Appears in the Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA), 2015

Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling

Nathan Beckmann, Po-An Tsai, Daniel Sanchez
Massachusetts Institute of Technology
{beckmann, poantsai, sanchez}@csail.mit.edu

***Abstract*—Cache hierarchies are increasingly non-uniform, so for systems to scale efficiently, data must be close to the threads that use it. Moreover, cache capacity is limited and contended among threads, introducing complex capacity/latency tradeoffs. Prior NUCA schemes have focused on managing data to reduce access latency, but have ignored thread placement; and applying prior NUMA thread placement schemes to NUCA is inefficient, as capacity, not bandwidth, is the main constraint.**

We present CDCS, a technique to jointly place threads and data in multicores with distributed shared caches. We develop novel monitoring hardware that enables fine-grained space allocation on large caches, and data movement support to allow frequent full-chip reconfigurations. On a 64-core system, CDCS outperforms an S-NUCA LLC by 46% on average (up to 76%) in weighted speedup and saves 36% of system energy. CDCS also outperforms state-of-the-art NUCA schemes under different thread scheduling policies.

***Index Terms*—cache, NUCA, thread scheduling, partitioning**

I. Introduction

The cache hierarchy is one of the main performance and efficiency bottlenecks in current chip multiprocessors (CMPs) [13, 21], and the trend towards many simpler and specialized cores further constrains the energy and latency of cache accesses [13]. Cache architectures are becoming increasingly non-uniform to address this problem (NUCA [34]), providing fast access to physically close banks, and slower access to far-away banks.

For systems to scale efficiently, data must be close to the computation that uses it. This requires keeping cached data in banks close to threads (to minimize on-chip traffic), while judiciously allocating cache capacity among threads (to minimize cache misses). Prior work has attacked this problem in two ways. On the one hand, dynamic and partitioned NUCA techniques [2, 3, 4, 8, 10, 11, 20, 28, 42, 51, 63] allocate cache space among threads, and then place data close to the threads that use it. However, these techniques ignore thread placement, which can have a large impact on access latency (Sec. II-B). On the other hand, thread placement techniques mainly focus on non-uniform memory architectures (NUMA) [7, 14, 29, 57, 59, 64] and use policies, such as clustering, that do not translate well to NUCA. In contrast to NUMA, where capacity is plentiful but bandwidth is scarce, *capacity contention is the main constraint for thread placement in NUCA* (Sec. II-B).

We find that to achieve good performance, the system must both manage cache capacity well *and* schedule threads to limit capacity contention. We call this *computation and data co-scheduling*. This is a complex, multi-dimensional optimization problem. We have developed CDCS, a scheme that performs computation and data co-scheduling effectively on modern CMPs. CDCS uses novel, efficient heuristics that achieve performance within 1% of impractical, idealized solutions. CDCS works on arbitrary mixes of single- and multi-threaded processes, and uses a combination of hardware and software techniques. Specifically, our contributions are:

- We develop a novel thread and data placement scheme that takes into account both data allocation and access intensity to jointly place threads and data across CMP tiles (Sec. IV).
- We design miss curve monitors that use geometric sampling to scale to very large NUCA caches efficiently (Sec. IV-G).
- We present novel hardware that enables incremental reconfigurations of NUCA caches, avoiding the bulk invalidations and long pauses that make reconfigurations expensive in prior NUCA techniques [4, 20, 41] (Sec. IV-H).

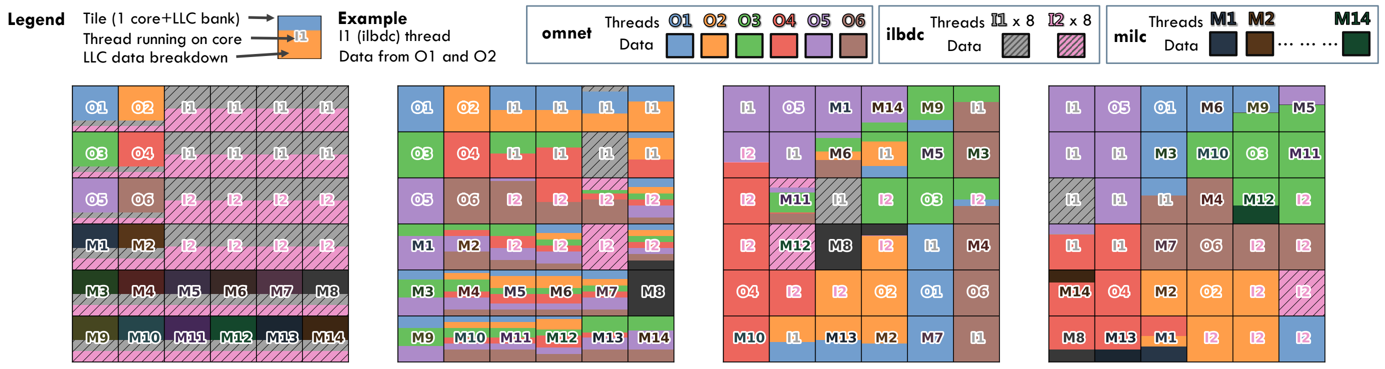
We prototype CDCS on Jigsaw [4], a partitioned NUCA baseline (Sec. III), and evaluate it on a 64-core system with lean OOO cores (Sec. VI). CDCS outperforms an S-NUCA cache by 46% gmean (up to 76%) and saves 36% of system energy. CDCS also outperforms R-NUCA [20] and Jigsaw [4] under different thread placement schemes. CDCS achieves even higher gains in under-committed systems, where not all cores are used (e.g., due to serial regions [25] or power caps [17]). CDCS needs simple hardware, works transparently to applications, and reconfigures the full chip every few milliseconds with minimal software overheads (0.2% of system cycles).

II. Background and Insights

We now discuss the prior work related to computation and data co-scheduling, focusing on the techniques that CDCS draws from. First, we discuss related work in multicore last-level caches (LLCs) to limit on- and off-chip traffic. Next, we present a case study that compares different NUCA schemes and shows that thread placement significantly affects performance. Finally, we review prior work on thread placement and show that NUCA presents an opportunity to improve thread placement beyond prior schemes.

A. Multicore caches

Non-uniform cache architectures: NUCA techniques [34] are concerned with data placement, but do not place threads or divide cache capacity among them. Static NUCA (S-NUCA) [34] spreads data across banks with a fixed line-bank mapping, and exposes a variable bank latency. Commercial CMPs often use



(a) R-NUCA (b) Jigsaw+Clustered (c) Jigsaw+Random (d) CDCS

Figure 1: Case study: 36-tile CMP with a mix of single- and multi-threaded workloads (omnet×6, milc×14, 8-thread ilbdc×2) under different NUCA organizations and thread placement schemes. Threads are labeled and data is colored by process.

S-NUCA [35]. Dynamic NUCA (D-NUCA) schemes adaptively place data close to the requesting core [2, 3, 8, 10, 11, 20, 28, 42, 51, 63] using a mix of placement, migration, and replication techniques. Placement and migration bring lines close to cores that use them, possibly introducing capacity contention between cores depending on thread placement. 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.

Most D-NUCA designs build on a *private-cache baseline*, where each NUCA bank is treated as a private cache. All banks are under a coherence protocol, which makes such schemes either hard to scale (in snoopy protocols) or require large directories that incur significant area, energy, latency, and complexity overheads (in directory-based protocols).

To avoid these costs, some D-NUCA schemes instead build on a *shared-cache baseline*: banks are not under a coherence protocol, and virtual memory is used to place data. Cho and Jin [11] use page coloring to map pages to banks. R-NUCA [20] specializes placement and replication policies for different data classes (instructions, private data, and shared data), outperforming prior D-NUCA schemes. Shared-baseline schemes are cheaper, as LLC data does not need coherence. However, remapping data is expensive as it requires page copies and invalidations.

Partitioned shared caches: Partitioning enables software to explicitly allocate cache space among threads or cores, but it is not concerned with data or thread placement. Partitioning can be beneficial because applications vary widely in how well they use the cache. Cache arrays can support multiple partitions with small modifications [9, 38, 40, 53, 56, 62]. Software can then set these sizes to maximize throughput [52], or to achieve fairness [44], isolation and prioritization [12, 18, 32], and security [47].

Unfortunately, partitioned caches scale poorly because they do not optimize placement. Moreover, they allocate capacity to cores, which works for single-threaded mixes, but incorrectly accounts for shared data in multi-threaded workloads.

Partitioned NUCA: Recent work has developed techniques to perform spatial partitioning of NUCA caches. These schemes

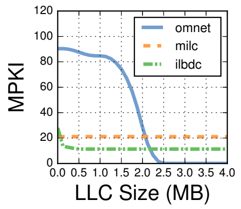


Figure 2: Application miss curves.

	omnet	ilbdc	milc	WSpdp
R-NUCA	1.09	0.99	1.15	1.08
Jigsaw+Cl	2.88	1.40	1.21	1.48
Jigsaw+Rnd	3.99	1.20	1.21	1.47
CDCS	4.00	1.40	1.20	1.56

Table 1: Per-app and weighted speedups for the mix studied.

jointly consider data allocation and placement, reaping the benefits of NUCA and partitioned caches. However, they do not consider thread placement. Virtual Hierarchies rely on a logical two-level directory to partition the cache [41], but they only allocate full banks, double directory overheads, and make misses slower. CloudCache [36] implements virtual private caches that can span multiple banks, but allocates capacity to cores, needs a directory, and uses broadcasts, making it hard to scale. Jigsaw [4] is a shared-baseline NUCA with partitionable banks and single-lookup accesses. Jigsaw lets software divide the distributed cache in finely-sized *virtual caches*, place them in different banks, and map pages to different virtual caches. Using utility monitors [52], an OS-level software runtime periodically gathers the miss curve of each virtual cache and co-optimizes data allocation and placement.

B. *Case study: Tradeoffs in thread and data placement*

To explore the effect of thread placement on different NUCA schemes, we simulate a 36-core CMP running a specific mix. The CMP is a scaled-down version of the 64-core chip in Fig. 3, with 6×6 tiles. Each tile has a simple 2-way OOO core and a 512 KB LLC bank. (See Sec. V for methodology details.)

We run a mix of single- and multi-threaded workloads. From single-threaded SPEC CPU2006, we run six instances of `omnet` (labeled O1-O6) and 14 instances of `milc` (M1-M14). From multi-threaded SPEC OMP2012, we run two instances of `ilbdc` (labeled I1 and I2) with eight threads each. We choose this mix because it illustrates the effects of thread and data placement—Sec. VI uses a comprehensive set of benchmarks.

Fig. 1 shows how thread and data are placed across the chip under different schemes. Each square represents a tile. The label on each tile denotes the thread scheduled in the tile's core (labeled by benchmark as discussed before). The colors on each tile show a breakdown of the data in the tile's bank. Each process uses the same color for its threads and data. For example, in Fig. 1b, the upper-leftmost tile has thread O1 (colored blue) and its data (also colored blue); data from O1 also occupies parts of the top row of banks (portions in blue).

Fig. 1a shows the thread and data breakdown under R-NUCA when applications are grouped by type (e.g., the six copies of omnet are in the top-left corner). R-NUCA maps thread-private data to each thread's local bank, resulting in very low latency. Banks also have some of the shared data from the multithreaded processes I1 and I2 (shown hatched), because R-NUCA spreads shared data across the chip. Finally, code pages are mapped to different banks using rotational interleaving, though this is not visible in this mix because apps have small code footprints. These policies excel at reducing LLC access latency to private data vs. an S-NUCA cache. This helps milc and omnet, as shown in Table 1. Overall, R-NUCA speeds up this mix by 8% over S-NUCA.

In R-NUCA, other thread placements would make little difference for this mix, as most capacity is used for either thread-private data, which is confined to the local bank, or shared data, which is spread out across the chip. But R-NUCA does not use capacity efficiently in this mix. Fig. 2 shows why, giving the miss curves of each app. Each miss curve shows the misses per kilo-instruction (MPKI) that each process incurs as a function of LLC space (in MB). omnet is very memory-intensive, and suffers 85 MPKI below 2.5 MB. However, over 2.5 MB, its data fits in the cache and misses turn into hits. ilbdc is less intensive and has a smaller footprint of 512 KB. Finally, milc gets no cache hits no matter how much capacity it is given—it is a streaming application. In R-NUCA, omnet and milc apps get less than 512 KB, which does not benefit them, and ilbdc apps use more capacity than they need.

Jigsaw uses capacity more efficiently, giving 2.5 MB to each instance of omnet, 512 KB to each ilbdc (8 threads), and near-zero capacity to each milc. Fig. 1b shows how Jigsaw tries to place data close to the threads that use it. By using partitioning, Jigsaw can share banks among multiple types of data without introducing capacity interference. However, the omnet threads in the corner heavily contend for capacity of neighboring banks, and their data is placed farther away than if they were spread out. Clearly, *when capacity is managed efficiently, thread placement has a large impact on capacity contention and achievable latency*. Nevertheless, because omnet's data now fits in the cache, its performance vastly improves, by 2.88× over S-NUCA (its AMAT improves from 15.2 to 3.7 cycles, and its IPC improves from 0.22 to 0.61). ilbdc is also faster, because its shared data is placed close by instead of across the chip; and because omnet does not consume memory bandwidth anymore, milc instances have more of it and speed up moderately (Table 1). Overall, Jigsaw speeds up this mix by 48% over S-NUCA.

Fig. 1c shows the effect of randomizing thread placement to spread capacity contention among the chip. omnet instances now have their data in neighboring banks (1.2 hops on average, instead of 3.2 hops in Fig. 1b) and enjoy a 3.99× speedup over S-NUCA. Unfortunately, ilbdc's threads are spread further, and its performance suffers relative to clustering threads (Table 1). This shows why one policy does not fit all: depending on capacity contention and sharing behavior, apps prefer different placements. Specializing policies for single- and multithreaded apps would only be a partial solution, since multithreaded apps with large per-thread footprints and little sharing also benefit from spreading.

Finally, Fig. 1d shows how CDCS handles this mix. CDCS spreads omnet instances across the chip, avoiding capacity contention, but clusters ilbdc instances across their shared data. CDCS achieves a 4× speedup for omnet and a 40% speedup for ilbdc. CDCS speeds up this mix by 56%.

In summary, this case study shows that partitioned NUCA schemes use capacity more effectively and improve performance, but they are sensitive to thread placement, as threads in neighboring tiles can aggressively contend for capacity. This presents an opportunity to perform smart thread placement, but fixed policies have clear shortcomings.

C. *Cache and NUMA-aware thread placement*

CRUISE [29] is perhaps the closest work to CDCS. CRUISE schedules single-threaded apps in CMPs with multiple fixed-size last-level caches, each shared by multiple cores and unpartitioned. CRUISE takes a classification-based approach, dividing apps in thrashing, fitting, friendly, and insensitive, and applies fixed scheduling policies to each class (spreading some classes among LLCs, using others as filler, etc.). CRUISE bin-packs apps into fixed-size caches, but partitioned NUCA schemes provide flexibly sized virtual caches that can span multiple banks. It is unclear how CRUISE's policies and classification would apply to NUCA. For example, CRUISE would classify omnet as thrashing if it was considering many small LLCs, and as insensitive if considering few larger LLCs. CRUISE improves on DI [64], which profiles miss rates and schedules apps across chips to balance intensive and non-intensive apps. Both schemes only consider single-threaded apps, and have to contend with the lack of partitioned LLCs.

Other thread placement schemes focus on NUMA systems. NUMA techniques have different goals and constraints than NUCA: the large size and low bandwidth of main memory limit reconfiguration frequency and emphasize bandwidth contention over capacity contention. Tam et al. [57] profile which threads have frequent sharing and place them in the same socket. DINO [7] clusters single-threaded processes to equalize memory intensity, places clusters in different sockets, and migrates pages along with threads. In on-chip NUMA (CMPs with multiple memory controllers), Tumanov et al. [59] and Das et al. [14] profile memory accesses and schedule intensive threads close to their memory controller. These NUMA schemes focus on equalizing memory bandwidth, whereas we find that proper cache allocations cause capacity contention to be the main constraint on thread placement.

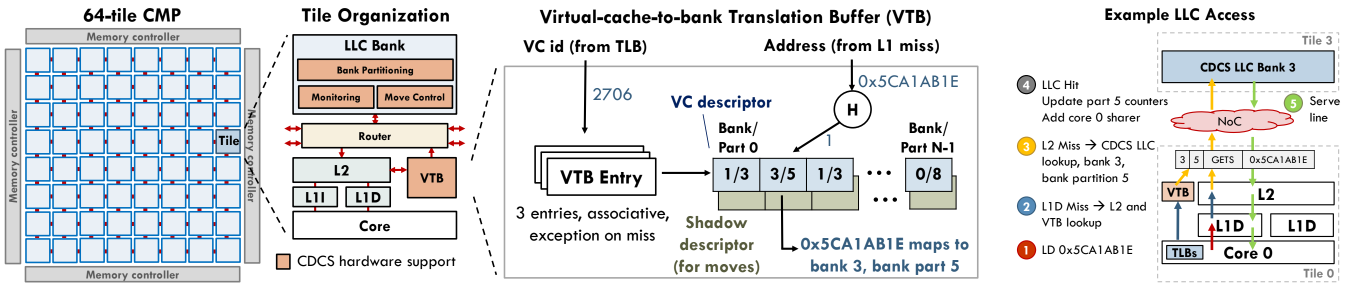


Figure 3: Target CMP (left), with tile configuration and microarchitectural additions introduced for CDCS. CDCS gangs portions of banks into virtual caches, and uses the VTB (center) to find the bank and bank partition to use on each access (right).

Finally, prior work has used high-quality *static mapping* techniques to spatially decompose regular parallel problems. Integer linear programming is useful in spatial architectures [45], and graph partitioning is commonly used in stream scheduling [49, 50]. While some of these techniques could be applied to the thread and data placement problem, they are too expensive to use dynamically, as we will see in Sec. VI-C.

III. CDCS Operation

Because Sec. II-B shows that judicious, fine-grained capacity allocation is highly beneficial, we adopt some of Jigsaw's mechanisms as a baseline for CDCS. We first explain how CDCS operates between reconfigurations (similar to Jigsaw), then how reconfigurations happen in Sec. IV (different from Jigsaw). For ease of understanding, we present CDCS in the concrete context of a partitioned NUCA scheme. In Sec. IV-I, we discuss how to generalize CDCS to other NUCA substrates.

Fig. 3 shows the tiled CMP architecture we consider, and the hardware additions of CDCS. Each tile has a core and a slice of the LLC. An on-chip network of arbitrary topology connects tiles, and memory controllers are at the edges. Pages are interleaved across memory controllers, as in Tilera and Knights Corner chips [6]. In other words, we do not consider on-chip NUMA-aware placement. We focus on NUCA over NUMA because many apps have working sets that fit on-chip, so LLC access latency dominates their performance. NUMA-aware placement is complementary and CDCS could be extended to also perform it (Sec. II-C), which we leave to future work.

Virtual caches: CDCS lets software divide each cache bank in multiple partitions, using Vantage [53] to efficiently partition banks at cache-line granularity. Collections of bank partitions are ganged and exposed to software as a single virtual cache (VC) (called a *share* in Jigsaw [4]). This allows software to define many VCs cheaply (several per thread), and to finely size and place them among banks.

Mapping data to VCs: Unlike other D-NUCAs, in CDCS lines do not migrate in response to accesses. Instead, between reconfigurations, each line can only reside in a single LLC bank. CDCS maps data to VCs using the virtual memory subsystem, similar to R-NUCA [20]. Each page table entry is tagged with a VC id. On an L2 miss, CDCS uses the line address and its VC id to determine the bank and bank partition that the line maps to.

The *virtual-cache translation buffer* (VTB), shown in Fig. 3, determines the bank and bank partition for each access. The VTB stores the configuration of all VCs that the running thread can access [4]. In our implementation, it is a 3-entry lookup table, as each thread only accesses 3 VCs (as explained below). Each VTB entry contains a VC descriptor, which consists of an array of N bank and bank partition ids (in our implementation, $N = 64$ buckets). As shown in Fig. 3, to find the bank and bank partition ids, the address is hashed, and the hash value (between 0 and $N-1$) selects the bucket. Hashing allows spreading accesses across the VC's bank partitions in proportion to their capacities, which makes them behave as a cache of their aggregate size. For example, if a VC consists of two bank partitions A of 1 MB and B of 3 MB, by setting array elements 0–15 in the VC descriptor to A and elements 16–63 to B, B receives 3× more accesses than A. In this case, A and B behave like a 4 MB VC [4, 5].

The VTB is small and efficient. For example, with $N = 64$ buckets, 64 LLC banks, and 64 partitions per bank, each VC descriptor takes 96 bytes (64 2×6-bit buckets, for bank and bank partition ids). Each of the 3 VTB entries has two descriptors (shadow descriptors, shown in Fig. 3, aid reconfigurations and are described in Sec. IV-H), making the VTB ~600 bytes. Since VTB lookups are cheap, they are performed in parallel with L2 accesses, so that L2 misses can be routed to the appropriate LLC bank immediately. Fig. 3 (right) shows how the VTB is used in LLC accesses.

Periodically (e.g., every 25 ms), CDCS software changes the configuration of some or all VCs, changing both their bank partitions and sizes. The OS recomputes all VC descriptors based on the new data placement, and cores coordinate via inter-processor interrupts to update the VTB entries simultaneously. Sec. IV-H details how CDCS hardware incrementally reconfigures the cache, moving or invalidating lines that have changed location to maintain coherence. The two-level translation of pages to VCs and VCs to bank partitions allows Jigsaw and CDCS to be more responsive and take more drastic reconfigurations than prior shared-baseline D-NUCAs: reconfigurations simply require changing the VC descriptors, and software need not copy pages or alter page table entries.

Types of VCs: CDCS's OS-level runtime creates one thread-private VC per thread, one per-process VC for each process, and a global VC. Data accessed by a single thread is mapped

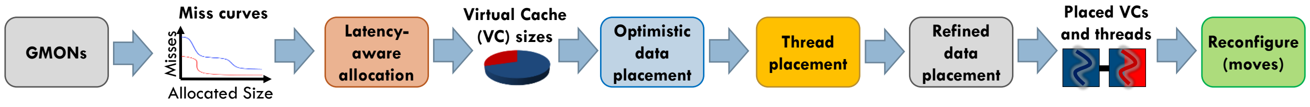


Figure 4: Overview of CDCS's periodic reconfiguration procedure.

to its thread-private VC, data accessed by multiple threads in the same process is mapped to the per-process VC, and data used by multiple processes is mapped to the global VC. Pages can be reclassified to a different VC efficiently [4] (e.g., when a page in a per-thread VC is accessed by another thread, it is remapped to the per-process VC), though in steady-state this happens rarely.

IV. CDCS Reconfigurations

CDCS reconfigurations use a combination of hardware and software techniques. Fig. 4 gives an overview of the steps involved. Novel, scalable geometric monitors (Sec. IV-G) sample the miss curves of each virtual cache. An OS runtime periodically reads these miss curves and uses them to jointly place VCs and threads using a 4-step procedure. Finally, this runtime uses hardware support to move cache lines to their new locations (Sec. IV-H). We first describe the software algorithm, then the monitoring and reconfiguration hardware. This hardware addresses overheads that would hinder CDCS performance, especially on large systems. However, these hardware techniques are useful beyond CDCS, e.g. to simplify coherence or reduce partitioning overheads.

All aspects of this process differ from Jigsaw [4]. Jigsaw uses a simple runtime that sizes VCs obliviously to their latency, places them greedily, and does not place threads. Jigsaw also uses conventional utility monitors [52] that do not scale to large caches, and reconfigurations require long pauses while banks invalidate data, which adds jitter.

A. A simple cost model for thread and data placement

As discussed in Sec. II-C, classification-based heuristics are hard to apply to NUCA. Instead, CDCS uses a simple analytical cost model that captures the effects of different placements on *total memory access latency*, and uses it to find a low-cost solution. This latency is better analyzed as the sum of on-chip (L2 to LLC) and off-chip (LLC to memory) latencies.

Off-chip latency: Assume a system with T threads and D VCs. Each thread t accesses VC d at a rate $a_{t,d}$ (e.g., 50 K accesses in 10 ms). If VC d is allocated s_d lines in the cache, its miss ratio is $M_d(s_d)$ (e.g., 10% of accesses miss). Then the total off-chip access latency is:

$$\text{Off-chip latency} = \sum_{t=1}^{T} \sum_{d=1}^{D} a_{t,d} \times M_d(s_d) \times \text{MemLatency} \quad (1)$$

where MemLatency is the latency of a memory access. This includes network latency, and relies on the average distance of all cores to memory controllers being the same (Fig. 3). Extending CDCS to NUMA would require modeling a variable main memory latency in Eq. 1.

On-chip latency: Each VC's capacity allocation s_d consists of portions of the N banks on chip, so that $s_d = \sum_{b=1}^{N} s_{d,b}$. The capacity of each bank B constrains allocations, so that $B = \sum_{d=1}^{D} s_{d,b}$. Limited bank capacities sometimes force data to be further away from the threads that access it, as we saw in Sec. II-B. Because the VTB spreads accesses across banks in proportion to capacity, the number of accesses from thread t to bank b is $\alpha_{t,b} = \sum_{d=1}^{D} \frac{s_{d,b}}{s_d} \times a_{t,d}$. If thread t is placed in a core c_t and the network distance between two tiles t_1 and t_2 is $D(t_1, t_2)$, then the on-chip latency is:

$$\text{On-chip latency} = \sum_{t=1}^{T} \sum_{b=1}^{N} \alpha_{t,b} \times D(c_t, b) \quad (2)$$

B. Overview of CDCS reconfiguration steps

With this cost model, the computation and data co-scheduling problem is to choose the c_t (thread placement) and $s_{t,b}$ (VC size and data placement) that minimize total latency, subject to the given constraints. However, finding the optimal solution is NP-hard [24, 48], and different factors are intertwined. For example, the size of VCs and the thread placement affect how close data can be placed to the threads that use it.

CDCS takes a multi-step approach to disentangle these interdependencies. CDCS first adopts optimistic assumptions about the contention introduced by thread and data placement, and gradually refines them to produce the final placement. Specifically, reconfigurations consist of four steps:

1) *Latency-aware allocation* divides capacity among VCs assuming data is compactly placed (no capacity contention).
2) *Optimistic contention-aware VC placement* places VCs among banks to avoid capacity contention. This step produces a rough picture of where data should be in the chip, e.g. placing `omnet` VCs far away enough in Fig. 1 to avoid the pathological contention in Fig. 1b.
3) *Thread placement* uses the optimistic VC placement to place threads close to the VCs they access. For example, in Fig. 1d, this step places `omnet` applications close to the center of mass of their data, and clusters `ilbdc` threads around their shared data.
4) *Refined VC placement* improves on the previous data placement to, now that thread locations are known, place data closer to minimize on-chip latency. For example, a thread that accesses its data intensely may swap allocations with a less intensive thread to bring its data closer; while this increases latency for the other thread, overall it is beneficial.

By considering data placement twice (steps 2 and 4), CDCS accounts for the circular relationship between thread and data placement. We were unable to obtain comparable results with a single VC placement; and CDCS performs almost as well as impractically expensive schemes, such as integer linear

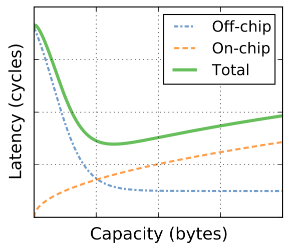


Figure 5: Access latency vs. capacity allocation.

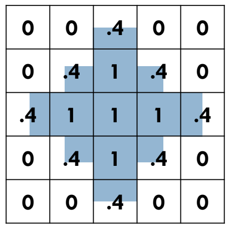
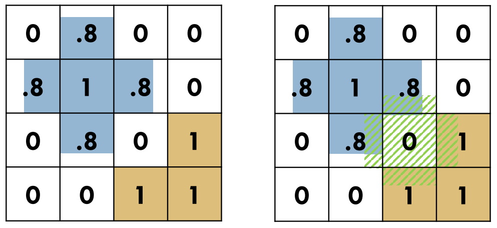


Figure 6: Optimistic uncontended virtual cache placement.



(a) Partial optimistic data placement (b) Estimating contention for VC (c) VC placed near least contended tile

Figure 7: Optimistic, contention-aware virtual cache placement.

programming (Sec. VI-C). CDCS uses arbitrary distance vectors, so it works with arbitrary topologies. However, to make the discussion concrete, we use a mesh topology in the examples.

C. Latency-aware capacity allocation

As we saw in Sec. II-B, VC sizes have a large impact on *both* off-chip latency and on-chip latency. Prior work has partitioned cache capacity to reduce cache misses [4, 52], i.e. off-chip latency. However, it is well-known that larger caches take longer to access [22, 23, 58]. Most prior partitioning work has targeted fixed-size LLCs with constant latency. But capacity allocation in NUCA caches provides an opportunity to also reduce on-chip latency: if an application sees little reduction in misses from a larger VC, the additional network latency to access it can negate the benefits of having fewer misses.

In summary, larger allocations have two competing effects: decreasing off-chip latency and increasing on-chip latency. This is illustrated in Fig. 5, which shows the average memory access latency to one VC (e.g., a thread's private data). Fig. 5 breaks latency into its off- and on-chip components, and shows that there is a "sweet spot" that minimizes total latency.

This has two important consequences. First, unlike in other D-NUCA schemes, it is sometimes better to leave cache capacity *unused*. Second, incorporating on-chip latency changes the curve's shape and also its marginal utility [52], leading to different cache allocations even when all capacity is used.

CDCS allocates capacity from *total memory latency curves* (the sum of Eq. 1 and Eq. 2) instead of miss curves. However, Eq. 2 requires knowing the thread and data placements, which are unknown at the first step of reconfiguration. CDCS instead uses an optimistic on-chip latency curve, found by compactly placing the VC around the center of the chip and computing the resulting average latency. For example, Fig. 6 shows the optimistic placement of an 8.2-bank VC accessed by a single thread, with an average distance of 1.27 hops.

With this simplification, CDCS uses the Peekahead optimization algorithm [4] to efficiently find the sizes of all VCs that minimize latency. While these allocations account for on-chip latency, they generally underestimate it due to capacity contention. Nevertheless, we find that this scheme works well because the next steps are effective at limiting contention.

D. Optimistic contention-aware VC placement

Once VC sizes are known, CDCS first finds a rough picture of how data should be placed around the chip to avoid placing large VCs close to each other. The main goal of this step is to *inform thread placement* by avoiding VC placements that produce high capacity contention, as in Fig. 1b.

To this end, we sort VCs by size and place the largest ones first. Intuitively, this works well because larger VCs can cause more contention, while small VCs can fit in a fraction of a bank and cause little contention. For each VC, the algorithm iterates over all the banks, and chooses the bank that yields the least contention with already-placed VCs as the center of mass of the current VC. To make this search efficient, we approximate contention by keeping a running tally of *claimed capacity* in each bank, and relax capacity constraints, allowing VCs to claim more capacity than is available at each bank. With N banks and D VCs, the algorithm runs in $\mathcal{O}(N \cdot D)$.

Fig. 7 shows an example of optimistic contention-aware VC placement at work. Fig. 7a shows claimed capacity after two VCs have been placed. Fig. 7b shows the contention for the next VC at the center of the mesh (hatched), where the uncontended placement is a cross. Contention is approximated as the claimed capacity in the banks covered by the hatched area—or 3.6 in this case. To place a single VC, we compute the contention around that tile. We then place the VC around the tile that had the lowest contention, updating the claimed capacity accordingly. For instance, Fig. 7c shows the final placement for the third VC in our example.

E. Thread placement

Given the previous data placement, CDCS tries to place threads closest to the center of mass of their accesses. Recall that each thread accesses multiple VCs, so this center of mass is computed by weighting the centers of mass of each VC by the thread's accesses to that VC. Placing the thread at this center of mass minimizes its on-chip latency (Eq. 2).

Unfortunately, threads sometimes have the same centers of mass. To break ties, CDCS places threads in descending *intensity-capacity product* (sum of VC accesses × VC size for each VC accessed). Intuitively, this order prioritizes threads for which low on-chip latency is important, and for which VCs are hard to move. For example, in the Sec. II-B case study, `omnet` accesses a large VC very intensively, so `omnet` instances are placed first. `ilbdc` accesses moderately-sized shared data at moderate intensity, so its threads are placed second, clustered around their shared VCs. Finally, `milc` instances access their private VCs intensely, but these VCs are tiny, so they are placed

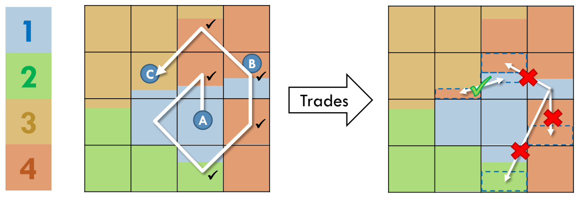


Figure 8: Trading data placement: Starting from a simple initial placement, VCs trade capacity to move their data closer. Only trades that reduce total latency are permitted.

last. This is fine because the next step, refined VC placement, can move small VCs to be close to their accessors very easily, with little effect on capacity contention.

For multithreaded workloads, this approach clusters shared-heavy threads around their shared VC, and spreads private-heavy threads to be close to their private VCs. Should threads access private and shared data with similar intensities, CDCS places threads relatively close to their shared VC but does not tightly cluster them, avoiding capacity contention among their private VCs.

F. Refined VC placement

Finally, CDCS performs a round of detailed VC placement to reduce the distance between threads and their data.

CDCS first simply round-robins VCs, placing capacity as close to threads as possible without violating capacity constraints. This greedy scheme, which was used in Jigsaw [4], is a reasonable starting point, but produces sub-optimal placements. For example, a thread's private VC always gets space in its local bank, regardless of the thread's memory intensity. Also, shared VCs can often be moved at little or no cost to make room for data that is more sensitive to placement. This is because moving shared data farther away from one accessing thread often moves it closer to another.

Furthermore, unlike in previous steps, it is straightforward to compute the effects of moving data, since we have an initial placement to compare against. CDCS therefore looks for beneficial trades between pairs of VCs after the initial, greedy placement. Specifically, CDCS computes the latency change from trading capacity between VC_1 at bank b_1 and VC_2 at bank b_2 using Eq. 2. The change in latency for VC_1 is:

$$\Delta\text{Latency} = \frac{\text{Accesses}}{\text{Capacity}} \times \left(D(\text{VC}_1, b_1) - D(\text{VC}_1, b_2)\right)$$

The first factor is VC_1's accesses per byte of allocated capacity. Multiplying by this factor accounts for the number of accesses that are affected by moving capacity, which varies between VCs. The equation for VC_2 is similar, and the net effect of the trade is their sum. If the net effect is negative (lower latency is better), then the VCs swap bank capacity.

Naïvely enumerating all possible trades is prohibitively expensive, however. Instead, CDCS performs a bounded search by iterating over all VCs: Each VC spirals outward from its center of mass, trying to move its data closer. At each bank b along the outward spiral, if the VC has not claimed all of b's capacity then it adds b to a list of desirable banks. These are the banks it will try to trade into later. Next, the VC tries to move its data placed in b (if any) closer by iterating over closer, desirable banks and offering trades with VCs that have data in these banks. If the trades are beneficial, they are performed. The spiral terminates when the VC has seen all of its data, since no farther banks will allow it to move any data closer.

Fig. 8 illustrates this for an example CMP with four VCs and some initial data placement. We now discuss how CDCS performs a bounded search for VC1. We spiral outward starting from VC1's center of mass at bank A, and terminate at VC1's farthest data at bank C. Desirable banks are marked with black checks on the left of Fig. 8. We only attempt a few trades, shown on the right side of Fig. 8. At bank B, VC1's data is two hops away, so we try to trade it to any closer, marked bank. For illustration, suppose none of the trades are beneficial, so the data does not move. This repeats at bank C, but suppose the first trade is now beneficial. VC1 and VC4 trade capacity, moving VC1's data one hop closer.

This approach gives every VC a chance to improve its placement. Since any beneficial trade must benefit one party, it would discover all beneficial trades. However, for efficiency, in CDCS each VC trades only once, since we have empirically found this discovers most trades. Finally, this scheme incurs negligible overheads, as we will see in Sec. VI-C.

These techniques are cheap and effective. We also experimented with more expensive approaches commonly used in placement problems: integer linear programming, simulated annealing, and graph partitioning. Sec. VI-C shows that they yield minor gains and are too expensive to be used online. We now discuss the hardware extensions necessary to efficiently implement CDCS on large CMPs.

G. Monitoring large caches

Monitoring miss curves in large CMPs is challenging. To allocate capacity efficiently, we should manage it in small chunks (e.g., the size of the L1s) so that it isn't over-allocated where it produces little benefit. This is crucial for VCs with small working sets, which see large gains from a small size and no benefit beyond. Yet, we also need miss curves that cover the full LLC because a few VCs may benefit from taking most capacity. These two requirements—fine granularity and large coverage—are problematic for existing monitors.

Conventional cache partitioning techniques use utility monitors (UMONs) [52] to monitor a fraction of sets, counting hits at each way to gather miss curves. UMONs monitor a fixed cache capacity per way, and would require a prohibitively large associativity to achieve both fine detail and large coverage. Specifically, in an UMON with W ways, each way models $1/W$ of LLC capacity. With a 32 MB LLC (Sec. V, Table 2) if we want to allocate capacity in 64 KB chunks, a conventional UMON needs 512 ways to have enough resolution. This is expensive to implement, even for infrequently used monitors.

Instead, we develop a novel monitor, called a *geometric monitor* (GMON). GMONs need fewer ways—64 in our evaluation—to model capacities from 64 KB up to 32 MB. This is possible because GMONs vary the sampling rate across ways, giving both fine detail for small allocations and large coverage, while

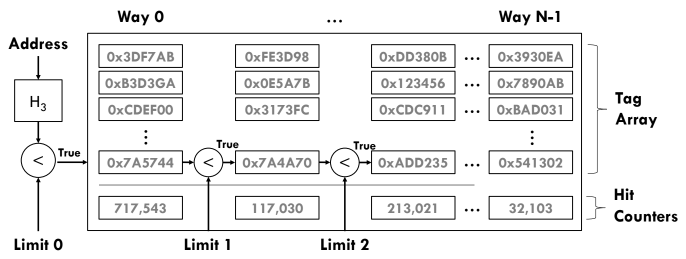


Figure 9: GMONs enhance UMONs with varying sampling rate across ways, controlled with per-way *limit registers*.

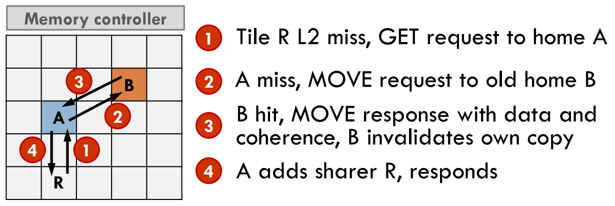
using many fewer ways than conventional UMONs. Fig. 9 shows this design: A GMON consists of small set-associative, tag-only array. Instead of storing address tags, GMON tags store 16-bit hashed addresses. GMONs also have a *limit register* per way. The limit registers progressively decrease across ways, and are used to filter out a fraction of lines per way as follows. In a conventional UMON, when an address is inserted or moved up to the first way, all other tags are moved to the next way. (This requires potentially shifting as many tags as ways, but only a small fraction of accesses are monitored, so the energy impact is small.) In a GMON, on each move, the hash value of the tag is checked against the way's limit register. If the value exceeds the limit register, the tag is discarded instead of moved to the next way, and the process terminates. Discarding lines achieves a variable, decreasing sampling rate per way, so GMONs model an increasing capacity per way [4, 5, 33].

We set limit registers to decrease the sampling rate by a factor $\gamma < 1$, so the sampling rate at way w is $k_w = \gamma^w$ less than at way zero, and then choose γ to cover the full cache capacity. For example, with a 32 MB LLC, a 64-way GMON with $\gamma \approx 0.95$ covers the full cache while having the first way model 64 KB. Modeled capacity per way grows by 26×, from 0.125 to 3.3 banks. This means GMONs miss curves are sparse, with high resolution at small sizes, and reduced resolution at large sizes. We find these GMONs work as well as the impractical UMONs described above (Sec. VI-C).

H. Incremental reconfigurations

Moving threads and data reduces steady-state network latency, but requires more drastic reconfigurations. Jigsaw reconfigures through bulk invalidations: all bank controllers walk the tag array and invalidate lines that should be mapped somewhere else, which requires pausing cores for tens to hundreds of thousands of cycles [4]. This is simple, but pauses, extra writebacks, and misses hurt performance. We observe that with few hardware modifications, we can spatially reconfigure the cache incrementally, without pausing cores, by moving lines instead of invalidating them. To our knowledge, this is the first scheme to achieve on-chip data migration through moves without requiring a coherence directory.

The key idea is to, upon a reconfiguration, temporarily treat the cache as a two-level hierarchy. We add a *shadow VC descriptor* to each VTB entry, as shown in Fig. 3. Upon reconfiguration, each core copies the VC descriptors into the shadow descriptors, and updates the normal VC descriptors with


Memory controller
A
B
R
1 Tile R L2 miss, GET request to home A
2 A miss, MOVE request to old home B
3 B hit, MOVE response with data and coherence, B invalidates own copy
4 A adds sharer R, responds
(a) Demand move, old bank hit.

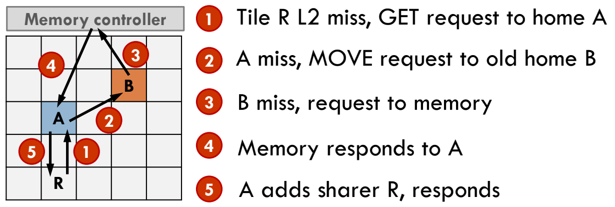

Memory controller
A
B
R
1 Tile R L2 miss, GET request to home A
2 A miss, MOVE request to old home B
3 B miss, request to memory
4 Memory responds to A
5 A adds sharer R, responds
(b) Demand move, old bank miss.

Figure 10: Messages and protocol used on incremental reconfigurations: demand moves when old bank hits or misses.

the new configuration. When the shadow descriptors are active, the VTB finds both the current and previous banks and bank partitions for the line, and, if they are different, sends the old bank id along with its request. Fig. 10 illustrates this protocol. If the current bank misses, it forwards the request to the old bank instead of memory. If the old bank hits, it sends both the line and its coherence state to the new bank, invalidating its own copy. This moves the line to its new location. We call this a *demand move* (Fig. 10a). If the old bank misses, it forwards the request to the memory controller (Fig. 10b). Demand moves have no races because all requests follow the same path through both virtual levels, i.e. they access the same sequence of cache banks. If a second request for the same line arrives at the new bank, it is queued at the MSHR that's currently being used to serve the first request.

Demand moves quickly migrate frequently used lines to their new locations, but the old banks must still be checked until all lines have moved. This adds latency to cache accesses. To limit this cost, banks walk the array in the background and incrementally invalidate lines whose location has changed. Unlike bulk invalidations, these *background invalidations* are not on the critical path and can proceed at a comparatively slow rate. For example, by scanning one set every 200 cycles, background invalidations finish in 100 Kcycles. Background invalidations begin after a short period (e.g., 50 Kcycles) to allow frequently-accessed lines to migrate via demand moves. After banks walk the entire array, cores stop using the shadow VTB descriptors and resume normal operation.

In addition to background invalidations, we also experimented with background moves, i.e. having banks send lines to their new locations instead of invalidating them. However, we found that background moves and background invalidations performed similarly—most of the benefit comes from not pausing cores as is done in bulk invalidations. We prefer background invalidations because they are simpler: background moves require additional state at every bank (viz., *where* the line needs to be moved, not just that its location has changed), and require a more sophisticated protocol (as there can be races between demand and background moves).

Overall, by taking invalidations off the critical path, CDCS can reconfigure the cache frequently without pausing cores or invalidating frequently-accessed data. As Sec. VI-C shows, background invalidations narrow the performance gap with an idealized architecture that moves lines immediately to their destination banks, and are faster than using bulk invalidations.

I. Putting it all together

Hardware overheads: Implementing CDCS as described imposes small overheads that the achieved system-wide performance and energy savings compensate for:

- Each bank is partitioned. With 512 KB banks and 64-byte lines, Vantage adds 8 KB of state per bank to support 64 bank partitions [4] (each tag needs a 6-bit partition id and each bank needs 256 bits of per-partition state).
- Each tile's VTB is 588 bytes: 576 bytes for 6 VC descriptors (3 normal + 3 shadow) and 12 bytes for the 3 tags.
- CDCS uses 4 monitors per bank. Each GMON has 1024 tags and 64 ways. Each tag is a 16-bit hash value (we do not store full addresses, since rare false positives are fine for monitoring purposes). Each way has a 16-bit limit register. This yields 2.1 KB monitors, and 8.4 KB overhead per tile.

This requires 17.1 KB of state per tile (2.9% of the space devoted to the tile's bank) and simple logic. Overheads are similar to prior partitioning-based schemes [4, 52].

Unlike Jigsaw [4], CDCS places each VC's monitor in a fixed location on the chip to avoid clearing or migrating monitor state. Since cores already hash lines to index the VTB, we store the GMON location at the VTB. For full LLC coverage with $\gamma = 0.95$ and 64 cores, we sample every 64th access. Monitoring is off the critical path, so this has negligible impact on performance (traffic is roughly $1/64^{\text{th}}$ of an S-NUCA cache).

Software overheads: Periodically (every 25 ms in our implementation), a software runtime wakes up on core 0 and performs the steps in Fig. 4. Reconfiguration steps are at most quadratic on the number of tiles (Sec. IV-D), and most are simpler. Software overheads are small, 0.2% of system cycles, and are detailed in Sec. VI-C and Table 3.

CDCS on other NUCA schemes: If partitioned banks are not desirable, CDCS can be used as-is with non-partitioned NUCA schemes [11, 28, 41]. To allow more VCs than threads, we could use several smaller banks per tile (e.g., 4×128 KB), and size and place VCs at bank granularity. This would eliminate partitioning overheads, but would make VTBs larger and force coarser allocations. We evaluate the effects of such a configuration in Sec. VI-C. CDCS could also be used in spilling D-NUCAs [51], though the cost model (Sec. IV-A) would need to change to account for the multiple cache and directory lookups.

V. Experimental Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [54], and model a 64-tile CMP connected by an 8×8 mesh NoC with 8 memory controllers at the edges. Each tile has one lean 2-way OOO core similar to Silvermont [30] and a 3-level cache hierarchy, as shown in Fig. 3, with parameters shown in Table 2. This system is similar to Knights Landing [27]. We use McPAT 1.1 [37] to derive the area and energy numbers of chip components (cores, caches, NoC, and memory controller) at 22 nm, and Micron DDR3L datasheets [43] for main memory. This system is implementable in 408 mm^2 with typical power consumption of 80-130 W in our workloads, consistent with area and power of scaled Silvermont-based systems [27, 30].

Cores	64 cores, x86-64 ISA, 2 GHz, Silvermont-like OOO [30]: 8B-wide ifetch; 2-level bpred with 512×10-bit BHSRs + 1024×2-bit PHT, 2-way decode/issue/rename/commit, 32-entry IQ and ROB, 10-entry LQ, 16-entry SQ
L1 caches	32 KB, 8-way set-associative, split D/I, 3-cycle latency
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	512 KB per tile, 16-way set-associative, inclusive, 9 cycles, S-NUCA/R-NUCA/Jigsaw/CDCS
Coherence protocol	MESI, 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	8 MCUs, 1 channel/MCU, 120 cycles zero-load latency, 12.8 GB/s per channel

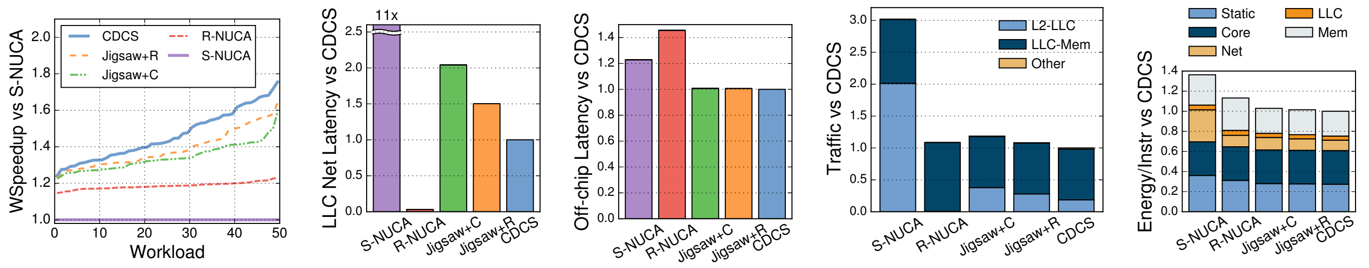
Table 2: Configuration of the simulated 64-core CMP.

Schemes: We compare CDCS with Jigsaw, R-NUCA, and S-NUCA organizations. R-NUCA and Jigsaw are implemented as proposed. R-NUCA uses 4-way rotational interleaving and page-based reclassification [20]. CDCS and Jigsaw use 64-way, 1 Kline GMONs from Sec. IV-G, and reconfigure every 25 ms.

Workloads: We simulate mixes of single and multithreaded workloads, with a methodology similar to prior work [4, 52, 53]. We simulate single-threaded mixes of SPEC CPU2006 apps. We use the 16 SPEC CPU2006 apps with ≥5 L2 MPKI: `bzip2`, `gcc`, `bwaves`, `mcf`, `milc`, `zeusmp`, `cactusADM`, `leslie3d`, `calculix`, `GemsFDTD`, `libquantum`, `lbm`, `astar`, `omnet`, `sphinx3`, and `xalancbmk`. We simulate mixes of 1–64 random apps. We fast-forward all apps for 20 billion instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, as in FIESTA [26]: We simulate each app alone, and measure how many instructions it executes in 1 billion cycles, I_i. Each experiment then runs the full mix until all apps execute at least I_i instructions, and consider only the first I_i instructions of each app to report performance. This ensures that each mix runs for at least 1 billion cycles.

We simulate multithreaded mixes of SPEC OMP2012 workloads. Since IPC is not a valid measure of work in multithreaded workloads [1], we instrument each app with heartbeats that report global progress (e.g., when each timestep or transaction finishes). For each app, 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 apps to the start of their parallel regions, and running the full mix until all apps complete their ROI.

We report weighted speedup over the S-NUCA baseline, which accounts for throughput and fairness [52, 55]. To achieve statistically significant results, we introduce small amounts of



(a) Distribution of weighted speedups over S-NUCA.
(b) Avg. on-chip latency on LLC accesses.
(c) Avg. off-chip latency (due to LLC misses).
(d) Breakdown of avg. on-chip network traffic per instruction.
(e) Breakdown of avg. energy per instruction.

Figure 11: Evaluation of S-NUCA, R-NUCA, Jigsaw, and CDCS across 50 mixes of 64 SPEC CPU2006 apps on a 64-core CMP.

non-determinism as in [1], and perform enough runs to achieve 95% confidence intervals $\leq$1%.

VI. Evaluation

A. *Single-threaded mixes*

Fig. 11a shows the distribution of weighted speedups that S-NUCA, R-NUCA, Jigsaw, and CDCS achieve in 50 mixes of 64 randomly-chosen, memory-intensive SPEC CPU2006 applications. We find that S-NUCA and R-NUCA are insensitive to thread placement (performance changes by $\leq$ 1%): S-NUCA because it spreads accesses among banks, and R-NUCA because its policies cause little contention, as explained in Sec. II-B. Therefore, we report results for both with a *random* scheduler, where threads are placed randomly at initialization, and stay pinned. We report Jigsaw results with two schedulers: *random* (Jigsaw+R) and *clustered* (Jigsaw+C), as in Sec. II-B. As we will see, neither choice is better in general—different mixes prefer one over the other. Each line shows the weighted speedup of a single scheme over the S-NUCA baseline, sorted along workload mixes (x-axis) by improvement (inverse CDF).

Fig. 11a shows that CDCS significantly improves system performance, achieving 46% gmean weighted speedup and up to 76%. Jigsaw+R achieves 38% gmean weighted speedup and up to 64%, Jigsaw+C achieves 34% gmean weighted speedup and up to 59%, and R-NUCA achieves 18% gmean weighted speedup and up to 23%. Jigsaw+C shows near-pathological behavior, as different instances of the same benchmark are placed close by, introducing capacity contention and hurting latency when they get large VCs (as in Fig. 1b). Jigsaw+R avoids this behavior and performs better, but CDCS avoids capacity contention much more effectively and attains higher speedups across all mixes. CDCS and Jigsaw widely outperform R-NUCA, as R-NUCA does not manage capacity efficiently in heterogeneous workload mixes (Sec. II-B).

Fig. 11 gives more insight into these differences. Fig. 11b shows the average network latency incurred by LLC accesses across all mixes (Eq. 2), normalized to CDCS, while Fig. 11c compares off-chip latency (Eq. 1). S-NUCA incurs 11$\times$ more on-chip network latency than CDCS on L2-LLC accesses, and 23% more off-chip latency. R-NUCA classifies most pages as private and maps them to the nearest bank, so its network latency for LLC accesses is negligible. However, the lack of capacity management degrades off-chip latency by 46% over CDCS. Jigsaw+C, Jigsaw+R and CDCS achieve similar off-chip latency, but Jigsaw+C and Jigsaw+R have 2$\times$ and 51% higher on-chip network latency for LLC accesses than CDCS.

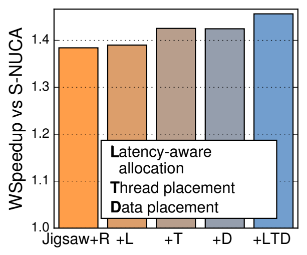
Fig. 11d compares the network traffic of different schemes, measured in flits, and split in L2-to-LLC and LLC-to-memory traffic. S-NUCA incurs 3$\times$ more traffic than CDCS, most of it due to LLC accesses. For other schemes, traffic due to LLC misses dominates, because requests are interleaved across memory controllers and take several hops (Sec. III). We could combine these schemes with NUMA-aware techniques [14, 15, 39, 57, 59, 60] to further reduce this traffic. Though not explicitly optimizing for it, CDCS achieves the lowest traffic.

Because CDCS improves performance and reduces network and memory traffic, it reduces energy as well. Fig. 11e shows the average energy per instruction of different organizations. Static energy (including chip and DRAM) decreases with higher performance, as each instruction takes fewer cycles. S-NUCA spends significant energy on network traversals, but other schemes make it a minor overhead; and R-NUCA is penalized by its more frequent memory accesses. Overall, Jigsaw+C, Jigsaw+R and CDCS reduce energy by 33%, 34% and 36% over S-NUCA, respectively.

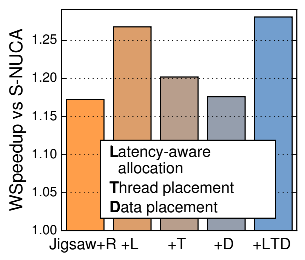
CDCS benefits apps with large cache-fitting footprints, such as `omnet`, `xalanc`, and `sphinx3`, the most. They require multi-bank VCs to work well, and benefit from lower access latencies. Apps with smaller footprints benefit from the lower contention, but their speedups are moderate.

Fig. 12a shows how each of the proposed techniques in CDCS improves performance when applied to Jigsaw+R individually. We show results for latency-aware allocation (+L), thread placement (+T), and refined data placement (+D); +LTD is CDCS. Since cache capacity is scarce, latency-aware allocation helps little, whereas thread and data placement achieve significant, compounding benefits.

Under-committed systems: Fig. 13 shows the weighted speedups achieved by S-NUCA, R-NUCA, Jigsaw+R, Jigsaw+C, and CDCS when the 64-core CMP is under-committed: each set of bars shows the gmean weighted speedup when running 50 mixes with an increasing number of single-threaded applications per mix, from 1 to 64. Besides characterizing these schemes on CMPs running at low utilization (e.g., due to limited power or parallelism), this scenario is similar to introducing a varying number of non-intensive benchmarks, for which LLC



(a) 64 apps



(b) 4 apps

Figure 12: Factor analysis of CDCS with 50 mixes of 64 and 4 SPEC CPU2006 applications on a 64-core CMP.

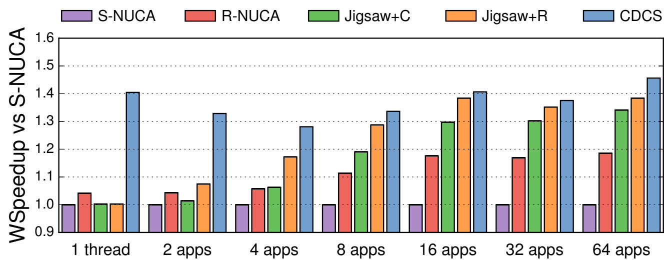


Figure 13: Weighted speedups for 50 mixes of 1, 2, 4, 6, 8, 16, 32, and 64 SPEC CPU2006 applications on a 64-core CMP.

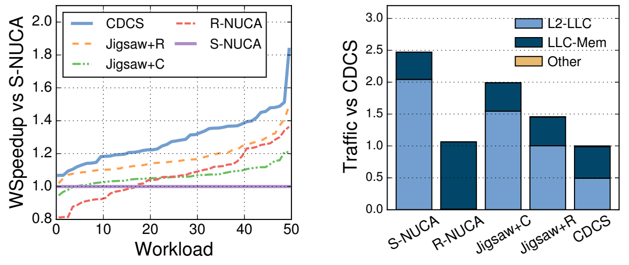
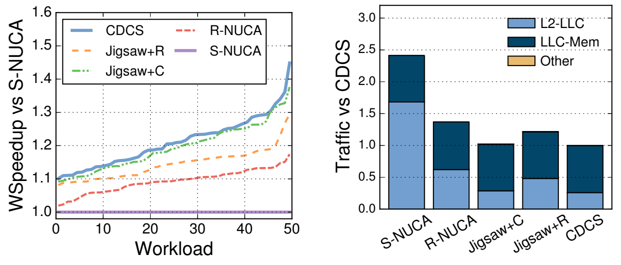


Figure 14: Weighted speedup distribution and traffic breakdown of 50 mixes of 4 SPEC CPU2006 apps on a 64-core CMP.



(a) Weighted speedups (b) Traffic breakdown

Figure 15: Weighted speedup distribution and traffic breakdown of 50 mixes of eight 8-thread SPEC OMP2012 apps on a 64-core CMP.

performance is a second-order effect.

Fig. 13 shows that CDCS maintains high weighted speedups throughout the whole range, while Jigsaw+R and Jigsaw+C work poorly on 1–8 app mixes. To see why, Fig. 14 shows the weighted speedup distribution and network traffic breakdown for the 4-app case. On-chip latency (L2-LLC) dominates Jigsaw's latency. In these mixes, cache capacity is plentiful, so large VC allocations hurt on-chip latency more than they help off-chip latency. CDCS's latency-aware allocation avoids using banks when detrimental, and yields most of the speedup in the 4-app mixes, as shown in Fig. 12b. At 4 apps, CDCS achieves 28% gmean weighted speedup, while Jigsaw+R sees 17% and Jigsaw+C sees 6%. Overall, latency-aware allocation becomes more important as capacity becomes plentiful.

B. Multithreaded mixes

Fig. 15a shows the distribution of weighted speedups for 50 mixes of eight 8-thread SPEC OMP2012 applications (64 threads total) running on the 64-core CMP. CDCS achieves gmean weighted speedup of 21%. Jigsaw+R achieves 14%, Jigsaw+C achieves 19%, and R-NUCA achieves 9%. Trends are reversed: on multi-threaded benchmarks, Jigsaw works better with clustered thread placement than with random (S-NUCA and R-NUCA are still insensitive). CDCS sees smaller benefits over Jigsaw+C. Fig. 15b shows that they get about the same network traffic, while others are noticeably worse.

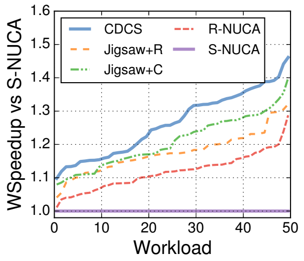
Fig. 16a shows the distribution of weighted speedups with under-committed system running mixes of four 8-thread applications. CDCS increases its advantage over Jigsaw+C, as it has more freedom to place threads. CDCS dynamically clusters or spreads each process as the context demands: shared-heavy processes are clustered, and private-heavy processes are spread out. Fig. 16b illustrates this behavior by showing the thread and data placement of a specific mix, where one of the apps, `mgrid` (process P0), is private and intensive, and the others, `md` (P1), `ilbdc` (P2), and `nab` (P3) access mostly shared data. CDCS gives most capacity to `mgrid`, spreads its threads over the CMP, and tightly clusters P1–3 around their shared data.

From the results of Sec. VI-A and Sec. VI-B, we can see that Jigsaw+R and Jigsaw+C help different types of programs, but no option is best in general. Yet by jointly placing threads and data, CDCS always provides the highest performance across all mixes. Thus, beyond improving performance, CDCS provides an important advantage in *guarding against pathological behavior incurred by fixed policies.*

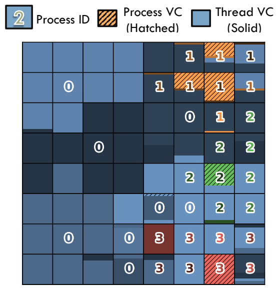
C. CDCS analysis

Reconfiguration overheads: Table 3 shows the CPU cycles spent, on average, in each of the steps of the reconfiguration procedure. Overheads are negligible: each reconfiguration consumes a mere 0.2% of system cycles. Sparse GMON curves improve Peekahead's runtime, taking 1.2 Mcycles at 64 cores instead of the 7.6 Mcycles it would require with 512-way UMONs [4]. Although thread and data placement have quadratic runtime, they are practical even at thousands of cores (1.2% projected overhead at 1024 cores).

Alternative thread and data placement schemes: We have considered more computationally expensive alternatives for thread and data placement. First, we explored using integer linear programming (ILP) to produce the best achievable data placement. We formulate the ILP problem by minimizing Eq. 2 subject to the bank capacity and VC allocation constraints, and solve it in Gurobi [19]. ILP data placement improves weighted speedup by 0.5% over CDCS on 64-app mixes. However, Gurobi



(a) Weighted speedups



(b) Case study

Figure 16: Weighted speedups for 50 mixes of four 8-thread SPEC OMP2012 apps (32 threads total) on a 64-core CMP, and case study with private-heavy and shared-heavy apps.

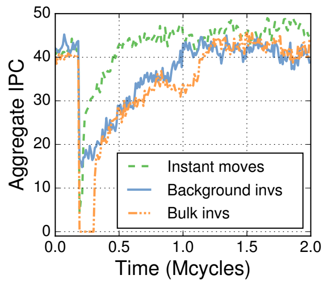


Figure 17: IPC throughput of a 64-core CMP with various data movement schemes during one reconfiguration.

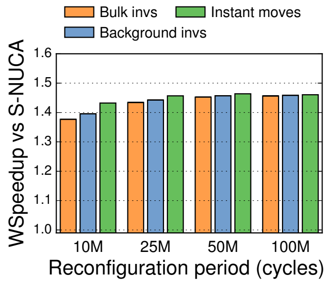


Figure 18: Weighted speedup of 64-app mixes for various data movement schemes vs. reconfiguration period.

Threads / Cores	**16 / 16**	**16 / 64**	**64 / 64**
Capacity allocation (Mcycles)	0.30	0.30	1.20
Thread placement (Mcycles)	0.29	0.80	3.44
Data placement (Mcycles)	0.13	0.36	1.85
Total runtime (Mcycles)	0.72	1.46	6.49
Overhead @ 25 ms (%)	0.09	0.05	0.20

Table 3: CDCS runtime analysis. Avg Mcycles per invocation of each reconfiguration step, total runtime, and relative overhead.

takes about 219 Mcycles to solve 64-cores, far too long to be practical. We also formulated the joint thread and data placement ILP problem, but Gurobi takes at best tens of minutes to find the solution and frequently does not converge.

Since using ILP for thread placement is infeasible, we have implemented a simulated annealing [61] thread placer, which tries 5000 rounds of thread swaps to find a high-quality solution. This thread placer is only 0.6% better than CDCS on 64-app runs, and is too costly (6.3 billion cycles per run).

We also explored using METIS [31], a graph partitioning tool, to jointly place threads and data. We were unable to outperform CDCS. We observe that graph partitioning methods recursively divide threads and data into equal-sized partitions of the chip, splitting around the center of the chip first. CDCS, by contrast, often clusters one application around the center of the chip to minimize latency. In trace-driven runs, graph partitioning increases network latency by 2.5% over CDCS.

Geometric monitors: 1K-line, 64-way GMONs match the performance of 256-way UMONs. UMONs lose performance below 256 ways because of their poor resolution: 64-way UMONs degrade performance by 3% on 64-app mixes. In contrast, unrealistically large 16K-line, 1K-way UMONs are only 1.1% better than 64-way GMONs.

Reconfiguration schemes: We evaluate several LLC reconfiguration schemes: demand moves plus background invalidations (as in CDCS), bulk invalidations (as in Jigsaw), and idealized, instant moves. The main benefit of demand moves is avoiding global pauses, which take 114 Kcycles on average, and up to 230 Kcycles. While this is a 0.23% overhead if reconfigurations are performed every 50 Mcycles (25 ms), many applications cannot tolerate such pauses [16, 46]. Fig. 17 shows a trace of aggregate IPC across all 64 cores during one representative reconfiguration. This trace focuses on a small time interval to show how performance changes right after a reconfiguration, which happens at 200 Kcycles. By serving lines with demand moves, CDCS prevents pauses and achieves smooth reconfigurations, while bulk invalidations pause the chip for 100 Kcycles in this case. Besides pauses, bulk invalidations add misses and hurt performance. With 64 apps (Fig. 11), misses are already frequent and per-thread capacity is scarce, so the average slowdown is 0.5%. With 4 apps (Fig. 14), VC allocations are larger and threads take longer to warm up the LLC, so the slowdown is 1.4%. Note that since SPEC CPU2006 is stable for long phases, these results may underestimate overheads for apps with more time-varying behavior. Fig. 18 compares the weighted speedups of different schemes when reconfiguration intervals increase from 10 Mcycles to 100 Mcycles. CDCS outperforms bulk invalidations, though differences diminish as reconfiguration interval increases.

Bank-partitioned NUCA: CDCS can be used without fine-grained partitioning (Sec. IV-I). With the parameters in Table 2 but 4 smaller banks per tile, CDCS achieves 36% gmean weighted speedup (up to 49%) over S-NUCA in 64-app mixes, vs. 46% gmean with partitioned banks. This difference is mainly due to coarser-grain capacity allocations, as CDCS allocates full banks in this case.

VII. Conclusions

We have identified how thread placement impacts NUCA performance, and presented CDCS, a practical technique to perform coordinated thread and data placement. CDCS uses a combination of hardware and software techniques to achieve performance close to idealized schemes at low overheads. As a result, CDCS improves performance and energy efficiency over both thread clustering and state-of-the-art NUCA techniques.

Acknowledgments

We sincerely thank Christina Delimitrou, Joel Emer, Mark Jeffrey, Harshad Kasture, Suvinay Subramanian, and the anonymous reviewers for their helpful feedback on prior versions of this manuscript. This work was supported in part by NSF grant CCF-1318384 and by DARPA PERFECT under contract HR0011-13-2-0005. Po-An Tsai was partially supported by a MIT EECS Jacobs Presidential Fellowship.

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] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *Proc. HPCA-21*, 2015.

[6] S. Bell, B. Edwards, J. Amann *et al.*, "TILE64 processor: A 64-core SoC with mesh interconnect," in *Proc. ISSCC*, 2008.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," in *Proc. USENIX ATC*, 2011.

[8] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proc. ISCA-33*, 2006.

[9] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "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. Cook, M. Moreto, S. Bird *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA-40*, 2013.

[13] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *SC Plenary Talk*, 2010.

[14] R. Das, R. Ausavarungnirun, O. Mutlu *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *Proc. HPCA-19*, 2013.

[15] M. Dashti, A. Fedorova, J. Funston *et al.*, "Traffic management: a holistic approach to memory placement on NUMA systems," in *Proc. ASPLOS-18*, 2013.

[16] J. Dean and L. Barroso, "The Tail at Scale," *CACM*, vol. 56, 2013.

[17] H. Esmaeilzadeh, E. Blem, R. St Amant *et al.*, "Dark Silicon and The End of Multicore Scaling," in *Proc. ISCA-38*, 2011.

[18] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proc. MICRO-40*, 2007.

[19] Gurobi, "Gurobi optimizer reference manual version 5.6," 2013.

[20] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proc. ISCA-36*, 2009.

[21] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, 2011.

[22] N. Hardavellas, I. Pandis, R. Johnson, and N. Mancheril, "Database Servers on Chip Multiprocessors: Limitations and Opportunities," in *Proc. CIDR*, 2007.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (5th ed.)*. Morgan Kaufmann, 2011.

[24] H. J. Herrmann, "Geometrical cluster growth models and kinetic gelation," *Physics Reports*, vol. 136, no. 3, pp. 153–224, 1986.

[25] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008.

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

[27] Intel, "Knights Landing: Next Generation Intel Xeon Phi," in *SC Presentation*, 2013.

[28] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi *et al.*, "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Par. Dist. Sys.*, vol. 18, no. 8, 2007.

[29] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam *et al.*, "CRUISE: Cache replacement and utility-aware scheduling," in *Proc. ASPLOS*, 2012.

[30] D. Kanter, "Silvermont, Intels Low Power Architecture," 2013. [Online]. Available: http://www.realworldtech.com/silvermont/

[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, 1998.

[32] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *Proc. ASPLOS-19*, 2014.

[33] R. Kessler, M. Hill, and D. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE T. Comput.*, vol. 43, 1994.

[34] C. Kim, D. Burger, and S. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.

[35] N. Kurd, S. Bhamidipati, C. Mozak *et al.*, "Westmere: A family of 32nm IA processors," in *Proc. ISSCC*, 2010.

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

[37] S. Li, J. H. Ahn, R. Strong *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, 2009.

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

[39] Z. Majo and T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *Proc. ISMM*, 2011.

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

[41] M. Marty and M. Hill, "Virtual hierarchies to support server consolidation," in *Proc. ISCA-34*, 2007.

[42] J. Merino, V. Puente, and J. Gregorio, "ESP-NUCA: A low-cost adaptive non-uniform cache architecture," in *Proc. HPCA-16*, 2010.

[43] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.

[44] M. Moreto, F. J. Cazorla, A. Ramirez *et al.*, "FlexDCP: A QoS framework for CMP architectures," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, 2009.

[45] T. Nowatzki, M. Tarm, L. Carli *et al.*, "A general constraint-centric scheduling framework for spatial architectures," in *Proc. PLDI-34*, 2013.

[46] J. Ousterhout, P. Agrawal, D. Erickson *et al.*, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, 2010.

[47] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint archive*, no. 2005/280, 2005.

[48] P. N. Parakh, R. B. Brown, and K. A. Sakallah, "Congestion driven quadratic placement," in *Proc. DAC-35*, 1998.

[49] J. Park and W. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proc. SPAA-22*, 2010.

[50] F. Pellegrini and J. Roman, "SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. HPCN*, 1996.

[51] M. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," in *Proc. HPCA-10*, 2009.

[52] 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.

[53] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proc. ISCA-38*, 2011.

[54] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA-40*, 2013.

[55] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proc. ASPLOS-8*, 2000.

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

[57] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proc. Eurosys*, 2007.

[58] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[59] A. Tumanov, J. Wise, O. Mutlu, and G. R. Ganger, "Asymmetry-aware execution placement on manycore chips," in *SFMA-3*, 2013.

[60] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," in *Proc. ASPLOS*, 1996.

[61] D. Wong, H. W. Leong, and C. L. Liu, *Simulated annealing for VLSI design*. Kluwer Academic Publishers, 1988.

[62] C. Wu and M. Martonosi, "A Comparison of Capacity Management Schemes for Shared CMP Caches," in *WDDD-7*, 2008.

[63] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA*, 2005.

[64] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. ASPLOS*, 2010.