Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-017 September 1, 2013

Jigsaw: Scalable Software-Defined Caches (Extended Version)

Nathan Beckmann and Daniel Sanchez

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Jigsaw: Scalable Software-Defined Caches (Extended Version)

Nathan Beckmann and Daniel Sanchez
Massachusetts Institute of Technology
{beckmann, sanchez}@csail.mit.edu

***Abstract*—Shared last-level caches, widely used in chip-multiprocessors (CMPs), face two fundamental limitations. First, the latency and energy of shared caches degrade as the system scales up. Second, when multiple workloads share the CMP, they suffer from interference in shared cache accesses. Unfortunately, prior research addressing one issue either ignores or worsens the other: NUCA techniques reduce access latency but are prone to hotspots and interference, and cache partitioning techniques only provide isolation but do not reduce access latency.**

We present Jigsaw, a technique that jointly addresses the scalability and interference problems of shared caches. Hardware lets software define *shares*, collections of cache bank partitions that act as virtual caches, and map data to shares. Shares give software full control over both data placement and capacity allocation. Jigsaw implements efficient hardware support for share management, monitoring, and adaptation. We propose novel resource-management algorithms and use them to develop a system-level runtime that leverages Jigsaw to both maximize cache utilization and place data close to where it is used.

We evaluate Jigsaw using extensive simulations of 16- and 64-core tiled CMPs. Jigsaw improves performance by up to 2.2× (18% avg) over a conventional shared cache, and significantly outperforms state-of-the-art NUCA and partitioning techniques.

***Index Terms*—cache, memory, NUCA, partitioning, isolation**

I. Introduction

Chip-multiprocessors (CMPs) rely on sophisticated on-chip cache hierarchies to mitigate the high latency, high energy, and limited bandwidth of off-chip memory accesses. Caches often take over 50% of chip area [21], and, to maximize utilization, most of this space is structured as a last-level cache shared among all cores. However, as Moore's Law enables CMPs with tens to hundreds of cores, shared caches face two fundamental limitations. First, the latency and energy of a shared cache degrade as the system scales up. In large chips with distributed caches, more latency and energy is spent on network traversals than in bank accesses. Second, when multiple workloads share the CMP, they suffer from interference in shared cache accesses. This causes large performance variations, precludes quality-of-service (QoS) guarantees, and degrades throughput. With the emergence of virtualization and cloud computing, interference has become a crucial problem in CMPs.

Ideally, a cache should both store data close to where it is used, and allow its capacity to be partitioned, enabling software to provide isolation, prioritize competing applications, or increase cache utilization. Unfortunately, prior research does not address both issues jointly. On one hand, prior non-uniform cache access (NUCA) work [2, 3, 8, 10, 11, 14, 18, 31, 33, 46] has proposed a variety of placement, migration, and replication policies to reduce network distance. However, these best-effort techniques often result in hotspots and additional interference [3]. On the other hand, prior work has proposed a variety of partitioning techniques [9, 25, 28, 42, 44], but these schemes only work on fully shared caches, often scale poorly beyond few partitions, and degrade throughput.

We present *Jigsaw*, a design that jointly addresses the scalability and interference problems of shared caches. On the hardware side, we leverage recent prior work on efficient fine-grained partitioning [37] to structure the last-level cache as a collection of distributed banks, where each bank can be independently and logically divided in many *bank partitions*. Jigsaw lets software combine multiple bank partitions into a logical, software-defined cache, which we call a *share*. By mapping data to shares, and configuring the locations and sizes of the individual bank partitions that compose each share, software has full control over both where data is placed in the cache, and the capacity allocated to it. Jigsaw efficiently supports reconfiguring shares dynamically and moving data across shares, and implements monitoring hardware to let software find the optimal share configuration efficiently.

On the software side, we develop a lightweight system-level runtime that divides data into shares and decides how to configure each share to both maximize cache utilization and place data close to where it is used. In doing so, we develop novel and efficient resource management algorithms, including *Peekahead*, an exact linear-time implementation of the previously proposed quadratic-time Lookahead algorithm [34], enabling global optimization with non-convex utilities on very large caches at negligible overheads.

We evaluate Jigsaw with simulations of 16- and 64-core tiled CMPs. On multiprogrammed mixes of single-threaded workloads, Jigsaw improves weighted speedup by up to 2.2× (18.4% gmean) over a shared LRU LLC, up to 35% (9.4% gmean) over Vantage partitioning [37], up to 2.05× (11.4% gmean) over R-NUCA [14], and up to 24% (6.3% gmean) over an idealized shared-private D-NUCA organization that uses twice the cache capacity [16]. Jigsaw delivers similar benefits on multithreaded application mixes, demonstrating that, given the right hardware primitives, software can manage large distributed caches efficiently.

A shorter version of this paper will appear in the proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013 [4]. This extended version provides a more detailed analysis of Peekahead's run-time and correctness, and contains additional experiments and results.

Scheme	High capacity	Low Latency	Capacity control	Isolation	Directory-less
Private caches	✗	✓	✗	✓	✗
Shared caches	✓	✗	✗	✗	✓
Partitioned shared caches	✓	✗	✓	✓	✓
Private-based D-NUCA	∽	✓	✓	✗	✗
Shared-based D-NUCA	∽	✓	✗	✗	✓
Jigsaw	✓	✓	✓	✓	✓

Table 1. Desirable properties achieved by main cache organizations.

II. Background and Related Work

This section presents the relevant prior work on multicore caching that Jigsaw builds and improves on: techniques to partition a shared cache, and non-uniform cache architectures. Table 1 summarizes the main differences among techniques.

A. Cache Partitioning

Cache partitioning requires a *partitioning policy* to select partition sizes, and a *partitioning scheme* to enforce them.

Partitioning schemes: A partitioning scheme should support a large number of partitions with fine-grained sizes, disallow interference among partitions, strictly enforce partition sizes, avoid hurting cache associativity or replacement policy performance, support changing partition sizes efficiently, and require small overheads. Achieving these properties is not trivial.

Several techniques rely on restricting the locations where a line can reside depending on its partition. Way-partitioning [9] restricts insertions from each partition to its assigned subset of ways. It is simple, but it supports a limited number of coarsely-sized partitions (in multiples of way size), and partition associativity is proportional to its way count, sacrificing performance for isolation. To avoid losing associativity, some schemes can partition the cache by sets instead of ways [35, 43], but they require significant changes to cache arrays. Alternatively, virtual memory and page coloring can be used to constrain the pages of a process to specific sets [25, 42]. While software-only, these schemes are incompatible with superpages and caches indexed using hashing (common in modern CMPs), and repartitioning requires costly recoloring (copying) of physical pages.

Caches can also be partitioned by modifying the allocation or replacement policies. These schemes avoid the problems with restricted line placement, but most rely on heuristics [28, 44, 45], which provide no guarantees and often require many more ways than partitions to work well. In contrast, Vantage [37] leverages the statistical properties of skew-associative caches [39] and zcaches [36] to implement partitioning efficiently. Vantage supports hundreds of partitions, provides strict guarantees on partition sizes and isolation, can resize partitions without moves or invalidations, and is cheap to implement (requiring $\approx 1\%$ extra state and negligible logic). For these reasons, Jigsaw uses Vantage to partition each cache bank, although Jigsaw is agnostic to the partitioning scheme.

Partitioning policies: Partitioning policies consist of a monitoring mechanism, typically in hardware, that profiles partitions, and a controller, in software or hardware, that uses this information and sets partition sizes to maximize some metric, such as throughput [34], fairness [25, 41], or QoS [23].

Utility-based cache partitioning (UCP) is a frequently used policy [34]. UCP introduces a *utility monitor* (UMON) per core, which samples the address stream and measures the partition's *miss curve*, i.e., the number of misses that the partition would have incurred with each possible number of allocated ways. System software periodically reads these miss curves and repartitions the cache to maximize cache utility (i.e., the expected number of cache hits). Miss curves are often not convex, so deriving the optimal partitioning is NP-hard. UCP decides partition sizes with the Lookahead algorithm, an $O(N^2)$ heuristic that works well in practice, but is too slow beyond small problem sizes. Although UCP was designed to work with way-partitioning, it can be used with other schemes [37, 45]. Instead of capturing miss curves, some propose to estimate them with analytical models [41], use simplified algorithms, such as hill-climbing, that do not require miss curves [28], or capture them offline [6], which simplifies monitoring but precludes adaptation. Prior work has also proposed approximating miss curves by their convex fits and using efficient convex optimization instead of Lookahead [6].

In designing Jigsaw, we observed that miss curves are often non-convex, so hill-climbing or convex approximations are insufficient. However, UCP's Lookahead is too slow to handle large numbers of fine-grained partitions. To solve this problem, we reformulate Lookahead in a much more efficient way, making it linear-time (Sec. IV).

B. Non-Uniform Cache Access (NUCA) Architectures

NUCA techniques [20] reduce the access latency of large distributed caches, and have been the subject of extensive research. Static NUCA (S-NUCA) [20] simply spreads the data across all banks with a fixed line-bank mapping, and exposes a variable bank access latency. Commercial designs often use S-NUCA [21]. Dynamic NUCA (D-NUCA) schemes improve on S-NUCA by adaptively placing data close to the requesting core [2, 3, 8, 10, 11, 14, 18, 31, 33, 46]. They involve a combination of *placement*, *migration*, and *replication* strategies. Placement and migration dynamically place data close to cores that use it, reducing access latency. Replication makes multiple copies of frequently used lines, reducing latency for widely read-shared lines (e.g., hot code), at the expense of some capacity loss.

Shared- vs private-based NUCA: D-NUCA designs often build on a *private-cache baseline*. Each NUCA bank is treated as a private cache, lines can reside in any bank, and coherence is preserved through a directory-based or snoopy protocol, which is often also leveraged to implement NUCA techniques. For example, Adaptive Selective Replication [2] controls replication by probabilistically deciding whether to store a copy of a remotely fetched line in the local L2 bank; Dynamic Spill-Receive [33] can spill evicted lines to other banks, relying on

remote snoops to retrieve them. These schemes are flexible, but they require all LLC capacity to be under a coherence protocol, so they are either hard to scale (in snoopy protocols), or incur significant area, energy, latency, and complexity overheads (in directory-based protocols).

In contrast, some D-NUCA proposals build on a *shared-cache baseline* and leverage virtual memory to perform adaptive placement. Cho and Jin [11] use page coloring and a NUCA-aware allocator to map pages to specific banks. Hardavellas et al. [14] find that most applications have a few distinct classes of accesses (instructions, private data, read-shared, and write-shared data), and propose R-NUCA, which specializes placement and replication policies for each class of accesses on a per-page basis, and significantly outperforms NUCA schemes without this access differentiation. Shared-baseline schemes are simpler, as they require no coherence for LLC data and have a simpler lookup mechanism. However, they may incur significant overheads if remappings are frequent or limit capacity due to restrictive mappings (Sec. VI).

Jigsaw builds on a shared baseline. However, instead of mapping pages to locations as in prior work [11, 14], we map pages to *shares* or logical caches, and decide the physical configuration of the shares independently. This avoids page table changes and TLB shootdowns on reconfigurations, though some reconfigurations still need cache invalidations.

Isolation and partitioning in NUCA: Unlike partitioning, most D-NUCA techniques rely on best-effort heuristics with little concern for isolation, so they often improve typical performance at the expense of worst-case degradation, further precluding QoS. Indeed, prior work has shown that D-NUCA often causes significant bank contention and uneven distribution of accesses across banks [3]. We also see this effect in Sec. VI — R-NUCA has the highest worst-case degradation of all schemes. Dynamic Spill-Receive mitigates this problem with a QoS-aware policy that avoids spills to certain banks [33]. This can protect a local bank from interference, but does not provide partitioning-like capacity control. Virtual Hierarchies rely on a logical two-level directory to partition a cache at bank granularity [29], but this comes at the cost of doubling directory overheads and making misses slower.

Because conventional partitioning techniques (e.g., way-partitioning) only provide few partitions and often degrade performance, D-NUCA schemes seldom use them. ASP-NUCA [12], ESP-NUCA [31], and Elastic Cooperative Caching [16] use way-partitioning to divide cache banks between private and shared levels. However, this division does not provide isolation, since applications interfere in the shared level. In contrast, Jigsaw partitions the cache into multiple isolated virtual caches that, due to smart placement, approach the low latency of private caches. These schemes often size partitions using hill-climbing (e.g., shadow tags [12] or LRU way hit counters [16]), which can get stuck in local optima, whereas Jigsaw captures full miss curves to make global decisions.

CloudCache [22] implements virtual private caches that can span multiple banks. Each bank is way-partitioned, and partitions are sized with a distance-aware greedy algorithm based on UCP with a limited frontier. Unfortunately, CloudCache scales poorly to large virtual caches, as it uses N-chance spilling on evictions, and relies on broadcasts to serve local bank misses, reducing latency at the expense of significant bandwidth and energy (e.g., in a 64-bank cache with 8-way banks, in a virtual cache spanning all banks, a local miss will trigger a full broadcast, causing a 512-way lookup and a chain of 63 evictions). In contrast, Jigsaw implements *single-lookup* virtual *shared* caches, providing coordinated placement and capacity management without the overheads of a globally shared directory or multi-level lookups, and performs global (not limited-frontier) capacity partitioning efficiently using novel algorithms (Sec. IV).

III. Jigsaw Hardware

Jigsaw exposes on-chip caches to software and enables their efficient management using a small set of primitives. First, Jigsaw lets software explicitly divide a distributed cache in collections of *bank partitions*, which we call *shares*. Shares can be dynamically reconfigured by changing the size of each bank partition. Second, Jigsaw provides facilities to map data to shares, and to quickly migrate data among shares. Third, Jigsaw implements share monitoring hardware to let software find the optimal share configuration efficiently.

A. Shares

Fig. 1 illustrates the overall organization of Jigsaw. Jigsaw banks can be divided in *bank partitions*. Jigsaw is agnostic to the partitioning scheme used, as well as the array type and replacement policy. As discussed in Sec. II, in our evaluation we select Vantage partitioning due to its ability to partition banks at a fine granularity with minimal costs.

Shares are configurable collections of bank partitions, visible to software. Each share has a unique id number and comprises a set of bank partitions that can be sized independently. The share size is the sum of its bank partition sizes. The share id is independent from the individual partition ids.

We could exploit shares in two ways. On the one hand, we could assign cores to shares, having shares behave as virtual private caches. This is transparent to software, but would require a coherence directory for LLC data. On the other hand, we can map data to shares. This avoids the need for coherence beyond the private (L2) caches, as each line can only reside in a single location. Mapping data to shares also enables specializing shares to different types of data (e.g., shared vs thread-private [14]). For these reasons, we choose to map data to shares.

Jigsaw leverages the virtual memory subsystem to map data to shares. Fig. 1 illustrates this implementation, highlighting the microarchitectural structures added and modified. Specifically, we add a share id to each page table entry, and extend the TLB to store the share id. Active shares must have unique ids, so we model 16-bit ids. Share ids are needed in L2 accesses, so these changes should not slow down page translations.

On a miss on the private cache levels, a per-core *share-bank translation buffer* (STB) finds the bank the line maps

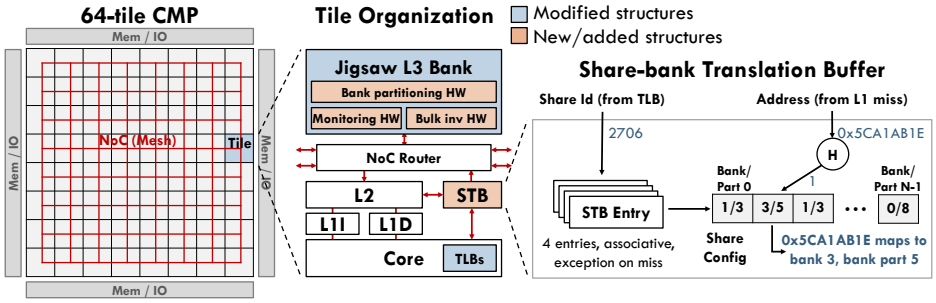


Figure 1. Jigsaw overview: target tiled CMP, tile configuration with microarchitectural changes and additions introduced by Jigsaw, and Share-Bank Translation Buffer (STB).

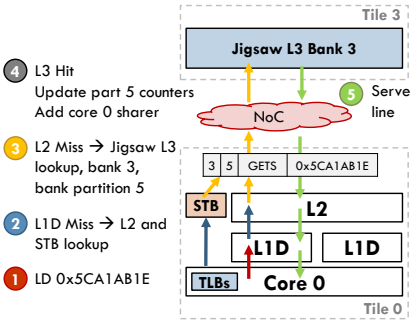


Figure 2. Jigsaw L3 access, including STB lookup in parallel with L2 access.

to, as well as its bank partition. Fig. 1 depicts the per-core STBs. Each STB has a small number of resident shares. Like in a software-managed TLB, an access to a non-resident share causes an exception, and system software can refill the STB. As we will see in Sec. IV, supporting a small number of resident shares per core (typically 4) is sufficient. Each share descriptor consists of an array of N bank and bank partition ids. To perform a translation, we hash the address, and use the hash value to pick the array entry used. We take the STB translation latency out of the critical path by doing it speculatively on L2 accesses. Fig. 2 details the different steps involved in a Jigsaw cache access.

There are several interesting design dimensions in the STB. First, the hash function can be as simple as bit-selection. However, to simplify share management, the STB should divide the requests into sub-streams *with statistically similar access patterns*. A more robust hash function can achieve this. Specifically, we use an H_3 hash function (H in Fig. 1), which is universal and efficient to implement in hardware [7]. All STBs implement the same hash function. Second, increasing N, the number of entries in a share descriptor, lets us fine-tune the load we put on each bank to adapt to heterogeneous bank partition sizes. For example, if a share consists of two bank partitions, one twice the size of the other, we'd like 66% of the requests to go to the larger bank partition, and 33% to the smaller one. $N = 2$ does not allow such division, but $N = 3$ does. In our implementation, we choose N equal to the number of banks, so shares spanning few bank partitions can be finely tuned to bank partition sizes, but large shares that span most banks can not. For a 64-core system with 64 banks in which each bank has 64 partitions, bank and bank partition ids are 6 bits, and each share descriptor takes 768 bits (96 bytes). Supporting four shares can be done with less than 400 bytes, a 0.2% storage overhead over the private cache sizes. Alternatively, more complex weighted hash functions or more restrictive mappings can reduce this overhead.

B. Dynamic Adaptation

So far we have seen how Jigsaw works on a static configuration. To be adaptive, however, we must also support both reconfiguring a share and remapping data to another share.

Share reconfiguration: Shares can be changed in two dimensions. First, per-bank partition sizes can be dynamically changed. This concerns the bank partitioning technique used (e.g., in Vantage, this requires changing a few registers [37]), and is transparent to Jigsaw. Second, the share descriptor (i.e., the mapping of lines to bank partitions) should also be changed at runtime, to change either the bank partitions that conform the share, or the load put on each bank partition.

To support share descriptor reconfiguration, we introduce hardware support for bulk invalidations. On a reconfiguration, the new STB descriptors are loaded and each bank walks the whole array, invalidating lines from shares that have been reassigned to other banks. When a bulk invalidation is in progress, accesses to lines in the same bank partition are NACKed, causing an exception at the requesting core. This essentially quiesces the cores that use the bank partition until the invalidation completes.

Bulk invalidations may seem heavy-handed, but they avoid having a directory. We have observed that bulk invalidations take 30-300 K cycles. Since we reconfigure every 50 M cycles, and only a fraction of reconfigurations cause bulk invalidations, this is a minor overhead given the hardware support. For our benchmarks, more frequent reconfigurations show little advantage, but this may not be the case with highly variable workloads. We defer investigating additional mechanisms to reduce the cost of bulk invalidations (e.g., avoiding stalls or migrating instead of invalidating) to future work.

These tradeoffs explain why we have chosen partitionable banks instead of a large number of tiny, unpartitionable banks. Partitionable banks incur fewer invalidations, and addressing a small number of banks reduces the amount of state in the share descriptor and STB. Finally, increasing the number of banks would degrade NoC performance and increase overheads [26].

Page remapping: To classify pages dynamically (Sec. IV), software must also be able to remap a page to a different share. A remap is similar to a TLB shootdown: the initiating core quiesces other cores where the share is accessible with an IPI; it then issues a bulk invalidation of the page. Once all the banks involved finish the invalidation, the core changes the share in the page table entry. Finally, quiesced cores update the stale TLB entry before resuming execution. Page remaps typically take a few hundred cycles, less than the associated TLB shootdown, and are rare in our runtime, so their performance effects are negligible.

Invalidations due to both remappings and reconfigurations

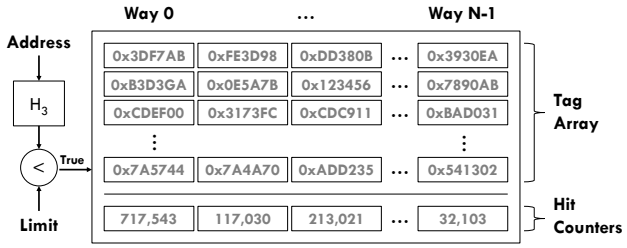


Figure 3. Jigsaw monitoring hardware. As in UMON-DSS [34], the tag array samples accesses and counts hits per way to produce miss curves. The limit register finely controls the UMON's sampling rate.

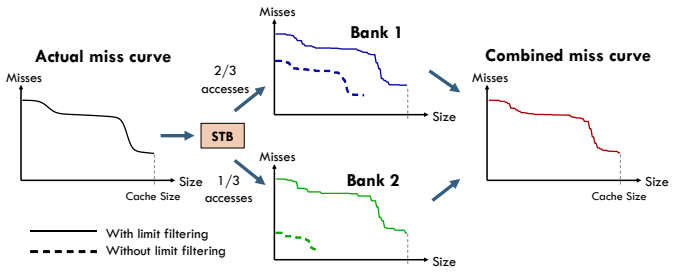


Figure 4. Example sampling for a share with two bank partitions, where the first bank receives twice the traffic. Configuration and combining are done in software at reconfiguration time.

could be avoided with an extra directory between Jigsaw and main memory. Sec. VI shows that this is costly and not needed, as reconfiguration overheads are negligible.

C. Monitoring

In order to make reasonable partitioning decisions, software needs monitoring hardware that gives accurate, useful and timely information. As discussed in Sec. II, utility monitors (UMONs) [34] are an efficient way to gather miss curves. Prior partitioning schemes use per-core UMONs [34, 37], but this is insufficient in Jigsaw, as shares can be accessed from multiple cores, and different cores often have wildly different access patterns to the same data. Instead, Jigsaw generates per-share miss curves by adding UMONs to each bank.

UMONs were originally designed to work with set-associative caches, and worked by sampling a small but statistically significant number of sets. UMONs can also be used with other cache designs [37] by sampling a fraction of cache accesses at the UMON. Given high enough associativity, a sampling ratio of UMON lines : S behaves like a cache of S lines. Moreover, the number of UMON ways determines the resolution of the miss curve: an N-way UMON yields $N+1$-point miss curves.

Partitioning schemes with per-core UMONs implicitly use a fixed sampling ratio UMON lines : cache lines. This is insufficient in Jigsaw, because shares can span multiple banks. To address this, we introduce an *adaptive sampling* mechanism, shown in Fig. 3. Each UMON has a 32-bit *limit register*, and only addresses whose hash value is below this limit are inserted into the UMON. Changing the limit register provides fine control over the UMON's sampling rate.

For single-bank shares, a ratio $r_0 =$ UMON lines : LLC lines lets Jigsaw model the full cache, but this is inadequate for multi-bank shares. To see why, consider a share allocated 100 KB, split between two bank partitions allocated 67 KB and 33 KB. The STB spreads accesses across banks, so each bank sees a statistically similar request stream, but sampled proportionally to bank partition size: $2/3$ of the accesses are sent to the first partition, and $1/3$ to the second. Consequently, the first bank partition's UMON would behave like a cache of $1.5\times$ the LLC size, and the second as a cache of $3\times$ the LLC size. Using a fixed sampling ratio of r_0 would be wasteful. By using $r_1 = 3/2 \cdot r_0$ and $r_2 = 3 \cdot r_0$, *Jigsaw counters the sampling introduced by the STB*, and both UMONs model LLC size precisely.

These issues appear visually in the miss curves, shown in Fig. 4. The miss curve of the application is shown on the left in black. Following the previous example, bank one receives $2/3$ of accesses and, without correcting the sampling ratio, models a $1.5\times$ larger cache. As a result, the application's miss curve would be scaled (dashed lines) and terminate at two-thirds of cache size. By correcting the sampling ratio (solid line), Jigsaw reverses this effect and re-scales the miss curve to its true dimensions. Bank two is a more extreme example since it receives $1/3$ of accesses, and its sampled miss curve would be scaled $3\times$ smaller. These problems could be corrected by upsampling in software, but doing so would lessen sampling quality (by wasting UMON ways) and unnecessarily complicate re-combination.

In general, if the STB sends a fraction f_i of requests to bank partition i, then a sampling ratio $r_i = r_0/f_i$ models LLC capacity, and Jigsaw produces the share's miss curve by averaging the bank partitions' curves. Moreover, when shares span multiple banks, one or a few UMONs suffice to capture accurate miss curves. Jigsaw therefore only implements a few UMONs per bank (four in our evaluation) and dynamically assigns them to shares using a simple greedy heuristic, ensuring that each share has at least one UMON. This makes the number of UMONs scale with the number of shares, not bank partitions.

Finally, in order to make sound partitioning decisions, miss curves must have sufficiently high resolution, which is determined by the number of UMON ways. While a small number of UMON ways is sufficient to partition small caches as in prior work [34, 37], partitioning a large, multi-banked cache among many shares requires higher resolution. For example, for a 1 MB cache, a 32-way UMON has a resolution of 32 KB. In a 64-bank cache with 1 MB banks, on the other hand, the same UMON's resolution is 2 MB. This coarse resolution affects partitioning decisions, hurting performance, as our evaluation shows (Sec. VI-D). For now, we ameliorate this problem by implementing 128-way UMONs and linearly interpolating miss curves. Though high, this associativity is still practical since UMONs only sample a small fraction of accesses. Results show that even higher associativities are beneficial, although this quickly becomes impractical. We defer efficient techniques for producing higher-resolution miss curves to future work.

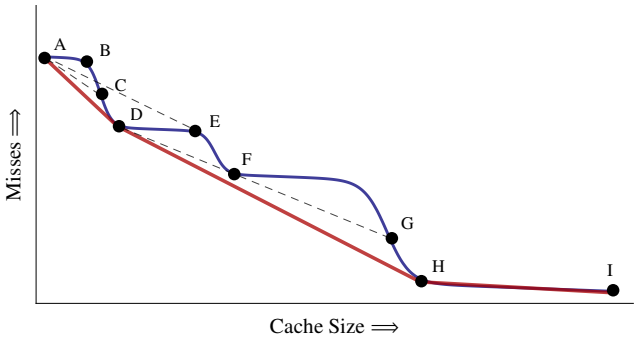


Figure 5. Non-convex miss curve (blue), and its convex hull (red).

		Maximum allocation, S'				
		$S' < D$	$D \leq S' < F$	$F \leq S' < G$	$G \leq S' < H$	$H \leq S'$
Start	A	S'	D	D	D	D
	D	-	S'	F	S'	H
	F	-	-	S'	-	-
	H	-	-	-	-	S'

Table 2. Maximal utility allocations for Fig. 5 across the entire domain from all possible starting positions.

IV. Jigsaw Software

Shares are a general mechanism with multiple potential uses (e.g., maximizing throughput or fairness, providing strict process isolation, implementing virtual local stores, or avoiding side-channel attacks). In this work, we design a system-level runtime that leverages Jigsaw to jointly improve cache utilization and access latency transparently to user-level software. The runtime first classifies data into shares, then periodically decides how to size and where to place each share.

A. Shares and Page Mapping

Jigsaw defines three types of shares: global, per-process, and per-thread. Jigsaw maps pages accessed by multiple processes (e.g., OS code and data, library code) to the (unique) global share. Pages accessed by multiple threads in the same process are mapped to a per-process share. Finally, each thread has a per-thread share. With this scheme, each core's STB uses three entries, but there are a large number of shares in the system.

Similar to R-NUCA [14], page classification is done incrementally and lazily, at TLB/STB miss time. When a thread performs the first access to a page, it maps it to its per-thread share. If another thread from the same process tries to access the page, the page is remapped to the per-process share. On an access from a different process (e.g., due to IPC), the page is remapped to the global share. When a process finishes, its shares are deallocated and bulk-invalidated.

B. Share Sizing: Peekahead

The Jigsaw runtime first decides how to size each share, then where to place it. This is based on the observation that reducing misses often yields higher benefits than reducing access latency, and considering sizing and placement independently greatly simplifies allocation decisions. Conceptually, sizing shares is no different than in UCP: the runtime computes the per-share miss curves as explained in Sec. III, then runs Lookahead (Sec. II) to compute the share sizes that maximize utility, or number of hits.

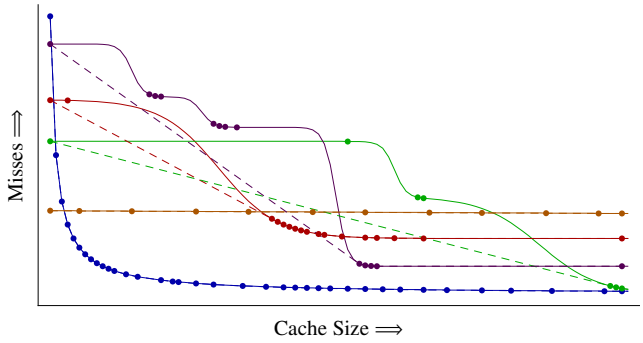


Figure 6. Points of interest (POIs) for several example miss curves. Dashed lines denote their convex hulls.

Unfortunately, using Lookahead is unfeasible. Lookahead greedily allocates space to the partition that provides the highest *utility per unit* (hits per allocation quantum). Because miss curves are not generally convex, on each step Lookahead traverses each miss curve looking for the maximum utility per unit it can achieve with the remaining unallocated space. This results in an $O(P \cdot S^2)$ run-time, where P is the number of partitions and S is the cache size in allocation quanta, or "buckets". With way-partitioning, S is small (the number of ways) and this is an acceptable overhead. In Jigsaw, banks can be finely partitioned, and we must consider all banks jointly. Lookahead is too inefficient at this scale.

To address this, we develop the *Peekahead* algorithm, an exact $O(P \cdot S)$ implementation of Lookahead. We leverage the insight that the point that achieves the maximum utility per unit is the next one in the *convex hull* of the miss curve. For example, Fig. 5 shows a non-convex miss curve (blue) and its convex hull (red). With an unlimited budget, i.e., abundant unallocated cache space, and starting from A, D gives maximal utility per unit (steepest slope); starting from D, H gives the maximal utility per unit; and so on along the convex hull. With a limited budget, i.e. if the remaining unallocated space limits the allocation to S', the point that yields maximum utility per unit is the next one in the convex hull of the miss curve in the region $[0, S']$. For example, if we are at D and are given limit S' between F and G, the convex hull up to S' is the line $\overline{DFS'}$ and F yields maximal utility per unit. Conversely, if S' lies between G and H, then the convex hull is $\overline{DS'}$, S' is the best option, and the algorithm terminates (all space is allocated).

If we know these *points of interest* (POIs), the points that constitute all reachable convex hulls, traversing the miss curves on each allocation becomes unnecessary: given the current allocation, the next relevant POI always gives the maximum utility per unit. For example, in Fig. 5, the only POIs are A, D, F, and H; Table 2 shows all possible decisions. Fig. 6 shows several example miss curves and their POIs. Note that some POIs do not lie on the full convex hull (dashed lines), but are always on the convex hull of some sub-domain.

Peekahead first finds all POIs in $O(S)$ for each partition. This is inspired by the three coins algorithm [30]. For example, we construct the convex hull $\overline{ADHI}$ in Fig. 5 by considering

Algorithm 1. The Peekahead algorithm. Compute all reachable convex hulls and use the convexity property to perform Lookahead in linear time. Letters in comments refer to points in Fig. 5. $\overline{AB}$ is the line connecting A and B.

Inputs: A single miss curve: M, Cache size: S
Returns: POIs comprising all convex hulls over $[0, X]\ \forall\ 0 \le X \le S$

```
1:  function AllHulls(M, S)
2:      start ← (0, M(0), ∞)                          ▷ POIs are (x, y, horizon)
3:      pois[...] ← {start}                            ▷ Vector of POIs
4:      hull[...] ← {pois.Head}      ▷ Current convex hull; references pois
5:      for x ← 1 to S:
6:          next ← (x, M(x), ∞)
7:          for i ← hull.Length − 1 to 1:              ▷ Backtrack?
8:              candidate ← hull[i]
9:              prev ← hull[i − 1]
10:             if candidate is not Below prev next:
11:                 hull.PopBack()                     ▷ Remove from hull
12:                 if candidate.x ≥ x − 1:
13:                     pois.PopBack()                 ▷ Not a POI (C, G)
14:                 else:
15:                     candidate.horizon ← x − 1      ▷ POI not on hull (F)
16:             else:
17:                 break          ▷ POI and predecessors valid (for now)
18:         pois.PushBack(next)                        ▷ Add POI
19:         hull.PushBack(pois.Tail)
20:     return pois
```

Inputs: Partition miss curves: $M_1 ... M_P$, Cache size: S
Returns: Partition allocations: A[...]

```
21: function Peekahead(M_1 ... M_P, S)
22:     pois[...] ← {AllHulls(M_1, S)...AllHulls(M_P, S)}
23:     current[...] ← {pois[1].Head...pois[p].Head}   ▷ Allocations
                       P times
24:     A[...] ← {0...0}
25:     heap ← MakeHeap()                              ▷ Steps sorted by ΔU
26:     function NextPOI(p)
27:         for i ← current[p] + 1 to pois[p].Tail:
28:             if i.x > current[p].x + S: break       ▷ No space left
29:             if i.horizon > current[p].x + S: return i   ▷ Valid POI
30:         x ← current[p].x + S                       ▷ Concave region; take S
31:         return (x, M_p(x), ∞)
32:     function Enqueue(p)
33:         next ← NextPOI(p)
34:         ΔS ← next.x − current[p].x
35:         ΔU ← (current[p].y − next.y) /ΔS
36:         heap.Push((p, ΔU, ΔS, next))
37:     Enqueue([1...P])
38:     while S > 0:                                   ▷ Main loop
39:         (p, ΔU, ΔS, next) ← heap.Pop()
40:         if S ≥ ΔS:                      ▷ Allocate if we have space
41:             current[p] ← next
42:             A[p] ← A[p] + ΔS
43:             S ← S − ΔS
44:         Enqueue(p)
45:     return A[...]
```

points from left to right. At each step, we add the next point to the hull, and then backtrack to remove previous points that no longer lie on the hull. We begin with the line $\overline{AB}$. C is added to form $\overline{ABC}$, and then we backtrack. Because B lies above $\overline{AC}$, it is removed, leaving $\overline{AC}$. Similarly, D replaces C, leaving $\overline{AD}$. Next, E is added to form $\overline{ADE}$, but since D lies below $\overline{AE}$, it is *not* removed. Continuing, F replaces E, G replaces F, H replaces G, and finally I is added to give the convex hull $\overline{ADHI}$.

We extend this algorithm to build all convex hulls over $[0, X]$ for any X up to S, which produces all POIs. We achieve this in $O(S)$ by not always deleting points during backtracking. Instead, we mark points in convex regions with the x-coordinate at which the point becomes obsolete, termed the *horizon* (e.g., F's horizon is G). Such a point is part of the convex hull up to its horizon, after which it is superseded by the higher-utility-per-unit points that follow. However, if a point is in a concave region then it is not part of any convex hull, so it is deleted (e.g., C and G).

Algorithm 2. Jigsaw's partitioning policy. Divide the LLC into shares to maximize utility and locality. Shares use budgets produced by Peekahead to claim capacity in nearby bank partitions in increments of Δ_0.

Inputs: Partition miss curves: $M_1 ... M_P$, Cache size: S, Num. banks: B, Banks sorted by distance: $D_1 ... D_P$
Returns: Share allocation matrix: $[A_{p,b}]$ where $1 \le p \le P$ and $1 \le b \le B$

```
1:  function Partition(M_1 ... M_P, S, B)
2:      budget[...] ← Peekahead(M_1 ... M_P, S)
                                  B times
3:      inventory[...] ← {S/B, S/B, S/B ... S/B}
4:      d[...] ← {D_1.Head ... D_P.Head}               ▷ Prefer closer banks
5:      A ← [0]_{1≤p≤P, 1≤b≤B}
6:      while Σ budget > 0:
7:          for s ← 1 to P:
8:              b ← d[i]                               ▷ Closest bank
9:              if inventory[b] > Δ_0:
10:                 Δ ← Δ_0                            ▷ Have space; take Δ_0
11:             else:
12:                 Δ ← inventory[b]   ▷ Empty bank; move to next closest
13:                 d[i] ← d[i]+1
14:             A_{p,b} ← A_{p,b} + Δ
15:             budget[s] ← budget[s] − Δ
16:             inventory[b] ← inventory[b] − Δ
17:     return A
```

1) Algorithm Listings: Algorithm 1 shows the complete Peekahead algorithm. First, AllHulls preprocesses each share's miss curve and computes its POIs. Then, Peekahead divides cache space across shares iteratively using a max-heap. In practice, AllHulls dominates the run-time of Algorithm 1 at $O(P \cdot S)$, as Sec. VI-D confirms.

AllHulls: In AllHulls, the procedure is as described previously: Two vectors are kept, pois and hull. pois is the full list of POIs whereas hull contains only the points along the current convex hull. Each POI is represented by its coordinates and its horizon, which is initially ∞.

The algorithm proceeds as described in the example. Points are always added to the current hull (lines 18-19). Previous points are removed from hull by backtracking (lines 7-17). A point (x, y) is removed from pois when it lies in a concave region. This occurs when it is removed from hull at $x+1$ (line 13). Otherwise, the point lies in a convex region and is part of a sub-hull, so it its horizon is set to x, but it is not deleted from pois (line 15).

Peekahead: Peekahead uses POIs to implement lookahead. POIs for partition p are kept in vector pois[p], and current[p] tracks its current allocation. heap is a max-heap of the maximal utility per unit steps for each partition, where a step is a tuple of: (partition, utility per unit, step size, POI).

Peekahead uses two helper routines. NextPOI gives the

next relevant POI for a partition from its current allocation and given remaining capacity. It simply iterates over the POIs following current[p] until it reaches capacity (line 28), a valid POI is found (line 29), or no POIs remain (line 31) which indicates a concave region, so it claims all remaining capacity. ENQUEUE takes the next POI and pushes a step onto heap. This means simply computing the utility per unit and step size.

The main loop repeats until capacity is exhausted taking steps from the top of heap. If capacity has dropped below the best next step's requirements, then the step is skipped and the best next step under current capacity enqueued (line 40).

2) Correctness: The correctness of Algorithm 1 relies on ALLHULLS constructing all relevant convex hulls and PEEKAHEAD always staying on a convex hull. The logic is as follows:

- The highest utility per unit step for a miss curves comes from the next point on its convex hull.
- If a point (x, y) is on the convex hull of $[0, Y]$, then it is also on the convex hull of $[0, X]$ for any $x \leq X \leq Y$.
- POIs from ALLHULLS and repeated calls to NEXTPOI with capacity $0 \leq S' \leq S$ produce the convex hull over $[0, S']$.
- Remaining capacity S' is decreasing, so current[p] always lies on the convex hull over $[0, S']$.
- Therefore PEEKAHEAD always makes the maximum-utility move, and thus correctly implements lookahead.

Definition 1. A *miss curve* is a decreasing function $M : [0, S] \rightarrow \mathbb{R}$.

Definition 2. Given a miss curve M and cache sizes x, y, the *utility per unit* between x and y is defined as:

$$\Delta U(M, x, y) = \frac{M(y) - M(x)}{x - y}$$

Which in plain language is simply the number of additional hits per byte gained between x and y.

We now define what it means to implement the lookahead algorithm. Informally, a lookahead algorithm is one that successively chooses the maximal utility per unit allocation until all cache space is exhausted.

Definition 3. Given P partitions, miss curves $M_1 ... M_P$, remaining capacity, S' and current allocations $x_1 ... x_P$, a *lookahead move* is a pair (p, s) such that partition p and allocation $0 < s \leq S'$ give maximal utility per unit:

$$\forall q : \Delta U(M_p, x_p, x_p + s) \geq \Delta U(M_q, x_q, x_q + s)$$

Definition 4. Given P partitions, miss curves $M_1 ... M_P$, cache size S, a *lookahead sequence* is a sequence $L = (p_1, s_1), (p_2, s_2) ... (p_N, s_N)$ such that the *allocations for p at i*, $x_{p,i} = \sum_{i=1}^{N} \{s_i : p_i \equiv p\}$, and *remaining capacity at i*, $S_i = \sum_{i=1}^{N} s_i$, satisfy the following:

1) For all i, (p_i, s_i) is a lookahead move for $M_1 ... M_P$, $S' = S - S_i$, and $x_{1,i} ... x_{P,i}$.
2) $S_N = S$.

Definition 5. A *lookahead algorithm* is one that produces a lookahead sequence for all valid inputs: $M_1 ... M_P$, and S.

Theorem 1. *UCP Lookahead is a lookahead algorithm.*

Proof: UCP Lookahead is the literal translation of the definition. At each step, it scans each partition to compute the maximum utility per unit move from that partition. It then takes the maximum utility per unit move across all partitions. Thus at each step is makes lookahead moves, and proceeds until capacity is exhausted. ■

We now show that PEEKAHEAD implements lookahead.

Lemma 1. *Given a miss curve M and starting from point A on the convex hull over $[0, S]$, the first point on the miss curve that yields maximal utility per unit is the next point on the miss curve's convex hull.*

Proof: Let the convex hull be H and the next point on the convex hull be B.

Utility per unit is defined as negative slope. Thus, maximal utility per unit equals minimum slope. Convex hulls are by definition convex, so their slope is non-decreasing. The minimum slope of the convex hull therefore occurs at A. Further, the slope of the hull between A and B is constant, so B yields the maximal utility per unit.

To see why it is the first such point, consider that all points on M between A and B lie above H. If this were not the case for some C, then the slope between A and C would be less than or equal to that between A and B, and C would lie on H. Therefore since all points lie above H, none can yield the maximal utility per unit. ■

Lemma 2. *For any miss curve M, if (x, y) is on H_Y, the convex hull of M over $[0, Y]$ then it is also on H_X, the convex hull of M over $[0, X]$, for any $x \leq X \leq Y$.*

Proof: By contradiction. Assume (x, y) is on H_Y but not on H_X. Then there must exist on $M : [0, X] \rightarrow \mathbb{R}$ points $A = (a, M(a))$ and $B = (b, M(b))$ such that $a < x < b$ and (x, y) lies above the line $\overline{AB}$. But $[0, X] \subset [0, Y]$ so A and B are also in $[0, Y]$. Thus (x, y) is not on H_Y. ■

Lemma 3. *In ALLHULLS on a miss curve M over domain $[0, S]$, upon completion of iteration $x \equiv X$, hull contains H_X, the convex hull of M over $[0, X]$.*

Proof: All points in $[0, X]$ are added to hull, so we only need to prove that all points *not* on the hull are removed. By the definition of a convex hull, a point $(a, M(a))$ is on H_X iff it is not below the line connecting any two other points on $[0, X]$. Now by induction:

Base case: At $X = 1$, hull contains the points $(0, M(0))$ and $(1, M(1))$. These are the only points on $[0, 1]$, so hull contains H_1.

Induction: Assume that hull contains H_X. Show that after iteration $X + 1$, hull contains H_{X+1}. This is guaranteed by backtracking (lines 7-17), which removes all points on H_X not on H_{X+1}.

Let $H_X = \{h_1, h_2 ... h_N\}$ and $h_{N+1} = (X + 1, M(X + 1))$. Then next $\equiv h_{N+1}$ and, at the first backtracking iteration, candidate $\equiv h_N$ and prev $\equiv h_{N-1}$.

First note that $H_{X+1} \subset H_X \cup \{h_{N+1}\}$. This follows from Lemma 2; any point on H_{X+1} other than h_{N+1} *must* lie on H_X.

Next we show that backtracking removes all necessary points by these propositions:

(i) *If* candidate *lies is on* H_{X+1} *then all preceding points are on* H_{X+1}.

Backtracking stops when a single point is valid (line 17). We must show that no points that should be removed from hull are missed by terminating the loop. This is equivalent to showing

$$h_i \notin H_{X+1} \Rightarrow h_{i+1}...h_N \notin H_{X+1}$$

Let h_i be the first point on H_X not on H_{X+1}. Then there exists $i < k \leq N+1$ such that h_i lies above $\ell_k = \overline{h_{i-1}h_k}$. Further, all h_j for $i < j \leq N$ lie above $\ell_i = \overline{h_{i-1}h_i}$. But since h_i lies above ℓ_k, ℓ_i is above ℓ_k to the right of h_i. Since all $i < j \leq N$ lie above ℓ_i, $k = N+1$. Therefore all h_j for $i < j \leq N$ lie above $\overline{h_{i-1}h_{N+1}}$.

(ii) *Points below* prev next *lie on* H_{X+1}.

Backtracking removes all points that do not lie below prev next. We must show this is equivalent to testing all pairs of points in $[0, X]$.

Note that for all $1 \leq i \leq N$, $M : [0, X] \to \mathbb{R}$ lies on or above $\overline{h_i h_{i+1}}$ by the definition of a convex hull.

Consider cases if candidate $\in H_{X+1}$:

candidate $\in H_{X+1}$: By the first proposition, prev is also on H_{X+1}. So by the previous note, M lies above both prev candidate and candidate next. Therefore candidate lies below prev next.

candidate $\notin H_{X+1}$: If prev $\in H_{X+1}$ then by definition of a convex hull, candidate lies above prev next. If not, then let h_i be the last point on H_{X+1} before candidate and h_k the first point on H_{X+1} after candidate. Then candidate lies above $\overline{h_i h_k}$. Note that $h_k \equiv$ next by the previous proposition.

h_i is also before prev because no points on H_X lie between candidate and prev. Furthermore, since next lies below $\overline{h_i h_{i+1}}$ but above $\overline{h_{i-1}h_i}$, $\overline{h_{i+1}}$ next lies above $\overline{h_i}$ next. H_X has non-decreasing slope and M is decreasing, so $\overline{h_j}$ next lies above $\overline{h_{j-1}}$ next for all $i < j \leq N$. In particular, candidate next lies above prev next and thus candidate lies above prev next.

$\therefore$ candidate $\in H_{X+1} \Leftrightarrow$ candidate is below prev next

(iii) *Backtracking removes all points on* H_X *not on* H_{X+1}.

Backtracking continues until candidate is on H_{X+1} or hull is empty. By the previous two propositions, this removes all such points.

Finally, h_{N+1} is added to hull (line 19), and hull contains H_{X+1}.

$\therefore$ hull $\equiv H_X$. ∎

Lemma 4. ALLHULLS *on a miss curve* M *over domain* $[0, S]$ *produces POIs for which* H_X*, the convex hull over* $[0, X]$*, is equivalent to:*

$$h_i = \begin{cases} (0, M(0)) & \text{if } i \equiv 1 \\ \text{NEXTPOI}(h_{i-1}) & \text{otherwise} \end{cases}$$

Proof: Consider how NEXTPOI works. It simply filters pois to the set

$$V = \{p : p \in \text{pois and } p.x < X \text{ and } p.horizon > X\} \cup \{(X, M(X), \infty)\}$$

By Lemma 3, we must show V is the same as the contents of hull at iteration $x \equiv X$.

First note that whenever points are removed from hull, they are either removed from pois (line 13) or their horizon is set (line 15). If a point obsolesces at $x \equiv \chi$, then its horizon is always $\chi - 1$.

$V \subset H_X$: Show that all points in $[0, X]$ not in H_X are either deleted from pois or have their horizon set smaller than X. By Lemma 3, at completion of iteration $x \equiv X$, hull contains H_X. Therefore all points not on H_X have been removed from hull and are either deleted from pois or have horizon set to a value less than X.

$H_X \subset V$: Show that no point in H_X is deleted from pois or has a horizon set smaller than X.

Points are only deleted or have their horizon set when they are removed from hull. By Lemma 3, no point on H_X is removed from hull before $x \equiv X$. So the earliest a point in H_X could be removed from hull is at $X+1$, and its horizon would be X.

Furthermore no point earlier than X will be deleted at $X+1$, because ALLHULLS only deletes a point $(a, M(a))$ if it obsolesces at $x \equiv a+1$. If the point $(X, M(X))$ is deleted from pois, then NEXTPOI will replace it (via the union in $V = ...$).

$\therefore H_X = V$. ∎

Lemma 5. *current[p] in Algorithm 1 always lies on the convex hull over* $[0, S]$ *over all iterations.*

Proof: current[p] is initially set to pois[p].HEAD, which lies on all convex hulls for M_p. It is subsequently updated by next = NEXTPOI (current[p]). Thus for some i, current[p] $= h_i$ as in Lemma 4, and current[p] lies on the convex hull over $[0, S]$. Note that S is decreasing over iterations of the main loop, so by Lemma 2 it is on the hull for all iterations. ∎

Theorem 2. PEEKAHEAD *is a lookahead algorithm.*

Proof: By Lemma 5, we know that NEXTPOI is called always starting from a point on the convex hull over remaining capacity, $[0, S]$. By Lemma 1 and Lemma 4, NEXTPOI returns the maximal utility per unit move for the partition. By Lemma 2 we know that so long as $s \leq S$, this move remains on the convex hull and by Lemma 1 remains the maximal utility per unit move. If $s > S$ then PEEKAHEAD recomputes the best move.

Because heap is a max-heap, heap.POP is the maximal utility per unit move across partitions. Thus each move made

by PEEKAHEAD is a lookahead move. PEEKAHEAD does not terminate until $S \equiv 0$. Furthermore, PEEKAHEAD is guaranteed forward progress, because at most P moves can be skipped before a valid move is found and S is reduced.

Therefore for all inputs PEEKAHEAD produces a lookahead sequence. ∎

3) Asymptotic Run-time: ALLHULLS has asymptotic run-time of $O(S)$, using the standard argument for the three-coins algorithm. All vector operations take constant time. Each point is added at most once in the main loop to pois and hull, giving $O(S)$. Finally backtracking removes points from hull at most once (and otherwise checks only the tail in $O(1)$), so *over the lifetime of the algorithm* backtracking takes $O(S)$. The overall run-time is thus $O(S + S) = O(S)$.

Now consider the run-time of PEEKAHEAD. Line 22 alone is $O(P \cdot S)$ (P invocations of ALLHULLS). The run-time of the remainder is complicated by the possibility of skipping steps (line 40). We first compute the run-time assuming no steps are skipped. In this case, NEXTPOI iterates over POIs at most once during the lifetime of the algorithm in $O(P \cdot S)$ total. PUSH (line 36) takes $O(\log P)$, and the main loop invokes $O(S)$ iterations for $O(\log P \cdot S)$. Thus altogether the run-time is $O(P \cdot S + \log P \cdot S) = O(P \cdot S)$.

If steps are skipped, then NEXTPOI can take $O(S)$ per skip to find the next POI. Further, $O(P)$ steps can be skipped per decrement of S. So it may be possible to construct worst-case miss curves that run in $O(P \cdot S^2)$. Note, however, that the number of skips is bounded by the number of concave regions. Miss curves seen in practice have at most a few concave regions and very few skips. The common case run-time is therefore bounded by ALLHULLS at $O(P \cdot S)$.

Indeed, Table 4 shows that the main loop (lines 38-44) runs in sub-linear time and ALLHULLS dominates, taking over 99% of execution time for large problem sizes. Contrast this with the common-case $O(P \cdot S^2)$ performance of UCP.

C. NUCA-Aware Share Placement

Once the Jigsaw runtime sizes all shares, it places them over cache banks using a simple greedy heuristic. Each share starts with its allocation given by PEEKAHEAD, called the budget. The goal of the algorithm is for each share to exhaust its budget on banks as close to the source as possible. The source is the core or "center of mass" of cores that generate accesses to a share. The distance of banks from the source is precomputed for each partition and passed as the lists $D_1 ... D_P$. Each bank is given an inventory of space, and shares simply take turns making small "purchases" from banks until all budgets are exhausted, as Algorithm 2 shows. PEEKAHEAD dominates the run-time of the complete algorithm at $O(P \cdot S)$.

V. EXPERIMENTAL METHODOLOGY

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [38], an x86-64 simulator based on Pin [27], and model tiled CMPs with 16 and 64 cores and a 3-level cache hierarchy, as shown in Fig. 1. We use both simple in-order core models, and detailed OOO models validated against a real Westmere system [38]. The 64-core CMP, with parameters shown in Table 3, is organized in 64 tiles, connected with an 8×8 mesh network-on-chip (NoC), and has 4 memory controllers at the edges. The scaled-down 16-core CMP has 16 tiles, a 4×4 mesh, and a single memory controller. The 16-core CMP has a total LLC capacity of 16 MB (1 MB/tile), and the 64-core CMP has 32 MB (512 KB/tile). We use McPAT [24] to derive the area and energy numbers of chip components (cores, caches, NoC, and memory controller) at 22 nm, and Micron DDR3L datasheets [32] to compute main memory energy. With simple cores, the 16-core system is implementable in 102 mm^2 and has a typical power consumption of 10-20 W in our workloads, consistent with adjusted area and power of Atom-based systems [13].

Cores	64 cores, x86-64 ISA, in-order IPC=1 except on memory accesses / Westmere-like OOO, 2 GHz
L1 caches	32 KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	512 KB/1 MB per tile, 4-way 52-candidate zcache, 9 cycles, inclusive, LRU/R-NUCA/Vantage/Jigsaw, or idealized shared-private D-NUCA with 2× capacity (IdealSPD)
Coherence protocol	MESI protocol, 64 B lines, in-cache directory, no silent drops; sequential consistency
Global NoC	8×8 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 1-cycle links
Memory controllers	4 MCUs, 1 channel/MCU, 120 cycles zero-load latency, 12.8 GB/s per channel

Table 3. Configuration of the simulated 64-core CMP.

Cache implementations: Experiments use an unpartitioned, shared (static NUCA) cache with LRU replacement as the baseline. We compare Jigsaw with Vantage, a representative partitioned design, and R-NUCA, a representative shared-baseline D-NUCA design. Because private-baseline D-NUCA schemes modify the coherence protocol, they are hard to model. Instead, we model an idealized shared-private D-NUCA scheme, IdealSPD, with 2× the LLC capacity. In IdealSPD, each tile has a private L3 cache of the same size as the LLC bank (512 KB or 1 MB), a fully provisioned 5-cycle directory bank that tracks the L3s, and a 9-cycle exclusive L4 bank (512 KB or 1 MB). Accesses that miss in the private L3 are serviced by the proper directory bank (traversing the NoC). The L4 bank acts as a victim cache, and is accessed in parallel with the directory to minimize latency. This models D-NUCA schemes that partition the LLC between shared and private regions, but gives the full LLC capacity to both the private (L3) and shared (L4) regions. Herrero et al. [16] show that this idealized scheme *always* outperforms several state-of-the-art private-baseline D-NUCA schemes that include shared-private partitioning, selective replication, and adaptive spilling (DCC [15], ASR [2], and ECC [16]), often by significant margins (up to 30%).

Vantage and Jigsaw both use 512-line (4 KB) UMONs with 128 ways (Sec. III-C), and reconfigure every 50 M cycles. Jigsaw uses 4 UMONs per 1 MB L3 bank, a total storage overhead of 1.4%. Vantage uses utility-based cache partitioning (UCP) [34]. R-NUCA is configured as proposed [14] with

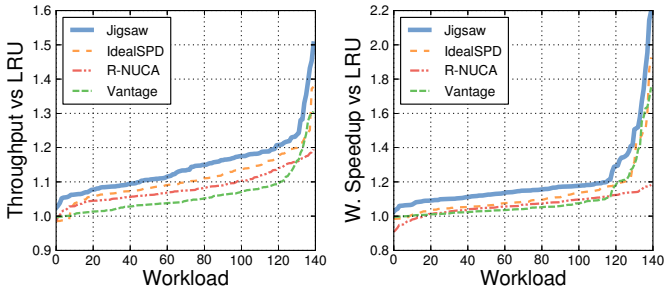


Figure 7. Throughput and weighted speedup of Jigsaw, Vantage, R-NUCA, and IdealSPD (with 2× cache) over LRU baseline, for 140 SPEC CPU2006 mixes on the 16-core chip with in-order cores.

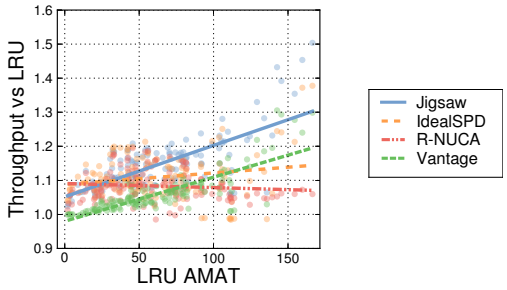


Figure 8. Performance with increasing memory intensity. Throughput is plotted against LRU's average memory access time for each mix on the 16-core chip. Lines indicate the best-fit linear regression.

4-way rotational interleaving and page-based reclassification. Jigsaw and R-NUCA use the page remapping support discussed in Sec. III, and Jigsaw implements bulk invalidations with per-bank pipelined scans of the tag array (with 1 MB 4-way banks, a scan requires 4096 tag array accesses). Jigsaw uses thread-private and per-process shares. In all configurations, banks use 4-way 52-candidate zcache arrays [36] with H_3 hash functions, though results are similar with more expensive 32-way set-associative hashed arrays.

Workloads and Metrics: We simulate mixes of single and multi-threaded workloads. For single-threaded mixes, we use a similar methodology to prior partitioning work [34, 37]. We classify all 29 SPEC CPU2006 workloads into four types according to their cache behavior: insensitive (`n`), cache-friendly (`f`), cache-fitting (`t`), and streaming (`s`) as in [37, Table 2], and build random mixes of all the 35 possible combinations of four workload types. We generate four mixes per possible combination, for a total of 140 mixes. We pin each application to a specific core, and fast-forward all applications for 20 billion instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, similar to FIESTA [17]: First, we run each application in isolation, and measure the number of instructions I_i that it executes in 1 billion cycles. Then, in each experiment we simulate the full mix until all applications have executed at least I_i instructions, and consider only the first I_i instructions of each application when reporting aggregate metrics. This ensures that each mix runs for at least 1 billion cycles. Our per-workload performance metric is $perf_i = IPC_i$.

For multi-threaded mixes, we use ten parallel benchmarks from PARSEC [5] (`blackscholes`, `canneal`, `fluidanimate`, `swaptions`), SPLASH-2 (`barnes`, `ocean`, `fft`, `lu`, `radix`), and BioParallel [19] (`svm`). We simulate 40 random mixes of four workloads. Each 16-thread workload is scheduled in one quadrant of the 64-core chip. Since IPC can be a misleading proxy for work in multithreaded workloads [1], we instrument each application with *heartbeats* that report global progress (e.g., when each timestep finishes in barnes). The ten applications we use are the ones from these suites for which we can add heartbeats without structural changes. For each application, we find the smallest number of heartbeats that complete in over 1 billion cycles from the start of the parallel region when running alone. This is the region of interest (ROI). We then run the mixes by fast-forwarding all workloads until the start of their parallel regions, running until all applications complete their ROI, and keep all applications running to avoid a lighter load on longer-running applications. To avoid biasing throughput by ROI length, our per-application performance metric is $perf_i = ROItime_{i,alone}/ROItime_i$.

We report throughput and fairness metrics: normalized throughput, $\sum_i perf_i / \sum_i perf_{i,base}$, and weighted speedup, $(\sum_i perf_i/perf_{i,base})/N_{apps}$, which accounts for fairness [34, 40]. To achieve statistically significant results, we introduce small amounts of non-determinism [1], and perform enough runs to achieve 95% confidence intervals $\leq$1% on all results.

VI. Evaluation

We first compare Jigsaw against alternative cache organizations and then present a focused analysis of Jigsaw.

A. Single-threaded mixes on 16-core CMP

We first present results with in-order cores, as they are easier to understand and analyze, then show OOO results.

Performance across all mixes: Fig. 7 summarizes both throughput and weighted speedup for the cache organizations we consider across the 140 mixes. Each line shows the performance improvement of a single organization against the shared LRU baseline. For each line, workload mixes (the x-axis) are sorted according to the improvement achieved. Lines are sorted independently, so these graphs give a concise summary of improvements, but should not be used for workload-by-workload comparisons among schemes.

Fig. 7 shows that Jigsaw is beneficial for all mixes, and achieves large throughput and fairness gains: up to 50% higher throughput, and up to 2.2× weighted speedup over an unpartitioned shared cache. Overall, Jigsaw achieves gmean throughput/weighted speedups of 14.3%/18.4%, Vantage achieves 5.8%/8.2%, R-NUCA achieves 8.2%/6.3%, and IdealSPD achieves 10.7%/11.4%. Partitioning schemes benefit weighted speedup more than throughput, improving fairness (despite using UCP, which optimizes throughput). R-NUCA favors throughput but not fairness, and IdealSPD favors both, but note this is an upper bound with twice the cache capacity.

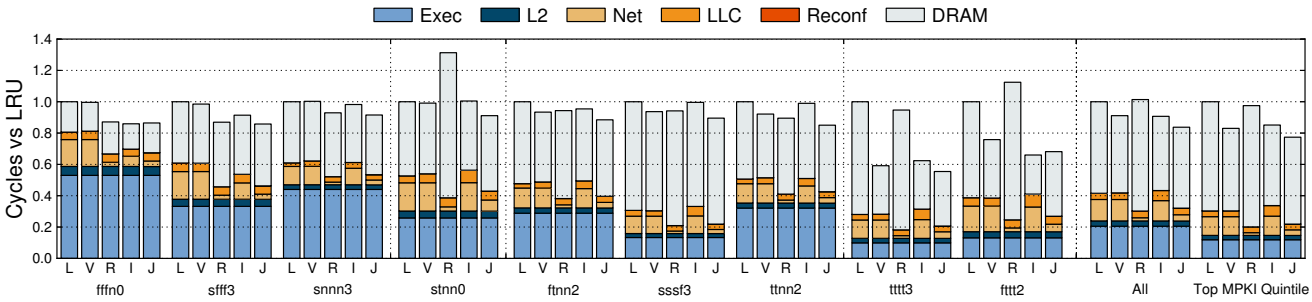


Figure 9. Execution time breakdown of LRU (L), Vantage (V), R-NUCA (R), IdealSPD (I), and Jigsaw (J), for representative 16-core single-thread mixes. Cycles are normalized to LRU's (*lower is better*).

Performance of memory-intensive mixes: These mixes have a wide range of behaviors, and many access memory infrequently. For the mixes with the highest 20% of memory intensities (aggregate LLC MPKIs in LRU), where the LLC organization can have a large impact, the achieved gmean throughputs/weighted speedups are 21.2%/29.2% for Jigsaw, 11.3%/19.7% for Vantage, 5.8%/4.8% for R-NUCA, and 8.6%/14% for IdealSPD. Jigsaw and Vantage are well above their average speedups, R-NUCA is well *below*, and IdealSPD is about the same. Most of these mixes are at the high end of the lines in Fig. 7 for Jigsaw and Vantage, but not for R-NUCA. R-NUCA suffers on memory-intensive mixes because its main focus is to *reduce LLC access latency, not MPKI*, and IdealSPD does not improve memory-intensive mixes because it provides *no capacity control in the shared region*.

Fig. 8 illustrates this observation, showing the performance of each mix versus memory intensity. Performance is measured by throughput normalized to LRU, and memory intensity is measured by LRU's average memory access time. The figure also includes the best-fit linear regression, which shows the performance-memory intensity correlation for each cache organization. Jigsaw and Vantage both perform increasingly well with increasing memory intensity, exhibiting similar, positive correlation. Vantage is the worst-performing scheme at low memory intensity, but under high intensity it is only bested by Jigsaw. In constract, R-NUCA's performance degrades with increasing memory intensity due to its limited capacity. Finally, IdealSPD shows only slightly increasing performance versus LRU despite having twice the cache capacity. In [16] IdealSPD was (predictably) shown to outperform other D-NUCA schemes by the largest amount on mixes with high memory intensity. Therefore, it's unclear if actual D-NUCA schemes would realize even the modest improvement with increasing memory intensity seen in Fig. 8. Jigsaw performs similarly to D-NUCA schemes at low memory intensity, and at high intensities is clearly the best organization.

Performance breakdown: To gain more insight into these differences, Fig. 9 shows a breakdown of execution time for nine representative mixes. Each bar shows the total number of cycles across all workloads in the mix for a specific configuration, normalized to LRU's (the inverse of each bar is throughput over LRU). Each bar further breaks down where cycles are spent, either executing instructions or stalled on a memory access. Memory accesses are split into their L2, NoC, LLC, and memory components. For R-NUCA and Jigsaw, we include time spent on reconfigurations and remappings, which is negligible. For IdealSPD, the LLC contribution includes time spent in private L3, directory, and shared L4 accesses.

We see four broad classes of behavior: First, in *capacity-insensitive mixes* (e.g., `fffn0`, `sfff3`, and `snnn3`) partitioning barely helps, either because applications have small working sets that fit in their local banks or have streaming behavior. Vantage thus performs much like LRU on these mixes. R-NUCA improves performance by keeping data in the closest bank (with single-threaded mixes, R-NUCA behaves like a private LLC organization without a globally shared directory). Jigsaw maps shares to their closest banks, achieving similar improvements. IdealSPD behaves like R-NUCA on low memory intensity mixes (e.g., `fffn0`) where the shared region is lightly used so all data lives in local banks. With increasing memory intensity (e.g. `sfff3` and `snnn3`) its directory overheads (network and LLC) begin to resemble LRU, so much so that its overall performance matches LRU at `snnn3`. For all remaining mixes in Fig. 9, IdealSPD has similar network and LLC overheads to LRU.

Second, *capacity-critical mixes* (e.g., `stnn0`) contain applications that do not fit within a single bank, but share the cache effectively without partitioning. Here, Vantage and IdealSPD show no advantage over LRU, but R-NUCA in particular performs poorly, yielding higher MPKI than the shared LRU baseline. Jigsaw gets the benefit of low latency, but without sacrificing the MPKI advantages of higher capacity.

Third, in *partitioning-friendly mixes* (e.g., `ftnn2`, `sssf3`, and `ttnn2`) each application gets different utility from the cache, but no single application dominates LLC capacity. Partitioning reduces MPKI slightly, whereas R-NUCA gets MPKI similar to the shared LRU baseline, but with lower network latency. IdealSPD performs somewhere between Vantage and LRU because it does not partition within the shared region. Jigsaw captures the benefits of both partitioning and low latency, achieving the best performance of any scheme.

`ftnn2` is typical, where Vantage gives modest but non-negligible gains, and IdealSPD matches this performance by allowing each application a private bank (capturing high-

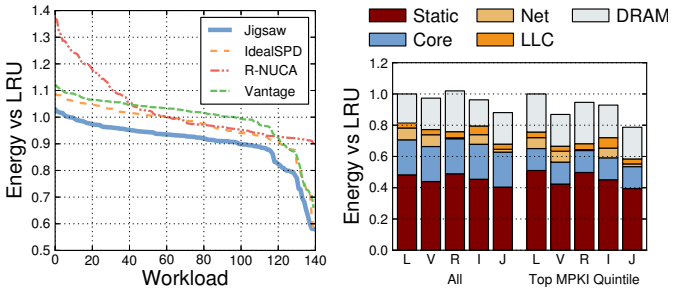


Figure 10. System energy across LLC organizations on 16-core, in-order chip: per-mix results, and average energy breakdown across mixes. Results are normalized to LRU's energy (*lower is better*).

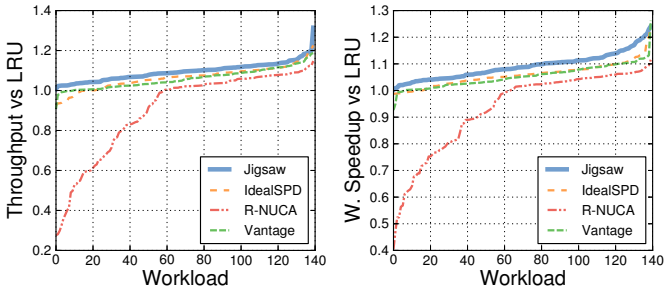


Figure 11. Throughput and weighted speedup of Jigsaw, Vantage, R-NUCA, and IdealSPD (2× cache) over LRU, for 140 SPEC CPU2006 mixes on 16-core chip with OOO cores and 4× memory bandwidth.

locality accesses) and dividing shared capacity among applications (giving additional capacity to high-intensity apps). R-NUCA gets very low latency, but at the cost of additional MPKI which makes it ultimately perform worse than Vantage. Jigsaw gets the benefits of both partitioning and low latency, and achieves the best performance. `sssf3` shows a similar pattern, but IdealSPD's directory overheads (specifically directory accesses, included in LLC access time) hurt its performance compared to Vantage.

`ttnn2` demonstrates the importance of capacity control. Vantage shows significant reductions in DRAM time, but IdealSPD resembles (almost identically) the shared LRU baseline. This performance loss is caused by the lack of capacity control within the shared region.

Fourth, *partitioning-critical mixes* (e.g., `tttt3` and `fttt2`) consist of cache-fitting apps that perform poorly below a certain capacity threshold, after which their MPKI drops sharply. In these mixes, a shared cache is ineffective at dividing capacity, and partitioning achieves large gains. R-NUCA limits apps to their local bank and performs poorly. Jigsaw is able to combine the advantages of partitioning with the low latency of smart placement, achieving the best performance.

In some mixes (e.g., `fttt2`), IdealSPD achieves a lower MPKI than Vantage and Jigsaw, but this is an artifact of having twice the capacity. Realistic shared-private D-NUCA schemes will always get less benefit from partitioning than Vantage or Jigsaw, as they partition between shared and private regions, but do not partition the shared region among applications. This effect is fairly common, indicating that our results may overestimate the top-end performance of private-baseline NUCA schemes.

Finally, Fig. 9 shows the average breakdown of cycles across all mixes and for the mixes with the top 20% of memory intensity. They follow the trends already discussed: Vantage reduces DRAM time but not network latency. R-NUCA reduces network latency significantly, but at the cost of additional DRAM time. IdealSPD performs similarly to Vantage; in particular, its global directory overheads are significant. Jigsaw is the only organization to achieve both low DRAM time and low network latency. The main difference on high memory intensity mixes is that Vantage and Jigsaw reduce DRAM time significantly more than other organizations.

Energy: Fig. 10 shows the system energy (full chip and main memory) consumed by each cache organization for each of the 140 mixes, normalized to LRU's energy. Lower numbers are better. Jigsaw achieves the largest energy reductions, up to 72%, 10.6% on average, and 22.5% for the mixes with the highest 20% of memory intensities. Fig. 10 also shows the per-component breakdown of energy consumption for the different cache organizations, showing the reasons for Jigsaw's savings: its higher performance reduces static (leakage and refresh) energy, and Jigsaw reduces both NoC and main memory dynamic energy. These results do not model UMON or STB energy overheads because they are negligible. Each UMON is 4 KB and is accessed infrequently (less than once every 512 accesses). STBs are less than 400 bytes, and each STB lookup reads only 12 bits of state.

Performance with OOO cores: Fig. 11 shows throughputs and weighted speedups for each organization when using Westmere-like OOO cores. We also quadruple the memory channels (51.2 GB/s) to maintain a balanced system given the faster cores. Jigsaw still provides the best gmean throughput/weighted speedup, achieving 9.9%/10.5% over the LRU baseline. Vantage achieves 3.2%/2.7%, R-NUCA achieves -1.4%/1.3%, and IdealSPD achieves 3.6%/2.2%. OOO cores tolerate memory stalls better, so improvements are smaller than with in-order cores. Additionally, OOO cores hide short latencies (e.g., LLC) better than long latencies (e.g., main memory), so reducing MPKI (Jigsaw/Vantage) becomes more important than reducing network latency (R-NUCA). Finally, R-NUCA underperforms LRU on 25% of the mixes, with up to 42% lower throughput. These are memory-intensive mixes, where R-NUCA's higher MPKIs drive main memory close to saturation, despite the much higher bandwidth. With infinite memory bandwidth, R-NUCA achieves 4.5%/7.1% average improvements with a worst-case throughput degradation of 15% vs LRU, while Jigsaw achieves 11.1%/11.8%.

B. Multi-threaded mixes on 64-core CMP

Fig. 12 shows throughput and weighted speedup results of different organizations on 40 random mixes of four 16-thread workloads in the 64-core CMP with in-order cores. We include two variants of Jigsaw: one with a single per-process share (Jigsaw (P)), and another with additional thread-private shares as discussed in Sec. IV (Jigsaw). Jigsaw achieves

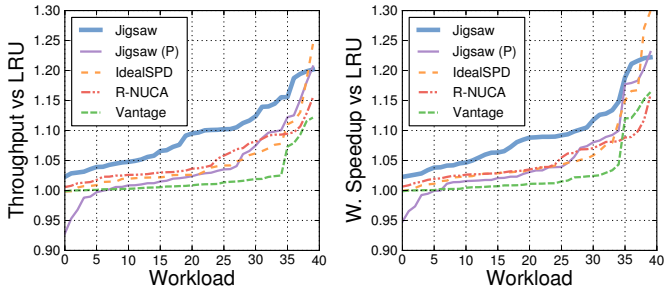
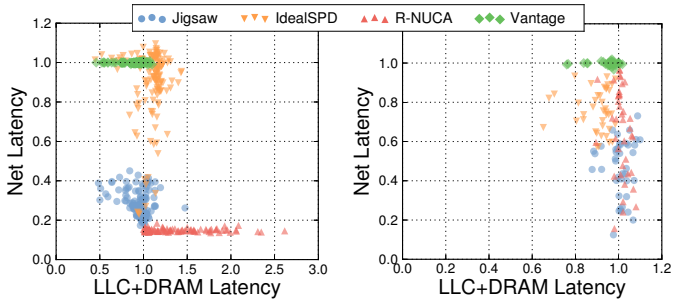


Figure 12. Throughput and weighted speedup of Jigsaw (w/ and w/o per-thread shares), Vantage, R-NUCA, and IdealSPD (2× cache) over LRU, for 40 4×16-thread mixes on 64-core chip with in-order cores.



(a) Single-threaded mixes, 16 cores **(b)** Multi-threaded mixes, 64 cores

Figure 13. Intrinsic MPKI and network latency reduction benefits of Jigsaw, Vantage, R-NUCA, and IdealSPD (with 2× cache) over LRU. Each point shows the average LLC + memory latency (x) and network latency (y) of one mix normalized to LRU's (*lower is better*).

the highest improvements of all schemes. Overall, gmean throughput/weighted speedup results are 9.1%/8.9% for Jigsaw, 1.9%/2.6% for Vantage, 5.0%/4.7% for R-NUCA, and 4.5%/5.5% for IdealSPD.

Unlike the single-threaded mixes, most applications are capacity-insensitive and have low memory intensity; only `canneal` is cache-friendly, and `ocean` is cache-fitting. This is why we model a 32 MB LLC: a 64 MB LLC improves throughput by only 3.5%. Longer network latencies emphasize smart placement, further de-emphasizing MPKI reduction. Consequently, Vantage yields small benefits except on the few mixes that contain `canneal` or `ocean`. IdealSPD enjoys the low latency of large local banks as well as a large shared cache, but read-write sharing is slower due to the deeper private hierarchy and global directory, ultimately yielding modest improvements. This drawback is characteristic of all private-based D-NUCA schemes. On the other hand, R-NUCA achieves low latency and, unlike the single-threaded mixes, does not suffer from limited capacity. This is both because of lower memory intensity and because R-NUCA uses shared cache capacity for data shared among multiple threads.

Jigsaw (P) does better than Vantage, but worse than Jigsaw due to the lack of per-thread shares. Jigsaw achieves lower network latency than R-NUCA and outperforms it further when partitioning is beneficial. Note that R-NUCA and Jigsaw reduce network latency by different means. R-NUCA places private data in the local bank, replicates instructions, and spreads shared data across all banks. Jigsaw just does placement: per-thread shares in the local bank, and per-process shares in the local quadrant of the chip. This reduces latency more than placing data throughout the chip and avoids capacity loss from replication. Because there is little capacity contention, we tried a modified R-NUCA that replicates read-only data (i.e., all pages follow a Private → Shared Read-only → Shared Read-write classification). This modified R-NUCA achieves 8.6%/8.5% improvements over LRU, bridging much of the gap with Jigsaw. While Jigsaw could implement fixed-degree replication a là R-NUCA, we defer implementing an adaptive replication scheme (e.g., using cost-benefit analysis and integrating it in the runtime) to future work.

Though not shown, results with OOO cores follow the same trends, with gmean throughput/weighted speedup improvements of 7.6%/5.7% for Jigsaw, 3.0%/3.7% for Vantage, 4.6%/2.1% for R-NUCA, and 4.4%/5.4% for IdealSPD.

C. Summary of results

Fig. 13 summarizes LLC performance for both 16- and 64-core mixes. For each cache organization, each mix is represented by a single point. Each point's x-coordinate is its LLC and main memory latency (excluding network) normalized to LRU, and the y-coordinate is its network latency normalized to LRU; lower is better in both dimensions. This representation tries to decouple each organization's intrinsic benefits in MPKI and latency reduction from the specific timing of the system. Overall, we draw the following conclusions:

- *Vantage is able to significantly reduce MPKI, but has no impact on network latency.* Vantage partitions within each bank, but does not trade capacity between banks to improve locality. In many mixes, network time far exceeds the savings in DRAM time, and a scheme that benefited locality could yield much greater improvements.
- *R-NUCA achieves low network latency, but at the cost of increased MPKI for a significant portion of mixes*. Often the losses in MPKI exceed the savings in network latency, so much so that R-NUCA has the worst-case degradation of all schemes.
- *IdealSPD is able to act as either a private-cache or shared-cache organization, but cannot realize their benefits simultaneously.* IdealSPD can match the main memory performance of Vantage on many mixes (albeit with twice the capacity) and R-NUCA's low latency on some mixes. However, IdealSPD struggles to do both due to its shared/private dichotomy, shown by its ⅂-shape in Fig. 13a. Mixes can achieve low latency only by avoiding the shared region. With high memory intensity, global directory overheads become significant, and it behaves as a shared cache.
- *Jigsaw combines the latency reduction of D-NUCA schemes with the miss reduction of partitioning*, achieving the best performance on a wide range of workloads.

D. Jigsaw analysis

Lookahead vs Peekahead: Table 4 shows the average core cycles required to perform a reconfiguration using both the UCP Lookahead algorithm and Peekahead as presented in Sec. IV. To run these experiments, we use the miss curves from

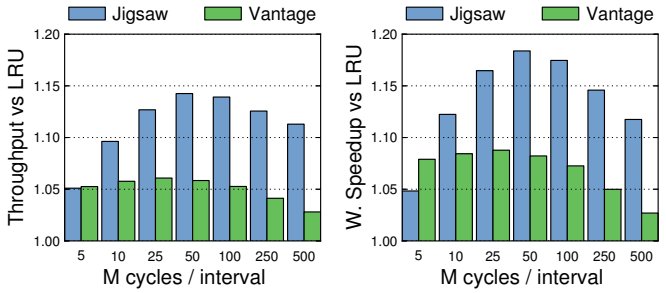


Figure 14. Mean improvements of Jigsaw and Vantage over LRU on the 140 16-core runs for a range of reconfiguration intervals.

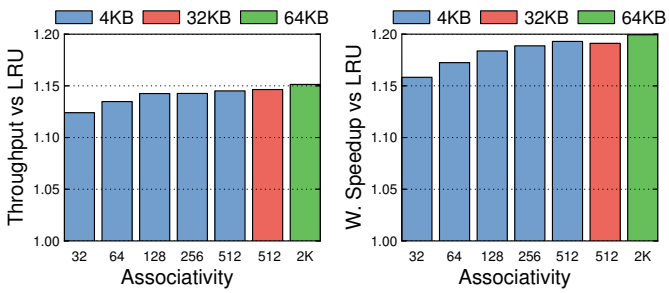


Figure 15. Mean improvements of Jigsaw over LRU on 140 single-threaded, 16-core runs for different UMON configurations.

Buckets	32	64	128	256	512	1024	2048	4096	8192
Lookahead	0.87	2.8	9.2	29	88	280	860	2,800	10,000
Peekahead	0.18	0.30	0.54	0.99	1.9	3.6	7.0	13	26
Speedup	4.8×	9.2×	17×	29×	48×	77×	125×	210×	380×
AllHulls %	87	90	92	95	96	98	99	99	99.5

Table 4. Performance of Peekahead and UCP's Lookahead [34]. Results given in M cycles per invocation.

the 140 16-core mixes at different resolutions. Conventional Lookahead scales near quadratically (3.2× per 2× buckets), while Peekahead scales sublinearly (1.9× per 2× buckets). AllHulls dominates Peekahead's run-time, confirming linear asymptotic growth. All previous results use 128 buckets (128-way UMONs), where Peekahead is 17× faster than Lookahead. Peekahead's advantage increases quickly with resolution. Overall, Jigsaw spends less than 0.1% of system cycles in reconfigurations at both 16 and 64 cores, imposing negligible overheads.

Sensitivity to reconfiguration interval: All results presented so far use a reconfiguration interval of 50 M cycles. Smaller intervals could potentially improve performance by adapting more quickly to phase changes in applications, but also incur higher reconfiguration overheads. Fig. 14 shows the gmean throughputs and weighted speedups achieved by both Jigsaw and Vantage on the 140 16-core mixes, for reconfiguration intervals of 5, 10, 25, 50, 100, 250, and 500 M cycles. Vantage is fairly insensitive to interval length, which is expected since its reconfigurations are fast and incur no invalidations, but also shows that for our target workloads there is little to gain from more frequent repartitionings. In contrast, Jigsaw benefits from longer intervals, as reconfigurations involve bulk invalidations. Performance quickly degrades below 10-25 M cycle intervals, and at 5 M cycles, the overheads from invalidations negate Jigsaw's benefits over Vantage. Both Jigsaw and Vantage degrade substantially with long intervals (250 and 500), but this may be an artifact of having few reconfigurations per run.

To elucidate this further, we also evaluated backing Jigsaw with a directory. We optimistically model an ideal, 0-cycle, fully-provisioned directory that causes no directory-induced invalidations. The directory enables migrations between lines in different banks after a reconfiguration, and avoids all bulk and page remapping invalidations. At 50 M cycles, directory-backed Jigsaw improves gmean throughput by 1.7%. We conclude that a directory-backed Jigsaw would not be beneficial. Even efficient implementations of this directory would require multiple megabytes and add significant latency, energy, and complexity. However, our workloads are fairly stable, we pin threads to cores, and do not overcommit the system. Other use cases (e.g., overcommitted systems) may change the tradeoffs.

Sensitivity to UMON *configuration:* Fig. 15 shows Jigsaw's performance over the 140 16-core mixes with different UMON configurations. These results show the impact of both associativity, which determines miss curve resolution, and UMON size, which determines sampling error. The blue bars show a sweep over associativity at 32, 64, 128, 256, and 512 ways for a 4 KB UMON (results use 128-way 4 KB UMONs). The red bar shows the impact of increasing UMON size 8× to 32 KB holding associativity at 512 ways; and the green bar is an idealized configuration with 2048-way, 64 KB UMONs shared among all bank partitions, eliminating sampling issues for multi-bank shares (Sec. III-C).

These results demonstrate a consistent performance improvement, in both throughput and weighted speedup, from 32 to 512 ways. Increasing associativity from 32 to 64 ways improves throughput/weighted speedup by 1.1%/1.4% over LRU. This benefit comes from being able to partition the cache at finer granularity. With low resolution, the runtime overallocates space to applications with sharp knees in their miss curves. This is because UMON data is missing around the knee in the curve, so the runtime cannot tell precisely where the knee occurs. Increasing UMON associativity improves resolution, and frees this space for other shares that make better use of it. Increasing to 128 ways improves performance by 0.8%/1.2%. Subsequent doublings of associativity improve performance by only 0.1%/0.4% over LRU on average. This indicates that while performance increases are steady, there are significantly diminishing returns. In contrast, increasing UMON size by 8× (red bar) improves throughput by just 0.1%. Clearly, sampling error is not a significant problem in Jigsaw. Finally, the ideal configuration (green bar) shows that while more performance is possible with ideal UMONs, the 128-way, 4 KB configuration comes within 0.8%/1.5%.

VII. Conclusions

We have presented Jigsaw, a cache organization that addresses the scalability and interference issues of distributed on-chip caches. Jigsaw lets software define shares, virtual caches

of guaranteed size and placement, and provides efficient mechanisms to monitor, reconfigure, and map data to shares. We have developed an efficient, novel software runtime that uses these mechanisms to achieve both the latency-reduction benefits of NUCA techniques and the hit-maximization benefits of controlled capacity management. As a result, Jigsaw significantly outperforms state-of-the-art NUCA and partitioning techniques over a wide range of workloads. Jigsaw can potentially be used for a variety of other purposes, including maximizing fairness, implementing process priorities or tiered quality of service, or exposing shares to user-level software to enable application-specific optimizations.

ACKNOWLEDGMENTS

We sincerely thank Deirdre Connolly, Christina Delimitrou, Srini Devadas, Frans Kaashoek, Harshad Kasture, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. This work was supported in part by DARPA PERFECT contract HR0011-13-2-0005 and Quanta Computer.

REFERENCES

[1] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.
[2] B. Beckmann, M. Marty, and D. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.
[3] B. Beckmann and D. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proc. MICRO-37*, 2004.
[4] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *Proc. PACT-22*, 2013.
[5] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT-17*, 2008.
[6] S. Bird and B. Smith, "PACORA: Performance aware convex optimization for resource allocation," in *Proc. HotPar-3*, 2011.
[7] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. STOC-9*, 1977.
[8] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proc. ISCA-33*, 2006.
[9] D. Chiou *et al.*, "Application-specific memory management for embedded systems using software-controlled caches," in *Proc. DAC-37*, 2000.
[10] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA-32*, 2005.
[11] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. MICRO-39*, 2006.
[12] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proc. HPCA-13*, 2007.
[13] G. Gerosa *et al.*, "A sub-1w to 2w low-power processor for mobile internet devices and ultra-mobile PCs in 45nm hi-k metal gate CMOS," in *ISSCC*, 2008.
[14] N. Hardavellas *et al.*, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proc. ISCA-36*, 2009.
[15] E. Herrero, J. González, and R. Canal, "Distributed Cooperative Caching," in *Proc. PACT-17*, 2008.
[16] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors," in *Proc. ISCA-37*, 2010.
[17] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," in *Proc. MoBS*, 2009.
[18] J. Jaehyuk Huh *et al.*, "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Par. Dist. Sys.*, vol. 18, no. 8, 2007.
[19] A. Jaleel, M. Mattina, and B. Jacob, "Last Level Cache (LLC) Performance of Data Mining Workloads On A CMP," in *HPCA-12*, 2006.
[20] C. Kim, D. Burger, and S. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS-10*, 2002.
[21] N. Kurd *et al.*, "Westmere: A family of 32nm IA processors," in *ISSCC*, 2010.
[22] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.
[23] B. Li *et al.*, "CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs," *J. Par. Dist. Comp.*, vol. 71, no. 5, 2011.
[24] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, 2009.
[25] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA-14*, 2008.
[26] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "NOC-Out: Microarchitecting a Scale-Out Processor," in *Proc. MICRO-45*, 2012.
[27] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.
[28] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-39*, 2012.
[29] M. Marty and M. Hill, "Virtual hierarchies to support server consolidation," in *Proc. ISCA-34*, 2007.
[30] A. A. Melkman, "On-line construction of the convex hull of a simple polyline," *Information Processing Letters*, vol. 25, no. 1, 1987.
[31] J. Merino, V. Puente, and J. Gregorio, "ESP-NUCA: A low-cost adaptive non-uniform cache architecture," in *Proc. HPCA-16*, 2010.
[32] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.
[33] M. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," in *Proc. HPCA-10*, 2009.
[34] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-39*, 2006.
[35] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. ISCA-27*, 2000.
[36] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proc. MICRO-43*, 2010.
[37] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proc. ISCA-38*, 2011.
[38] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proc. ISCA-40*, 2013.
[39] A. Seznec, "A case for two-way skewed-associative caches," in *ISCA-20*, 1993.
[40] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proc. ASPLOS-8*, 2000.
[41] S. Srikantaiah, M. Kandemir, and Q. Wang, "SHARP control: Controlled shared cache management in chip multiprocessors," in *MICRO-42*, 2009.
[42] D. Tam *et al.*, "Managing shared l2 caches on multicore systems in software," in *WIOSCA*, 2007.
[43] K. Varadarajan *et al.*, "Molecular caches: A caching structure for dynamic creation of app-specific heterogeneous cache regions," in *MICRO-39*, 2006.
[44] C. Wu and M. Martonosi, "A Comparison of Capacity Management Schemes for Shared CMP Caches," in *WDDD-7*, 2008.
[45] Y. Xie and G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.
[46] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA-32*, 2005.