger than with UCP and associativity is thus sufficient. KPart improves performance by 17% over the baseline on this mix.

Overview: Fig. 2 provides an overview of KPart's design. First, KPart profiles applications to collect information about their cache access patterns. Profiling can be done either offline, once per application (Sec. III-A), or online, adapting dynamically to application behavior (Sec. IV). Second, given an application mix, KPart uses a hierarchical clustering algorithm to group applications with compatible cache-access behaviors into cache-sharing clusters (Sec. III-B). Third, KPart combines performance estimation heuristics and UCP's Lookahead algorithm to generate the LLC partitioning plan for each possible number of clusters, which we denote as K (Sec. III-C). Finally, KPart automatically selects the number of clusters K estimated to produce the best throughput for the application mix (K_{auto}), and partitions the LLC according to the partitioning plan for this cluster count (Sec. III-D).

A. Profiling information

KPart uses information about an application's memory-access patterns to guide its decisions, particularly several curves that capture how sensitive the application's performance is to cache capacity, collected either offline or online. In offline profiling, the performance curves are constructed by executing each application with various cache partition sizes (in terms of cache ways) on the target machine. KPart measures the following metrics in each run: cache misses (in misses per kiloinstruction (MPKI)), average instructions per cycle (IPC), and average memory bandwidth (in GB/s).

The cache miss measurements provide the application's *miss curve*, which describes how misses change as a function of available cache capacity. KPart's clustering algorithm (Sec. III-B) uses such per-application miss curves to compute a distance metric between applications in a given workload mix and group them into clusters. Similarly KPart builds per-application *IPC curves* and *memory bandwidth curves* from IPC and bandwidth readings at different cache sizes.

While KPart's clustering algorithm only uses miss curves, the IPC and bandwidth curves are important to find the cache partitioning plan for a given set of clusters and to choose the right number of clusters. In particular, IPC curves give a more direct measure of application performance than miss curves. For example, we find that, due to prefetching and memory-level parallelism, an application can have large reductions in misses but little to no change in IPC.

Online profiling produces the same curves described above, but during workload execution, using a novel and efficient sampling technique (detailed in Sec. IV).

B. Clustering applications

KPart groups applications based on their cache-sharing compatibility levels. To achieve this, KPart uses a distance metric between application miss curves that captures *how many additional cache misses are expected* when two applications share a partition versus when they are isolated in separate partitions.

Distance metric: Our pairwise distance metric is the same as that proposed for Whirlpool [38]. To quantify the expected additional cache misses when two applications share a cache partition, KPart first estimates the "combined" and the "partitioned" miss curves for those applications:

- The **combined miss curve** gives the expected misses when two applications share a cache partition for the range of possible partition sizes. For each partition capacity, Whirlpool's model [38] estimates the cache space each application would occupy on average when sharing the partition. The expected misses is the sum of the misses of individual applications at their estimated cache occupancies (computed from the applications' miss curves).[1]

- The **partitioned miss curve** gives the estimated total number of cache misses when a specific amount of cache capacity is partitioned among two applications (using UCP's Lookahead algorithm).

KPart then computes the distance between two applications as the area between their combined and partitioned miss curves. At each cache size, the partitioned miss curve has lower misses than the combined miss curve, indicating the benefit of cache partitioning. The area between the two curves reflects how many additional misses are expected when these applications share the cache, aggregated over different cache sizes. A larger area indicates an overall higher relative performance gain by isolating the two applications over letting them freely share cache capacity.

Note that Whirlpool [38] uses the distance metric for a different objective: to cluster a single application's data structures into pools and partition the cache dynamically among them. In contrast, KPart aims to group applications into clusters so that they can share a single cache partition.

Hierarchical clustering: For a given mix of N applications, the objective of KPart's clustering algorithm is to learn how the applications should be mapped to a given number of clusters K ($K \in [1,N]$). With K=1 (which we call NoPart) the LLC is left unpartitioned, while with K=N (which we call NoClust) each application receives its own partition.

KPart uses a simple *hierarchical clustering* [24] algorithm that produces the application groupings for all possible cluster counts in one pass. The algorithm works iteratively. It starts with each application in a separate cluster. In each iteration, it computes the pairwise distances between all clusters using the above distance metric and merges the two closest clusters.

To compute the miss curve of the new cluster, KPart combines the miss curves of the individual clusters with the same miss curve combining algorithm used to compute the pairwise distances. This new per-cluster combined miss curve is then used in the next iteration to compute the pairwise distances between the new cluster and remaining clusters. Note that the miss curves capture the estimated total cache misses, so clusters of different sizes (containing different numbers of application instances) are properly weighted in the distance calculation and comparison. This process repeats until all applications are merged into a single cluster.

Clustering example: Fig. 3 illustrates KPart's hierarchical clustering algorithm on an example 6-application mix. Each iteration merges two clusters. After 5 iterations, the algorithm produces clustering results for all K values, from 6 down to 1. Fig. 3 shows the application clusters generated by the algorithm for each K. For example, with K=2, `hmmer` and `sphinx` share a cluster (shown in green), while the remaining applications share the second cluster (in red). With

[1] We use Whirlpool's combined miss curve model because it is simple and accurate. Prior work has proposed other ways to estimate combined miss curves [10, 52, 63]. KPart could be used with these techniques instead.

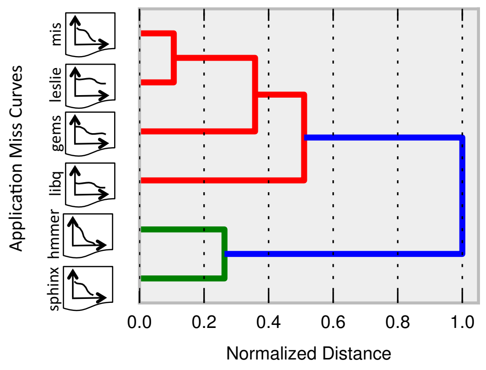


Figure 3: KPart's hierarchical clustering on an example 6-application mix.

K=3, KPart places `libq`, a streaming application, in its own cluster to separate it from its more cache-sensitive peers.

C. Partitioning for a fixed number of clusters

KPart's hierarchical clustering algorithm produces application groups for all possible numbers of clusters K. We now explain how KPart partitions cache capacity among clusters for a given K (Sec. III-D discusses how to select the best K). Performing cache partitioning among clusters is very similar to partitioning among applications. In fact, we could simply apply standard UCP, feeding the *per-cluster miss curves* to the Lookahead algorithm to find the sizes that minimize total cache misses across the system. The miss curve of each cluster is readily available—it is the combined miss curve used by the clustering algorithm.

Optimizing speedups: Partitioning using miss curves works well, as we will see in Sec. V, but we can do better. Because KPart seeks to maximize weighted speedup, it is better to optimize on *speedup curves*. We define an application's speedup curve as:

$$Speedup(s) = \frac{IPC(s)}{IPC(C/N)} \quad (1)$$

where $IPC(s)$ is the IPC curve, i.e., the application's instructions per cycle when it is given cache size s (Sec. III-A), C is the total cache capacity, and N is the number of applications (hence C/N is the amount of cache that would correspond to the application if the cache was divided evenly). The speedup curve reflects how application performance changes when it is given more or less cache space than its fair share.

Since KPart partitions among clusters, it needs the speedup curve of each cluster. The per-cluster speedup curve for a given cache size s is the *sum* of the speedups of the cluster's applications $1,...,j$, at the cache sizes they consume $s_1,...,s_j$, i.e., $Speedup(s) = \sum_{i=1}^{j} Speedup_i(s_i)$, where $s = \sum_{i=1}^{j} s_i$. Luckily, $s_1,...,s_j$ are readily available, because the algorithm that we use to compute the combined miss curve (Sec. III-B) already tracks the fraction of space that each application in the cluster would consume at each size s.

Once KPart has computed the per-cluster speedup curves, it simply feeds them into the Lookahead algorithm to find the partition sizes that maximize aggregate speedup.

Prior simulation-based studies proposed to perform partitioning on IPC curves directly, but found minor throughput gains [33]. We do observe about 5% higher weighted speedup on average when using speedup curves instead of miss curves, with larger gains in some mixes. The reason is that some applications have large differences in misses over partition sizes, but their IPCs barely change. This happens when prefetchers are effective or the core can issue many memory references in parallel, reducing their latency impact.

Accounting for memory bandwidth contention: In addition to cache capacity, applications also compete for main memory bandwidth. We find that memory bandwidth is often saturated, which affects applications' performance (i.e., IPC) differently. Hence, it is important to account for bandwidth contention in KPart's decisions. KPart uses a simple technique to achieve this. This technique leverages each application's memory bandwidth curve, which reflects how much data is transferred to and from memory at each possible cache size (Sec. III-A). Once the partitioning plan is done, KPart uses the bandwidth curves to estimate the total memory bandwidth consumption. If this value exceeds the system's memory bandwidth, KPart derates IPCs to stay within the system's memory bandwidth. Specifically, KPart scales each application's IPC curve by a constant derating factor. KPart then performs a second round of partitioning using these scaled IPC curves to find the final partition sizes.

In general, it is hard to infer exactly how slower memory accesses slow down each application. KPart does something very simple: it derates each application's performance in proportion to its memory bandwidth (e.g., an application that consumes 4 GB/s will be slowed down twice as much as an application that consumes 2 GB/s). This is simplistic but captures the first-order effect of bandwidth contention. It relies on the assumption that each application has identical memory-level parallelism. Although prefetchers make this assumption grossly inaccurate in theory, real prefetchers throttle themselves when memory bandwidth is limited, so we find our approach is reasonably accurate in practice. KPart could be combined with more sophisticated slowdown estimation techniques [51], which we leave to future work.

By default, KPart uses speedup curves and accounts for memory bandwidth contention to partition cache capacity. In Sec. V, we will see how these techniques improve performance over partitioning solely based on miss curves.

D. Selecting the best number of clusters

KPart predicts the number of clusters K that would produce the highest improvement for a given workload using a simple but effective approach. A nice side effect of using speedup curves to size partitions, as presented above, is that the optimization algorithm produces an estimated aggregate speedup, which is directly correlated to weighted speedup. Therefore, KPart produces partitioning plans across the whole range of numbers of clusters K, and then chooses the K that is estimated to perform best. We refer to this automatically selected K as K_{auto}.

We also experimented with other machine-learning clustering indexes that estimate a ratio between inter-cluster dissimilarity and intra-cluster similarity, but found our simple technique to be more effective. Although our technique requires repeating the partitioning process across the full range of K values, we find it has small overheads at the scales we study.

E. Overhead

The key overhead in KPart is running the quadratic-time Lookahead algorithm [39] for each possible K. For a specific number of clusters K, the runtime complexity to partition a N-way LLC with the Lookahead algorithm is $O(K \cdot N^2)$. Therefore, for a system with a N-way LLC running M applications, the runtime complexity of K_{auto} is $O(\sum_{k=1}^{M} k \cdot N^2) = O(M^2 \cdot N^2)$. In our experiments, with 8 applications and 12 ways, the whole KPart algorithm finishes in less than 1 ms. However, larger systems with hundreds of cores and applications may need more efficient approaches, such as sampling a few K values or using Peekahead [4], an $O(K \cdot N)$ implementation of Lookahead.

IV. DynaWay: Dynamic profiling using way-partitioning

KPart uses miss, IPC, and bandwidth curves to guide its decisions. Ideally, the system should include hardware monitors that sample this information for free. Unfortunately, existing systems lack such monitoring infrastructure.

Specifically, prior work has proposed utility monitors (UMONs) [39] to profile miss curves in hardware. UMONs are small set-associative tag arrays (e.g., 1–4 KB per core) that sample a fraction of the accesses. UMONs leverage address sampling to simulate large caches efficiently and the stack property of LRU to profile multiple partition sizes at once. Although UMONs alone do not provide IPC curves, these could be accurately estimated from miss curves given additional performance counters designed to predict the impact of varying miss rates, such as CPI stacks [17]. Similarly, bandwidth curves can be easily derived from miss curves given the fraction of dirty data in the LLC, as their key difference is that they include dirty writebacks.

Lacking this support, prior work has proposed many software techniques to estimate miss curves in software [7, 44, 46, 55]. Like UMONs, these methods rely on *address sampling*, which is expensive to perform in software, requiring either binary instrumentation [44] or interrupting the thread on every memory access [7, 46]. These tools then use efficient analytical models, such as StatCache [7] or StatStack [15], to estimate miss curves from address traces. Even with careful time- or phase-based sampling [44, 46], these techniques incur substantial slowdowns, of 21% to over 2× [46], making them too slow for our purposes

(RapidMRC [55] has lower overheads because it uses IBM POWER5's SDAR performance counters, which are not available in commodity systems).

To overcome the lack of hardware monitors while avoiding the overheads of software address sampling, we design DynaWay, a simple technique that exploits way-partitioning to perform online profiling.

DynaWay works by periodically taking control of way-partitioning hardware for a brief profiling phase. In this phase, it profiles one application at a time. It splits cache capacity into two partitions: one for the target application and the other shared by all remaining applications. Then, it sweeps the application's partition size over multiple measurement intervals. At the end of each interval, DynaWay measures the LLC misses, IPC, and memory accesses for the profiled application. This yields the miss, IPC, and bandwidth curves for all applications.

During each profiling phase, DynaWay's sweeps will degrade the performance of cache-sensitive applications. However, as we will see in Sec. V, this has a minor impact on overall performance, for two reasons. First, profiling sparingly (e.g., one profiling phase per 40 B cycles) is sufficient. Second, we introduce two simple optimizations that reduce the number of measurement intervals and their length:

Optimization 1—Reducing the number of measurement intervals: On a system with A applications and a W-way cache, naive DynaWay would require $A \cdot W$ measurement intervals. Instead of sweeping all possible cache allocations per application, we find it is sufficient to sample few allocations and interpolate to estimate the whole curve. On our 12-way LLC, we profile 6 evenly spaced allocations (1, 3, ..., 11 ways), halving the number of measurement intervals.

Optimization 2—Reducing measurement interval length: On each measurement interval, DynaWay should ideally capture the steady-state behavior of the profiled partition size. The problem is that each measurement phase incurs a significant *transient period*: when DynaWay changes partition sizes, threads take time to fill and leverage their new allocations. These transients can be tens of milliseconds long [28], so it is crucial to minimize them.

We observe that an effective strategy to reduce warmup length is to *sample decreasing partition sizes*. Fig. 4 illustrates this process. Each application is first given a single warmup interval where it is allowed to take over the cache. Then, each successive measurement interval decreases its allocation. Because part of its allocation is given to *all other applications* in the system, they fill this space quickly, resulting in a short transient between successive intervals. Profiling increasing partition sizes (e.g., starting at 1 way and giving 2 extra ways each interval) is comparatively much slower, as a single application takes more time to fill in cache space.

This simple strategy effectively reduces measurement interval sizes. With our strategy, 20 Mcycle intervals produce

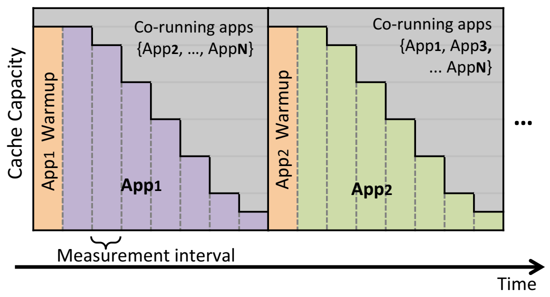

Cache Capacity
App1 Warmup
Co-running apps {App2, ..., AppN}
App1
App2 Warmup
Co-running apps {App1, App3, ... AppN}
App2
...
Measurement interval
Time

Figure 4: DynaWay's profiling phase. DynaWay samples decreasing partition sizes to reduce the impact of transients.

accurate curves. Meanwhile, sampling increasing partition sizes requires 80–100 Mcycle intervals to achieve comparable accuracy. Our strategy thus yields a 4–5× reduction in measurement interval length.

Overhead: DynaWay, and indeed any other software miss curve profiling technique, will cause some performance degradation on each profiling phase. In DynaWay, the length of the profiling phase scales linearly with core count. This adds small overheads on systems with up to tens of cores, as we show in our evaluation (Sec. V-F). However, systems with hundreds of cores would suffer from long profiling phases, and may need less-frequent sampling or hardware support for profiling (e.g., UMONs [39]).

V. Evaluation

A. Experimental methodology

Platform: We evaluate KPart on an 8-core Intel Broadwell D-1540 processor that supports way-partitioning [19]. Table I details the system's main characteristics and configuration.

Applications: We use applications from the SPEC CPU2006 [18] and PBBS [48] benchmark suites. We select the 18 applications that are LLC-intensive, i.e., have an L2 miss rate of at least 5 MPKI. We classify each application

Table I: Specifications of the system used.

Core	8 Xeon D-1540 cores (Broadwell), 2.0 GHz
L1 caches	32 KB, per-core, split D/I, 8-way set-associative
L2 cache	256 KB, private per-core, 8-way set-associative
L3 cache	12 MB, shared, 12-way set-associative, inclusive, DRRIP high-performance replacement policy [23, 61]; Way-partitioning with Intel CAT [19], supports 12 partitions
Memory	32 GB (2×16 GB DIMMs), DDR4 2133 MT/s
OS	Ubuntu 14.04, Linux kernel version 4.2.3

Table II: Applications used in our evaluation (CS = Cache-Sensitive, ST = Streaming).

Suite	Class	Applications
SPEC CPU2006 [18]	CS	`mcf`, `omnetpp`, `sphinx3`, `xalancbmk`
	ST	`hmmer`, `milc`, `leslie3d`, `GemsFDTD`, `lbm`, `libquantum`
PBBS [48]	CS	`bfs`, `delaunay`, `matching`, `refine`, `sa`, `st`
	ST	`mis`, `remDups`

as either cache-sensitive (CS) or streaming (ST), similar to prior work [22]. Cache-sensitive applications are those that observe an IPC improvement of at least 10% over cache allocations, while streaming or cache-insensitive applications do not. Table II lists these applications and their classification.

Workload mixes: For most of our evaluation, we execute 30 different multiprogrammed workloads (i.e., mixes). Each workload consists of 8 applications selected randomly (with replacement) from our pool of LLC-intensive applications (Table II). Each application is pinned to a different core, thereby utilizing all cores in the system. Most of our experiments also enforce a balanced 1:1 ratio between cache-sensitive and streaming applications on each mix. Sec. V-E explores different workload sizes and ratios of cache-sensitive and streaming applications.

Like prior work, we use a fixed-work methodology [20] to run each workload. All applications in the mix start together and are kept running until each application completes at least 10 billion instructions. We consider only the first 10 billion instructions of each application when reporting performance.

Metrics and techniques: We measure system throughput using weighted speedup [16, 49], calculated as follows for a mix of N applications:

$$WeightedSpeedup = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_i}{IPC_{i,NoPart}} \quad (2)$$

NoPart refers to the *baseline* configuration, where the LLC is left unpartitioned and is shared among all applications.

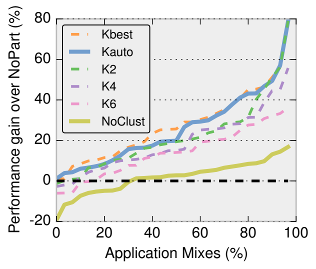
For a given mix of N applications, we evaluate KPart while varying K (the number of clusters) from 2 to N, in separate runs. We refer to K=N as NoClust, since this scheme represents a conventional policy where each application receives its own partition, with no clustering.

We exhaustively evaluate all possible K values and not only our estimated K_{auto} (from Sec. III-C) to better understand how KPart performs under different K choices, as well as to verify K_{auto}'s quality. In particular, we examine the K that produced the best actual throughput, K_{best}, and compare its performance to that of our predicted K_{auto}.

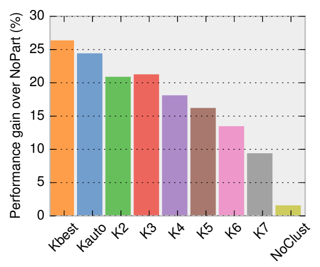
Profiling information: Secs. V-B through V-E focus on analyzing KPart alone, and use per-application profiling information gathered offline. Sec. V-F evaluates KPart+DynaWay, which uses online profiling.

B. KPart performance

Fig. 5 summarizes the performance of KPart and other partitioning schemes when running 30 different 8-application mixes using offline application profiles. Fig. 5a shows the distribution of weighted speedups (S-Curve or inverse CDF) of KPart with fixed and automatically-chosen numbers of clusters. Each line shows the distribution of weighted speedups over NoPart for a single scheme. Each line is sorted independently. (For clarity, we omit the lines with K=3, 5, and 7 from the S-curve.) Fig. 5b shows the average performance gains across all mixes for all schemes.



(a) Distribution of performance gains (S-Curve). (Higher is better.)

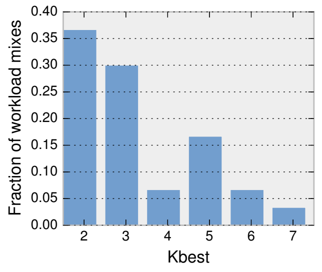


(b) Average performance gains. (Higher is better.)

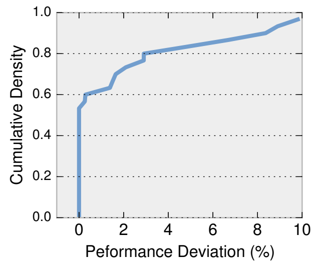
Figure 5: Performance of KPart over NoPart for various configurations. K_{best} is the best-performing number of clusters (K). K_{auto} is KPart's automatically chosen K. $K = 2 ... 7$ are KPart with a fixed number of clusters. NoClust, i.e., $K = 8$, is KPart's partitioning without application clustering.

These results show that KPart improves system throughput by up to **78.9%** over an unpartitioned system and by **24.2%** on average under KPart's estimated K_{auto}. By contrast, we find that using a conventional, per-application partitioning policy like NoClust with coarse-grained way-partitioning improves average performance by 1.7% only, and hurts performance for 30% of the mixes.

We now take a closer look at K_{best}, the K value that actually produces the highest throughput for each mix. Fig. 6a shows the distribution of K_{best}. Overall, smaller cluster counts work better: grouping applications into 2 or 3 cache-sharing clusters yielded the highest throughput in 66% of our mixes. However, a significant fraction of mixes perform best with a large number of clusters. For example, 15% of mixes have $K_{best} = 5$. Moreover, grouping applications into as many as 7 clusters still outperforms NoClust significantly.



(a) Distribution of K_{best} values.



(b) Distribution of performance deviation of K_{auto} w.r.t. K_{best}.

Figure 6: Choosing the K in KPart: (a) Lower K values (2, 3) are generally better. (b) KPart correctly estimates K_{best} around 60% of the time and deviates from K_{best}'s throughput by only 1.7% on average.

How good is KPart's K_{auto}? Fig. 6b studies the quality of KPart's estimated K_{auto} across the mixes by quantifying how much KPart's throughput improvement deviated from that of the best possible improvement for the same mix, given by the oracle K_{best}. A zero deviation implies that KPart was able to select the exact K_{best}, which is the case in around 60% of the mixes. Overall, KPart's delivers high-quality performance estimations and automatic selection of K: the performance of K_{auto} deviates on average from the performance of K_{best} across the mixes by only 1.7% (mispredicting mostly among K values that perform very similarly).

C. Analysis of KPart's performance gains

To deepen our understanding of where KPart's performance gains come from, we isolate and study the effect of two key components in the design of KPart: application clustering and performance-based partitioning.

Effect of clustering: KPart's hybrid sharing-partitioning approach relies on miss curves to cluster applications. We investigate how sensitive KPart is to the quality of the produced clustering configuration, by designing another scheme, RandClust, that maps applications to clusters *randomly*. RandClust is also a hybrid sharing-partitioning scheme, but unlike KPart, RandClust is oblivious to the cache behavior of the applications when it clusters them.

After random clustering, RandClust applies the same subsequent steps as KPart does and produces a per-cluster partitioning plan. We evaluate RandClust with the same 30 mixes. Fig. 7 compares the average performance of RandClust with that of KPart under different cluster counts.

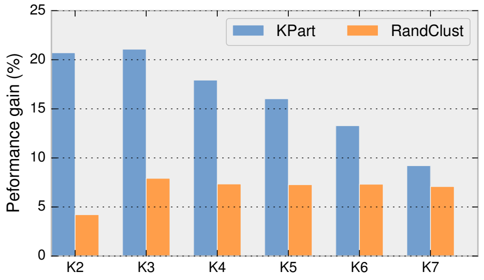


Figure 7: Performance gains of KPart and RandClust (which randomly clusters applications) at various K values.

The results show that KPart outperforms RandClust significantly. RandClust achieves improvements of 4–8% across different K values, and the smallest improvement happens at K=2, where it is crucial to choose the right applications. By contrast, KPart consistently improves performance with fewer clusters, achieving gains of over 20% with K=2 and K=3. Since the only difference between KPart and RandClust is how they group applications, we conclude that KPart's cache-aware clustering algorithm is essential to realize the potential of hybrid partitioning-sharing.

Effect of performance-based partitioning: As discussed in Sec. III-C, KPart performs partitioning using per-cluster speedup curves and accounts for memory bandwidth contention. This is different from conventional policies like UCP, which determine partition sizes using miss curves only and do not account for memory contention. To study the effect of these partitioning enhancements, we implemented two derivatives of KPart that perform application clustering exactly like KPart, but determine the per-cluster partition sizes differently. The first scheme partitions using speedup curves like KPart, while the second partitions based on per-cluster miss curves (i.e., like UCP but on a per-cluster basis). Neither scheme takes into account bandwidth contention.

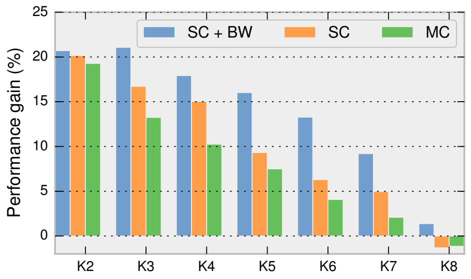


Figure 8: Comparison of partitioning techniques: speedup curve partitioning with bandwidth contention estimation (SC + BW), speedup curve partitioning (SC), and miss curve partitioning (MC).

Fig. 8 compares KPart with the two simplified schemes. It shows that KPart's partitioning approach outperforms traditional miss-curve-based partitioning significantly, and that accounting for memory contention enhances KPart's partitioning quality, particularly for larger cluster counts. In K=8 (i.e. NoClust), performance degrades over NoPart when disabling these partitioning enhancements.

D. A closer look into KPart: Case studies

We now take a closer look at selected, representative mixes to better understand how different applications within a mix interact, as well as the impact of their grouping or isolation. In particular, we analyze the following scenarios:

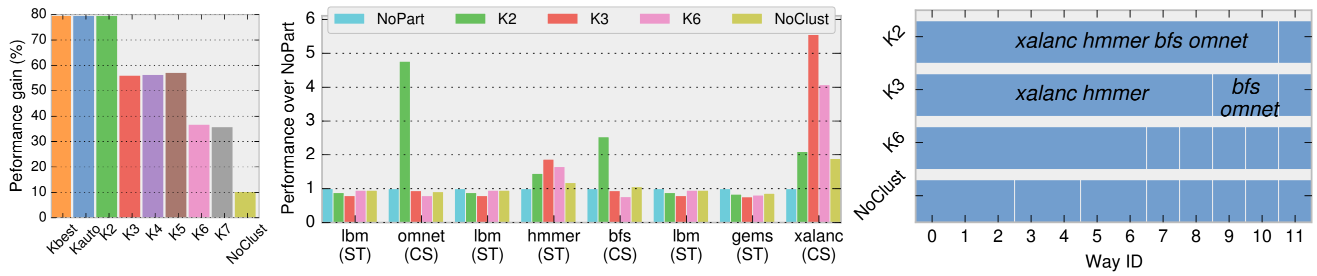
- **Case 1:** The mix that exhibits the highest throughput improvement under KPart and where a low K (number of clusters) works better.
- **Case 2:** A mix where a high K works better.
- **Case 3:** A mix where cache partitioning does not significantly improve performance.

Fig. 9 shows detailed performance results for these cases. The left-most plots show system throughput for each mix under different schemes; the middle plots display per-application speedups (measured as $\text{IPC}_{i,scheme}/\text{IPC}_{i,NoPart}$) for selected schemes; and the right-most plots report the corresponding cache partitioning plans.

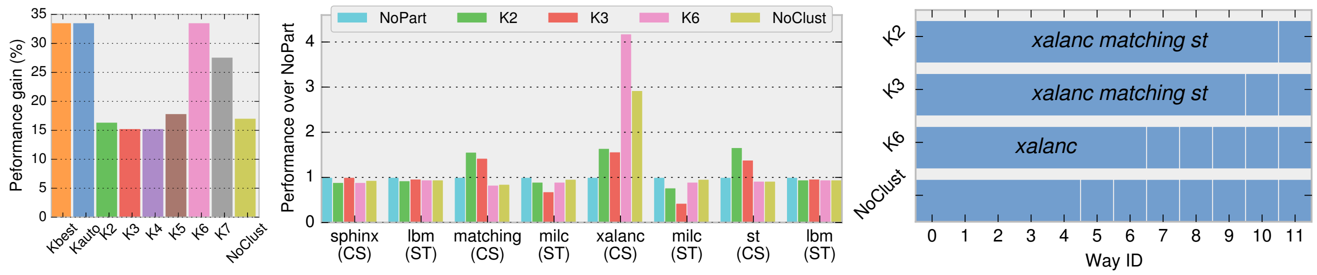
As expected, we find that cache-sensitive (CS) applications typically benefit the most from KPart's hybrid approach. This is the case even when KPart places these CS applications in a shared partition with several other applications.

For example, in **Case 1** (K=2), four different CS applications share a large cache partition of 11 ways: `xalanc`, `bfs`, `omnet`, and `hmmer`. KPart with K=2 ends up isolating these CS applications from the streaming applications in the mix, which are all assigned to share a small, 1-way partition. This configuration benefits the CS applications without hurting the streaming ones. For instance, `omnet` performs 4.8× times better under K=2 compared to the NoPart configuration, as can be seen in Fig. 9a. Overall, KPart improves system throughput for this mix by 78.9% over NoPart.

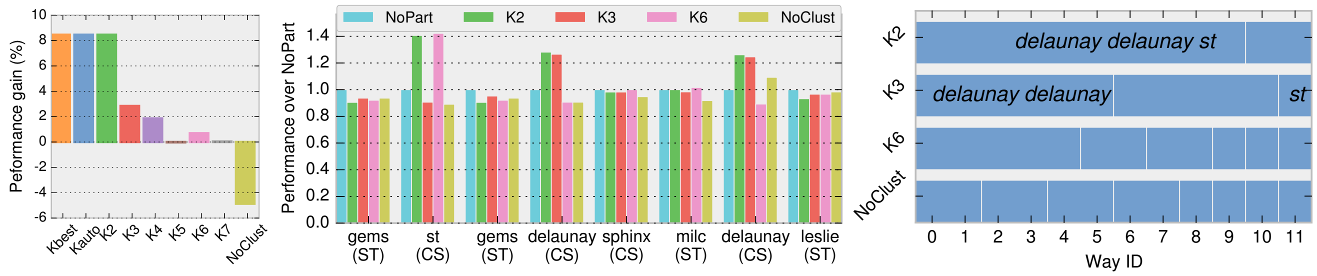
However, simply grouping applications into two partitions, one for CS and another for ST ones, is rarely the best configuration. We examined all K=2 configurations in our



(a) Case 1: Mix with the highest throughput gains (K_{best} = 2).



(b) Case 2: Mix with average throughput gains (K_{best} = 6).



(c) Case 3: Mix with no significant improvement over NoPart.

Figure 9: Case studies: performance gain of each scheme (left), per-application speedups over NoPart (middle), and LLC partitioning plans (right).

mixes and found that in only 4 mixes KPart groups all CS applications in a cluster and all ST applications in another.

Case 2 (Fig. 9b) is a mix where K=6 works best. This can be explained by the significant performance gains achieved by one CS application, `xalanc`, which is given 7 cache ways under K=6. Other CS applications in the mix (e.g., `matching` and `st`) benefit more from sharing a large partition under K=2, but their gains do not improve system throughput as much as `xalanc` does with K=6. This shows that cache-aware clustering is highly workload-dependent: while K=2 or K=3 work well in most cases, there are cases where a larger number of clusters excels.

Finally, **Case 3** (Fig. 9c) is a mix with relatively low improvements under most partitioning schemes. Under the best scheme for this mix, K=2, three CS applications (two copies of `delaunay` and `st`) benefit from sharing a 10-way partition, improving system throughput by 8.5% over NoPart. Using more partitions does not give any additional benefit. In fact, NoClust degrades performance by 4.8%.

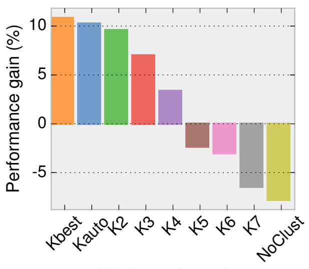
Summary of case studies: The mixes discussed above highlight the importance of combining cache sharing and partitioning to make the best of coarse-grained way-partitioning. Our clustering algorithm is effective in exploiting cache-sharing compatibility between co-running applications, particularly the cache-sensitive applications, by letting them share partitions while staying isolated from streaming applications that would otherwise pollute the cache. These streaming, cache-insensitive applications share a small cache partition, which saves LLC capacity and improves system throughput significantly.

E. Sensitivity analysis

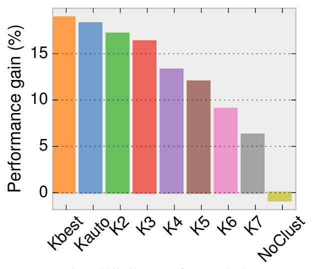
We now analyze KPart's sensitivity to the characteristics of the workload mix, particularly the ratio between cache-sensitive and streaming applications and the mix size.

Effect of workload cache-sensitivity distribution: To assess KPart's effectiveness working with diverse workload mixes, which may not have a balanced distribution between cache-sensitive and streaming members, we generate random mixes while enforcing specific ratios between these two classes of applications.

Fig. 10 reports KPart's performance when running 10 different workloads that are dominated either by (a) cache-sensitive applications, or (b) streaming applications.



(a) CS:ST ratio = 3:1



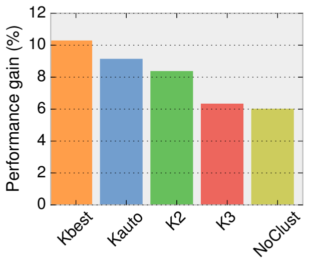
(b) CS:ST ratio = 1:3

Figure 10: Effect of the ratio between cache-sensitive (CS) and streaming (ST) applications in 8-application workloads.

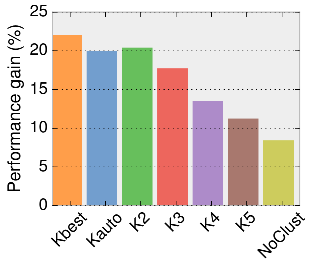
We find that when mixes are dominated by cache-sensitive applications (Fig. 10a), the benefits of partitioning become more muted, and KPart improves performance by 10% on average. The need for KPart however remains crucial here, to avoid the significant throughput *degradation* that would otherwise happen under a policy like NoClust (which hurts throughput by 7.6% on average). Unsurprisingly, with more cache-sensitive applications, NoClust is forced to spread cache ways thin, hurting most applications due to reduced associativity. For similar reasons, high cluster counts with KPart also end up hurting system throughput.

By contrast, when streaming applications dominate the mix (Fig. 10b), KPart improves system throughput by 18.3% on average and up to 64.5%, as it packs these cache-insensitive applications together and gives them small LLC allocations, saving the majority of ways for the cache-hungry minority.

Effect of workload size: So far we have evaluated KPart while fully utilizing all eight cores available. We now study KPart's performance under smaller workload mixes that consist of 4 and 6 applications. Fig. 11 reports these results. As expected, the improvements are higher when the number of co-executing applications grows, due to the increased cache contention, for both space and associativity. With both 4- and 6-application mixes, K_{auto} still performs reasonably close to K_{best}, and smaller K values also fare better. Meanwhile, the no-clustering strategy, NoClust, is less harmful than with 8-application mixes, as the low per-application allocation problem is less severe with fewer applications per mix. These results suggest that the benefit of KPart is likely to be more visible on systems where the core-to-way ratio is higher or the cores tend to be fully occupied (such as datacenter nodes with optimized VM/task scheduling).



(a) 4 applications.



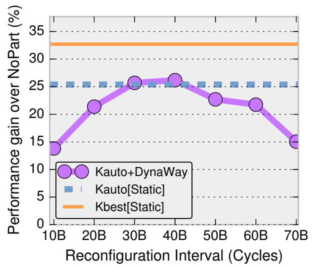
(b) 6 applications.

Figure 11: Effect of workload mix size on the 8-core system.

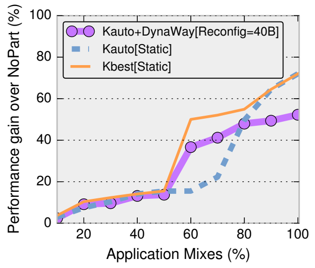
F. KPart with DynaWay's online profiling

The results presented so far were obtained when KPart uses offline profiles. But offline profiling limits KPart's potential in certain scenarios, such as when program input deviates significantly from that used for profiling or when application cache demands vary over time (e.g., due to phase changes).

KPart+DynaWay addresses these issues (Sec. IV). We have implemented KPart+DynaWay in C++ and evaluate it with the same Broadwell system and applications used so far. To better study the value of online profiling and explore a wide range of reconfiguration intervals, we run the workloads in these experiments for a longer period. Specifically, we run mixes until all applications within a mix complete at least 200 B instructions, with the first 60 B instructions used as a warmup period, during which DynaWay performs one or more profiling phases.



(a) Average performance gains under KPart+DynaWay as a function of the reconfiguration interval.



(b) Distribution of performance gains under KPart+DynaWay (S-Curve), when profiling every 40B cycles.

Figure 12: Performance of KPart with online profiling and dynamic cache reconfiguration (KPart+DynaWay) versus offline profiling and static configuration (KPart[Static]).

Fig. 12a shows KPart+DynaWay's average performance gains across 10 workload mixes as a function of the reconfiguration interval used (i.e., the time between profiling phases). We find that, for this system and applications, profiling applications and reconfiguring the cache every 40 B cycles strikes the best balance between profiling overheads and performance gains, improving system throughput by **26.2%** on average over an unpartitioned system. This performance gain not only matches but even surpasses the gain achieved by K_{auto}[Static] (25.4% for these workloads), which does not perform any online profiling. Fig. 12b zooms in on the KPart+DynaWay[40B] performance by showing the distribution of the throughput gains across the mixes. KPart+DynaWay[40B] improves system throughput by up to **52.3%** over an unpartitioned system.

In terms of time spent profiling (which overlaps with regular computation since applications are not stopped during profiling), we find that, under KPart+DynaWay[40B], total profiling time comprises only 1.7% of workload execution time on average across mixes. Moreover, since applications are kept running during profiling, the performance impact is significantly smaller than 1.7%.

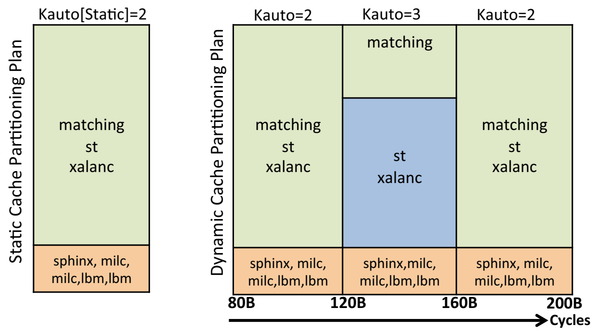
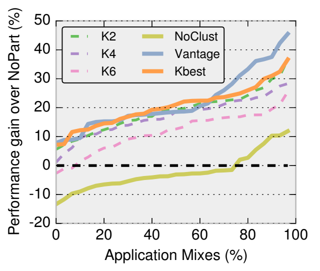
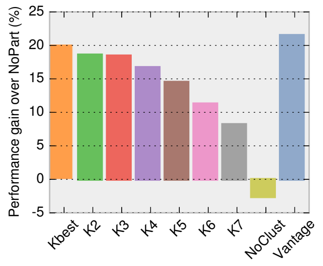

Kauto[Static]=2
Static Cache Partitioning Plan
matching
st
xalanc
sphinx, milc, milc,lbm,lbm
Kauto=2
Kauto=3
Kauto=2
Dynamic Cache Partitioning Plan
80B
120B
160B
200B
Cycles

Figure 13: Case study demonstrating the differences between static KPart and KPart+DynaWay's decisions for a mix of applications that benefits more from dynamic profiling and partitioning.



(a) Distribution of performance gains (S-Curve).



(b) Average performance gains.

Figure 14: Simulation results show that KPart achieves most of the performance benefits of Vantage, a fine-grained partitioning technique that does not exist in real hardware.

Case study: KPart+DynaWay does not use a priori profiling information, and since it profiles applications periodically, it improves performance when applications have time-varying cache needs. We demonstrate this through a case study.

The mix shown in Fig. 13 benefits more significantly from KPart+DynaWay (36.6% improvement over baseline) than static KPart (22.3% improvement over baseline). Static KPart groups these applications into two clusters and dedicates a large 11-way partition to three cache-sensitive applications based on their offline profiles, which capture *average* behavior over the entire execution interval. By contrast, KPart+DynaWay's decisions change over time in response to varying applications needs: KPart+DynaWay's decisions match static KPart's on certain phases, but on phases where `xalanc`'s cache demand increases, KPart+DynaWay adapts its clustering and partitioning decisions accordingly (e.g., 120–160 Bcycles in Fig. 13). As a result, `xalanc` benefits more from KPart+DynaWay, increasing system throughput.

G. KPart vs. high-performance partitioning

Finally, after evaluating KPart on a real system with hardware way-partitioning, we study how KPart fares against fine-grained cache-partitioning techniques *that do not exist yet in real systems*. Our goal is to understand how much of the gap between way-partitioning and more advanced partitioning techniques does KPart bridge. Specifically, we compare against Vantage [42], which supports hundreds of fine-grained partitions without degrading performance.

For these experiments, we perform microarchitectural, execution-driven simulation using zsim [43]. We simulate a 8-core system with a 12MB LLC that matches the configuration of the Broadwell machine described in Table I. We evaluate way-partitioned schemes on a 12-way hashed set-associative cache and Vantage using a ZCache [41] with 4 ways and 52 replacement candidates.

We use the same methodology described in Sec. V-A to generate 30 mixes of SPEC CPU2006 and PBBS apps. However, to avoid long simulation times, we only simulate 2 billion instructions after fast-forwarding for 10 billion.

All the experiments we present use partitioning based on miss curves (not speedup curves), because Vantage uses miss curves. Therefore, we report KPart results for fixed K values and K_{best}, but not K_{auto}, which relies on speedup-curve partitioning. We use the simulator to gather miss curves, and run Lookahead on them. Note that, like NoClust, Vantage gives each application a different cache partition.

Fig. 14 shows the performance gains of these schemes over NoPart. Fig. 14a shows the distribution of performance gains, and Fig. 14b shows the average gain of each scheme. Trends are similar to the real-system results (Fig. 5): K=2 and K=3 perform best among KPart variations, while performance drops for higher K values, and NoClust hurts performance on average. Vantage performs best overall, with a 22% speedup over NoPart on average. K=2 and K=3 both achieve a 18% speedup over NoPart, and K_{best} achieves 20%.

These results show that KPart achieves most of the benefits of advanced, fine-grained partitioning techniques that do not exist yet in real hardware, when partitioning is used to improve system throughput. Effectively, KPart bridges the gap between current and future cache partitioning techniques without requiring any OS modifications or special hardware.

H. KPart with latency-critical applications

Datacenters must often run both latency-critical (LC) and batch applications. LC applications require low *tail* (e.g., 95^{th} percentile) latencies. They thus suffer from low utilization when running alone and are hard to colocate with other applications [3, 14]. To address this issue, recent work [28, 34, 67] proposes to leverage cache partitioning to avoid tail latency violations when colocating LC and batch applications.

Since KPart is a throughput-oriented partitioning scheme, it will hurt the tail latency of LC applications if applied directly [28]. Fortunately, it is straightforward to combine KPart with a QoS-oriented cache partitioning policy, such as Ubik [28] or Heracles [34], to improve the throughput of batch applications while avoiding tail latency violations of the LC application. In this setup, the QoS-oriented cache partitioning technique first determines the capacity required by the LC application, and KPart divides the remaining capacity among batch applications.

Methodology: We use the same 8-core Broadwell-D processor described in Sec. V-A. We dedicate four cores to execute one multithreaded LC application, configured to span four threads. The remaining four cores are used to run a mix of four randomly-selected batch applications. As in prior experiments, all threads are pinned to cores.

For the LC workloads, we experimented with three applications from the TailBench suite [29], which represent request-driven workloads typical in datacenter environments: `moses`, a statistical machine translation system; `masstree`, a scalable in-memory key-value store; and `xapian`, an open-source online search engine. We use TailBench's standard, open-loop harness: requests are sent at a fixed rate (i.e., queries per second, QPS) without waiting for responses to previous requests. Each workload's QPS is set to the value that, when the workload runs alone with full cache capacity, achieves 50% CPU utilization. This is a typical setup [27, 29].

We implement and compare the following schemes:

- NoPart uses an unpartitioned LLC, with the LC and batch applications sharing all 12 ways.
- HC+NoPartBatch uses two partitions, one for the LC application and another for all batch applications. The technique seeks to maintain a *tail latency target* for the LC application by using simple feedback-based hill-climbing (HC), as in Heracles [34]. Initially, the cache is divided equally. If the LC application's current tail latency is below the target, the LC application donates capacity to the batch applications; otherwise, the LC application steals capacity from the batch applications to reduce its tail latency.
- HC+KPartBatch uses the same hill-climbing technique as above to determine the LC partition capacity. However the cache capacity available to batch applications is partitioned using KPart+DynaWay instead.

The HC policies use a different tail latency target for each application. This target is determined by first measuring the LC application's tail latency when given 50% of cache capacity and co-running with the batch applications, which share the remaining 50% of the cache. The target is this tail latency, plus 2% to allow for small performance degradations.

Results: Fig. 15 compares the above schemes. Each plot reports results for a single mix, and each dot shows, for a single scheme, the throughput of the batch applications (x-axis) and the 95th percentile latency of the LC application (y-axis). All results are normalized to HC+NoPartBatch's.

We find that HC+NoPartBatch effectively maintains LC applications at or under their latency targets. However, HC+NoPartBatch has the lowest batch throughput of all schemes, because it leaves relatively little capacity to batch applications, and this capacity is unpartitioned.

By contrast, NoPart improves batch throughput by up to 14% over HC+NoPartBatch. But `xapian` and `masstree` violate their target tail latencies, as they suffer from insufficient cache capacity. Curiously, NoPart improves tail latency for `moses`. This happens because `moses` is bandwidth-sensitive, and giving more cache to batch applications reduces their memory bandwidth demand, leaving more for `moses`.

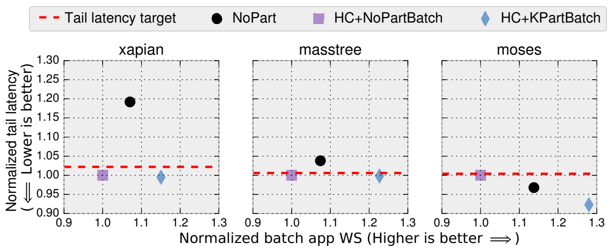


Figure 15: Combining KPart with simple hill-climbing (HC) for latency-critical applications improves throughput for batch applications while protecting/improving the tail latency of the latency-critical application.

Finally, HC+KPartBatch improves batch throughput significantly (by up to 28%) while meeting the tail latency targets for all LC applications. For `moses`, HC+KPartBatch improves tail latency over HC+NoPartBatch even more (by 7%), because KPart+DynaWay reduces the bandwidth demands of batch applications even further.

VI. Conclusion

We have presented KPart, a hybrid cache partitioning-sharing technique that significantly improves throughput in commodity multicores with way-partitioned caches, without requiring OS changes. KPart groups applications into clusters based on their cache-behavior compatibility, then partitions the cache among these clusters. KPart uses detailed profiling information, gathered either offline or online. To achieve low-overhead online profiling, we introduce DynaWay, a simple technique that leverages way-partitioning for profiling. KPart improves throughput by 24% on average (up to 79%) on a Broadwell-D system, whereas a conventional per-application partitioning policy improves throughput by just 1.7%. Moreover, KPart+DynaWay enables low-overhead online profiling, matching the performance of KPart with offline profiling while avoiding its drawbacks. KPart is publicly available at http://kpart.csail.mit.edu.

Acknowledgments

We sincerely thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CAREER-1452994 and by a grant from the Qatar Computing Research Institute, HBKU.

References

[1] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO-32*, 1999.
[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *MICRO-33*, 2000.
[3] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, 2007.
[4] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *PACT-22*, 2013.
[5] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *HPCA-21*, 2015.
[6] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *HPCA-21*, 2015.
[7] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *SIGMETRICS*, 2005.

[8] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang *et al.*, "Optimal cache partition-sharing," in *ICPP-44*, 2015.

[9] Cavium Inc., "ThunderX family of workload optimized processors," http://cavium.com/pdfFiles/ThunderX_PB_p12_Rev1.pdf, archived at https://perma.cc/M2LG-9NP7, 2013.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA-11*, 2005.

[11] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *DAC-37*, 2000.

[12] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA-40*, 2013.

[13] J. Corbet, "Transparent huge pages in 2.6.38," *LWN*, http://lwn.net/Articles/423584, archived at https://perma.cc/4MRR-49RE, 2011.

[14] J. Dean and L. A. Barroso, "The tail at scale," *Comm. ACM*, vol. 56, no. 2, 2013.

[15] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *ISPASS*, 2010.

[16] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, 2008.

[17] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *ASPLOS-XII*, 2006.

[18] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[19] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos *et al.*, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *HPCA-22*, 2016.

[20] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," *MoBS*, 2009.

[21] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource," in *PACT-15*, 2006.

[22] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr *et al.*, "Adaptive insertion policies for managing shared caches," in *PACT-17*, 2008.

[23] A. Jaleel, K. Theobald, S. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction," in *ISCA-37*, 2010.

[24] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, 1967.

[25] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems," in *HPCA-16*, 2010.

[26] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *ICPP-38*, 2009.

[27] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *MICRO-48*, 2015.

[28] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *ASPLOS-XIX*, 2014.

[29] H. Kasture and D. Sanchez, "TailBench: A benchmark suite and evaluation methodology for latency-critical applications," in *IISWC*, 2016.

[30] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT-13*, 2004.

[31] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *HPCA-17*, 2011.

[32] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA-14*, 2008.

[33] X. Lin and R. Balasubramonian, "Refining the utility metric for utility-based cache partitioning," *WDDD-9*, 2011.

[34] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ISCA-42*, 2015.

[35] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *ISCA-39*, 2012.

[36] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Intl. Workshop on Recent Advances in Intrusion Detection*, 2015.

[37] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero, "FlexDCP: A QoS framework for CMP architectures," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, 2009.

[38] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *ASPLOS-XXI*, 2016.

[39] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO-39*, 2006.

[40] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *ISCA-27*, 2000.

[41] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *MICRO-43*, 2010.

[42] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *ISCA-38*, 2011.

[43] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[44] D. Schuff, M. Kulkarni, and V. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *PACT-19*, 2010.

[45] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. Gomez, "Application clustering policies to address system fairness with Intel's Cache Allocation Technology," in *PACT-26*, 2017.

[46] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase guided profiling for fast cache modeling," in *CGO*, 2012.

[47] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *ICS'99*, 1999.

[48] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola *et al.*, "Brief announcement: The problem based benchmark suite," in *SPAA*, 2012.

[49] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," in *ASPLOS-IX*, 2000.

[50] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Transactions on Computers*, vol. 41, no. 9, 1992.

[51] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO-48*, 2015.

[52] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *ICS'01*, 2001.

[53] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *HPCA-8*, 2002.

[54] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *WIOSCA*, 2007.

[55] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in *ASPLOS-XIV*, 2009.

[56] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *ISCA-44*, 2017.

[57] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer *et al.*, "Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions," in *MICRO-39*, 2006.

[58] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *MICRO-47*, 2014.

[59] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support," in *HPCA-23*, 2017.

[60] X. Wang and J. F. Martínez, "ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment," in *ASPLOS-XXI*, 2016.

[61] H. Wong, "Intel Ivy Bridge cache replacement policy," http://blog.stuffedcow.net/2013/01/ivb-cache-replacement, archived at https://perma.cc/M59C-HPBN, 2013.

[62] C.-J. Wu and M. Martonosi, "A comparison of capacity management schemes for shared CMP caches," in *WDDD-7*, 2008.

[63] M.-J. Wu and D. Yeung, "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," in *PACT-20*, 2011.

[64] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA-36*, 2009.

[65] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *PACT-23*, 2014.

[66] L. Zhang, Y. Liu, R. Wang, and D. Qian, "Lightweight dynamic partitioning for last-level cache of multicore processor on real system," *The Journal of Supercomputing*, vol. 69, no. 2, 2014.

[67] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," in *ASPLOS-XXI*, 2016.