Nexus: A New Approach to Replication in Distributed Shared Caches

Po-An Tsai
MIT CSAIL
poantsai@csail.mit.edu

Nathan Beckmann
CMU SCS
beckmann@cs.cmu.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

Abstract—Last-level caches are increasingly distributed, consisting of many small banks. To perform well, most accesses must be served by banks near requesting cores. An attractive approach is to *replicate* read-only data so that a copy is available nearby. But replication introduces a delicate tradeoff between capacity and latency: too little replication forces cores to access faraway banks, while too much replication wastes cache space and causes excessive off-chip misses.

Workloads vary widely in their desired amount of replication, demanding an adaptive approach. Prior adaptive replication techniques only replicate data in each tile's local bank, so they focus on selecting *which* data to replicate. Unfortunately, data that is not replicated still incurs a full network traversal, limiting the performance of these techniques.

We argue that a better strategy is to let cores share replicas and that adaptive schemes should focus on selecting *how much* to replicate (i.e., how many replicas to have across the chip). This idea fully exploits the latency-capacity tradeoff, achieving qualitatively higher performance than prior adaptive replication techniques. It can be applied to many prior cache organizations, and we demonstrate it on two: Nexus-R extends R-NUCA, and Nexus-J extends Jigsaw. We evaluate Nexus on HPC and server workloads running on a 144-core chip, where it outperforms prior adaptive replication schemes and improves performance by up to 90% and by 23% on average across all workloads sensitive to replication.

Keywords-cache, data replication, NUCA, multicore

I. INTRODUCTION

To scale beyond a few cores, future systems must tackle the high costs of data movement, which are orders of magnitude more expensive than basic compute operations for most applications [15, 20]. To this end, last-level caches (LLCs) have become distributed and expose non-uniform cache access (NUCA [30]): each core enjoys fast and cheap accesses to nearby cache banks, but accesses to faraway banks are slow and expensive. To scale, future systems must serve most accesses from nearby cache banks.

Prior work in dynamic NUCA (D-NUCA) has studied *replication* of read-only data as an effective way to reduce data movement, allowing multiple replicas of the same line to exist in different cache banks [49]. Replication exploits the insight that an application's working set often does not use the full cache capacity. For example, Fig. 1 shows the data placement for different replication policies in a tiled multicore. Each square represents a tile, with a core and an LLC bank. There are four addresses (A, B, C, and D). Each letter in Fig. 1 represents a replica of the corresponding data. At one extreme, Fig. 1a shows a data layout with no replication (i.e., only one replica, which we call a *replication*
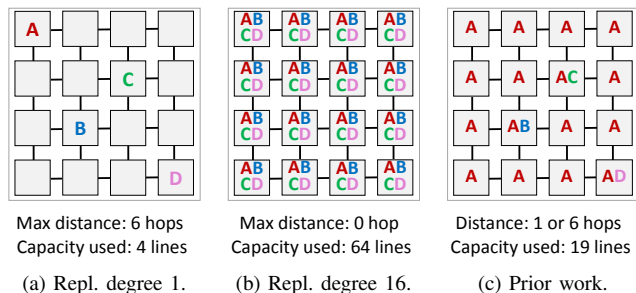

Figure 1: Data replication in NUCA with different degrees. Higher replication degree reduces the distance between cores and data, but consumes more cache space.

degree of 1). Data is spread across the chip, and, in the worst case, cores have to cross the chip to access the data. At the other extreme, Fig. 1b shows the layout with full replication (i.e., a replica in all 16 banks, or *replication degree* of 16). Full replication eliminates network traversals, but consumes much more cache space.

Neither of these extremes works well in all cases. For example, consider a multithreaded application that uniformly accesses 16 MB of read-only data on a system with 64×1 MB banks. Not replicating data (like Fig. 1a) is inefficient, since cores must cross the chip to access data while three-quarters of the cache remains empty. But replicating data into the local bank (like Fig. 1b) also works poorly because the footprint of read-only data exceeds the capacity of the local bank. In this system, only 1 MB of the read-only data fits in a local bank, so the remaining 15 MB must be served elsewhere.

Prior work has proposed *adaptive replication* to navigate these extremes. These schemes were developed in the context of *directory-based D-NUCAs*, where a core first accesses its local cache bank and, upon a miss, must check a global directory, incurring a full-chip network traversal (Sec. II). Because capacity in the local bank is scarce, prior adaptive replication schemes [4, 16, 21, 31, 34, 49] focus on *selecting which lines to replicate*. That is, they select which of A, B, C, and D to replicate into the local bank, adapting between no (Fig. 1a) and full (Fig. 1b) replication for different data. Compare, e.g., A vs. B, C, and D in Fig. 1c. These schemes are beneficial when some data is accessed particularly frequently, but since they only adapt between the extremes of replication, their performance improvements are often limited—e.g., they can serve *at most $1/16^{th}$* of accesses locally in the above example.

A new approach to adaptive replication: A more effective strategy is to spread replicas across cache banks and let cores access their closest replica. In this example, the best policy is to replicate the 16 MB *four times* across the chip (*replication degree* of 4, Fig. 2). Further replication is undesirable because a 64 MB LLC can only fit four copies of a 16 MB working set, so further replication would cause expensive off-chip misses. With four replicas, *all* accesses are served from nearby banks (albeit at slightly larger distance), a major improvement over prior schemes that serve just 1/16th locally.
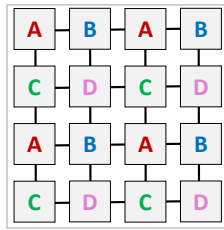
Max distance: 2 hops
Capacity used: 16 lines

Figure 2: Replication degree of 4 is best.

One challenge is how cores can quickly find their closest replica. Classic directory-based D-NUCAs must check the global directory before accessing nearby banks, but more recent NUCA organizations like R-NUCA [19] and Jigsaw [6] avoid a global directory and thus support fast accesses to nearby banks. This grants these schemes enormous flexibility: data can be mapped to the local bank, nearby banks, or banks across the chip, and cores can share replicas.

Different replication degrees make different tradeoffs between capacity and latency, but prior adaptive replication schemes do not fully exploit this tradeoff. Instead, they replicate either everywhere or not at all, and focus on selecting *which* data to replicate. This sufficed for small systems, where there are few degrees between the extremes, but leaves significant performance on the table as systems scale. For many workloads, the best replication degree achieves qualitatively better performance than either extreme. Thus, the key idea of this paper is that adaptive replication techniques should focus on *how much* to replicate.

Why focus on *how much* to replicate? Fig. 3 answers this using a simple analytical model that generalizes the example in Fig. 1. We model a multithreaded application where each thread regularly scans over shared read-only data, running on a 144-core system with a 512 KB L2 bank per core, as in our evaluation (Sec. VI-A). Our model gives the average distance to the closest replica under different replication schemes. Fig. 3 shows how the average L2 access latency (*y*-axis) changes with the footprint of the read-only data (*x*-axis, log-scale). We have verified this model against a microbenchmark in simulation. See the appendix for details.

At one extreme, **full replication** has each core first check its local L2 bank and then, upon a miss, check a remote, "home" L2 bank. When the read-only data fits in an L2 bank (i.e., footprint < 512 KB), cores can access it at low latency. However, as the footprint grows beyond a single bank, an increasing fraction of accesses miss and **full replication**'s access latency is quickly dominated by memory latency.

At the other extreme, **no replication** combines all L2 banks into a single shared cache, so all accesses are to remote banks.
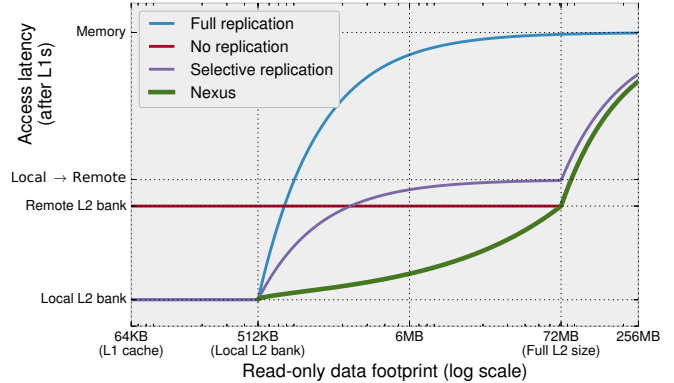
Figure 3: Adapting the replication degree lets Nexus achieve qualitatively better performance than prior work over a wide range of data footprints.

As long as the read-only data fits in the L2 (i.e., footprint < 72 MB), cores can find the data on-chip in a remote L2 bank. Beyond the L2 size, an increasing fraction of accesses miss and, once again, access latency is quickly dominated by memory latency. **No replication** thus can fit a larger footprint than **full replication**, but incurs a higher latency to do so.

Prior **selective replication** schemes combine the benefits of **full** or **no** replication, choosing whether to replicate depending on the workload. When the footprint is small (i.e., footprint < 512 KB), **selective replication** achieves access latency as low as **full replication**, and when the footprint is larger but still fits in the L2 (i.e., 512 KB < footprint < 72 MB), **selective replication** achieves access latency close to **no replication**. (But slightly worse because **selective replication** must check the local bank first.)

We propose that last-level caches should adapt the replication degree, and we call this concept **Nexus**. Nexus adapts replication degree to the workload, creating multiple replicas that are shared among cores. Moreover, because Nexus builds on recent directory-less D-NUCAs (Sec. II), cores only need to check a *single L2 bank* to find the closest replica. At the extremes, **Nexus** performs similarly to **selective replication**: when the read-only data fits in a single L2 bank, cores access their local L2 bank; and when it barely fits in the L2, cores access a remote bank. (**Nexus** slightly outperforms **selective replication** since it only checks a single L2 bank).

However, as Fig. 3 shows, at intermediate read-only footprints, **Nexus** handily outperforms prior approaches. For example, when the footprint is 6 MB, **Nexus** creates twelve 6 MB replicas spread evenly throughout the chip (similar to Fig. 2). Cores enjoy low access latency to their nearby replica, whereas in **selective replication** cores must access a remote L2 bank. The latency gap is large: 2.6× at 6 MB. Thus, for a large range of footprints—between 512 KB and 72 MB—**Nexus** achieves qualitatively better performance than **selective replication**.

Contributions: Applications vary widely in their working set size, how frequently they access read-only data, and how they share read-only data [3]. We find that *applications thus have strong preferences for different replication degrees,*

with performance differing by up to $2.5\times$ across degrees. Moreover, some applications prefer different degrees on different inputs, demanding a dynamic approach (Sec. III).

We present two implementations of Nexus that transparently optimize the replication degree: Nexus-R extends R-NUCA [19] to replicate all read-only data (not just instructions) and adapt the replication degree through set-sampling (Sec. IV). Nexus-J extends Jigsaw [6] to recognize read-only data and extends Jigsaw's runtime to adapt the replication degree (Sec. V). Nexus-R is simple and works well for single multithreaded programs, while Nexus-J adds modest complexity to perform better on multiprogram workloads.

We evaluate Nexus with scientific and server multithreaded workloads, which often have large read-only data and code footprints [3, 27] (Sec. VI). At 144 cores, Nexus improves performance by 23% on average and by up to 90%, outperforming a state-of-art selective replication scheme by 20%.

II. BACKGROUND

Dividing capacity across several banks mitigates growing wire delays, but exposes non-uniform cache access (NUCA) to different banks. Common many-core NUCA designs, such as Intel Knights Landing [45] and Tilera TILE-Gx [46], use a tiled organization like the one shown in Fig. 4. Each tile has a core, private caches, and an LLC bank. Tiles communicate through an on-chip network. The simplest scheme is static NUCA (S-NUCA) [30], which uniformly spreads data across banks with a fixed line-bank mapping. While S-NUCA is simple, it suffers from a large average network distance.
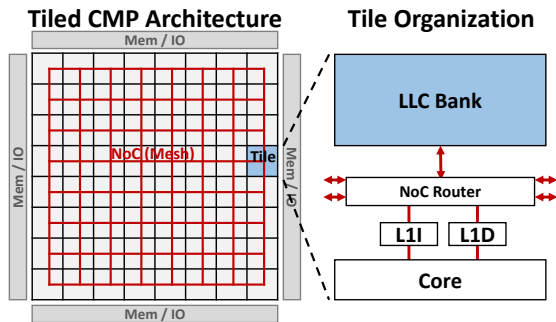

Figure 4: Tiled architecture. Each tile contains a core, private L1I/D, and a shared LLC bank. A mesh NoC connects all tiles.

Dynamic NUCA schemes (D-NUCAs) reduce data movement by dynamically placing data near where it is used. D-NUCAs and their replication strategies can be broadly classified into two categories based on if they require a global directory. Historically, directory-based D-NUCAs came first and are the context in which prior work has considered adaptive replication. Table I summarizes the key differences among replication strategies, which we explain in detail next.
Directory-based D-NUCAs: D-NUCAs originated from a private-cache baseline. The local bank at each tile is treated as a private cache and other tiles act as a large, backing shared cache [5]. Cores first check their local bank, then

TABLE I: COMPARISON OF NEXUS AND PRIOR D-NUCAS.

Scheme	Data replication	Single lookup	Dynamic replication	Replicas shared across cores
Victim replication [49]	✔	✘	✘	✘
ASR [4]	✔	✘	✔	✘
Locality-aware replication [31]	✔	✘	✔	✘
R-NUCA [19]	▲[1]	✔	✘	✔
Jigsaw [6]	✘	✔	✘	✘
Nexus	✔	✔	✔	✔

[1]Instructions only

access the line's home bank on a miss, where a directory stores the location of all sharers of the line. These designs place all LLC capacity under a directory coherence protocol, which adds significant area and energy overheads.

Replication naturally arises in these organizations as local banks keep copies of read-shared data [49]. However, since capacity is limited, a key question becomes *which data should be replicated?* Prior work has explored many techniques to decide what lines should be replicated [4, 9, 11, 22, 23, 31, 34, 38]. However, these schemes are still limited to replicating in the local bank, and cores do not share replicas. This is a significant limitation—read-only data often does not fit in a single cache bank, and directory-based schemes cannot fully exploit the latency-capacity tradeoff, as we have seen in Sec. I.

Moreover, private caches (e.g., L1s) already replicate data, serving most programs' hottest data before it ever reaches the LLC. It is only the lukewarm data, which is somewhat frequently accessed but does not fit in private caches, that benefits from selective replication in the LLC. Thus, a more important problem at the LLC is deciding *how much* to replicate the remaining data, not selecting *which* of the remaining data merits replication.

Directory-less D-NUCAs: Some D-NUCA schemes instead build from a shared-cache baseline and do not require a directory [2, 6, 10, 12, 19, 26]. Banks are not under a coherence protocol; instead, virtual memory is used to place and replicate data. A page's on-chip location is tracked in software by extending each page table entry with metadata that directly or indirectly determines the page's LLC bank. This metadata is cached in the TLB so that the LLC bank can be quickly determined upon an L1 miss. Page table entries are updated in response to program behavior; e.g., if a read-only page is written, its LLC bank might change, forcing the OS to invalidate all replicas through a TLB shootdown. These reclassifications are expensive but very rare.

Prior work [14, 19, 41] has studied how these virtual memory modifications interact with other OS mechanisms, such as TLB shootdowns and thread migrations, and has shown that they introduce modest complexity and overheads. Parallel TLB shootdowns [13, 37] can be used to significantly reduce page reclassification overheads. We assume the same basic OS support for directory-less D-NUCA as our baseline schemes [6, 19], and Nexus introduces no new complications.
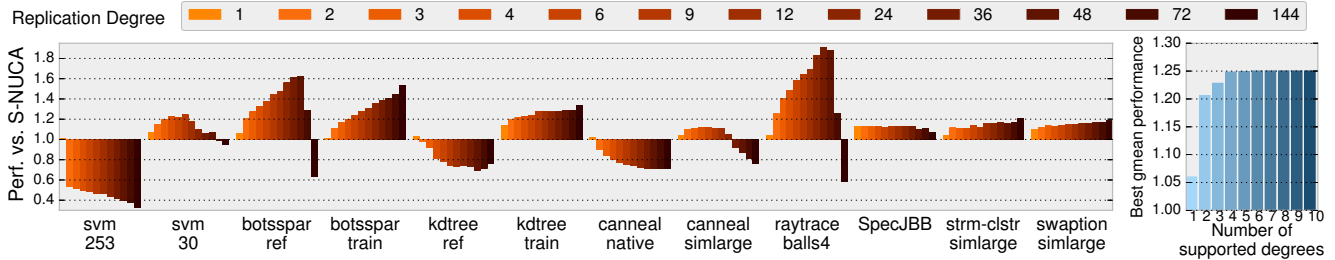
Figure 5: Exhaustive replication degree study. Left: Performance of replication-sensitive applications using modified R-NUCA with different fixed degrees. Right: Best gmean performance when supporting *n* different degrees.

The main benefit of directory-less D-NUCAs is that, since software maintains coherence at coarse granularity, each core needs to lookup *only a single location* for a given address, without first checking a global directory. This grants these schemes enormous flexibility in where they place data and how cores share replicas, the key advantage that lets Nexus exploit many different replication degrees.

Unfortunately, adaptive replication has received scant attention in directory-less D-NUCAs. While some schemes divide pages into classes and use different placement policies for each class, none adapt the replication degree to the application. R-NUCA [19] specializes placement for three classes of data (instructions, private data, and shared data), replicates instructions at a statically *fixed* degree, and does not replicate shared data. Jigsaw [6, 7] lets software divide the distributed cache into finely-sized *virtual caches* and maps pages to different virtual caches, but Jigsaw never replicates data. Even though prior work [23] has argued for flexible sharing degrees, none has studied how to adapt to the right degree on-the-fly for directory-less D-NUCAs.

Nexus fills this gap. We show that adapting the replication degree in directory-less D-NUCAs gives significant performance benefits and outperforms prior, directory-based adaptive replication strategies (Fig. 3, Sec. VI-B). We also show that simply replicating read-only data in directory-less D-NUCAs at a fixed degree is insufficient: *adapting the replication degree is the key to high performance* (Sec. VI-C).

III. WORKLOAD CHARACTERIZATION

To motivate Nexus's design, we study applications' sensitivity to replication degree and characterize how they share data. We choose representative *replication-sensitive* applications that illustrate our points as simply and clearly as possible and defer an exhaustive evaluation to Sec. VI.

A. Sensitivity to replication degree

A few replication degrees suffice: The best replication degree varies across applications and inputs, but how many different degrees must the system support? One could imagine supporting degrees at fine granularity, but in fact a few coarsely-spaced degrees are enough.

To see why, consider the cluster of tiles sharing a single replica (e.g., quadrants in Fig. 2). Increasing the replication

degree decreases both the distance across the cluster and the cache capacity in the cluster. But the distance does not decrease proportionally—in a mesh, it follows the square root of cluster size. And misses do not decrease proportionally, either—a typical rule-of-thumb is that miss rate follows the square root, too. Since larger clusters give diminishing returns, it suffices to consider a few, unevenly spaced degrees.

Fig. 5 validates this intuition by showing the performance of many workloads on R-NUCA, modified to replicate all read-only data (see Sec. IV), at 12 different replication degrees. While the best replication degree varies widely across applications, similar degrees generally yield similar performance. For example, degrees 9 and 12 perform similarly across studied workloads, as do other pairs. Therefore, a small subset of degrees gets most of the benefit.

The right of Fig. 5 shows the gmean performance when limited to supporting a few replication degrees. That is, we consider all possible combinations of *n* degrees from the 12 in Fig. 5 and use the best of these *n* for each application. From all combinations of *n* degrees, we then select the combination that achieves the best gmean performance. This graph shows that supporting 4 degrees matches the gmean performance of supporting all 10 degrees. With 144 cores, the best choice of 4 degrees is 1, 9, 36, and 144. These are the degrees that we will support in Nexus.

No single replication degree works in all cases: Fig. 6 shows the performance of three applications with different inputs on a 144-core system, again using modified R-NUCA.

`kdtree` prefers full replication (degree 144) with input `train`, but no replication (degree 1) with input `ref`. Meanwhile, `svm` prefers replication degree 9 for input 30, but no replication (degree 1) for input 253. Finally, `botsspar` prefers full replication on input `train`, but replication degree 36 on input `ref`. No single replication degree works well for all applications, and each replication degree is best for some application and input set. Moreover, the choice matters: choosing the wrong replication degree affects performance by up to $2.5\times$ (`botsspar` with large input). Therefore, *it is crucial to adapt the replication degree to applications*.

To understand these results, consider the *read-only footprint* for these applications with different replication degrees, shown in Fig. 7. We define the footprint as the smallest cache size where the miss rate is $< 1\%$. Dashed lines indicate the
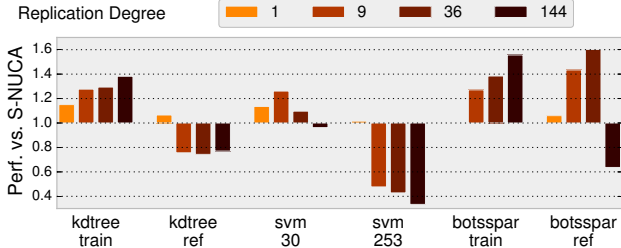
Figure 6: Performance of R-NUCA vs. static NUCA for different replication degrees of read-only data. No single replication degree works well in all cases.
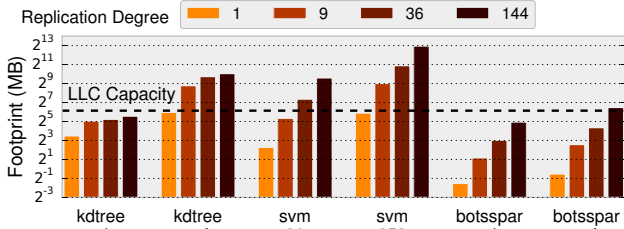

Figure 7: Memory footprints of read-only data with different replication degrees.

total LLC capacity in the system. Since these applications are dominated by read-only data, *replication is beneficial up until the read-only footprint exceeds the LLC capacity*.

Given this insight, it is not surprising that applications strongly prefer different replication degrees, as they have very different read-only footprints. For `kdtree-train` and `botsspar-train`, replication degree of 144 is best because the read-only footprint for each thread is very small, so the total footprint with replication degree 144 is still smaller than the LLC. On the other hand, when the read-only footprint is very large, such as in `kdtree-ref` and `svm-253`, replication is harmful because the footprint does not fit in the LLC, and replication increases the number of misses.

Applications want intermediate replication degrees: However, many applications whose per-thread footprint exceeds a single LLC bank can still benefit from replication. In `svm-30` and `botsspar-ref`, the per-thread read-only footprint is large, so the total read-only footprint with replication degree 144 is larger than the LLC. The best choice for these apps is to use an intermediate replication degree: degree 9 for `svm-30` and degree 36 for `botsspar-train`. By replicating less aggressively, the system can cache the entire working set *and* reduce network traversals, as in Fig. 2. Prior adaptive replication schemes, which replicate only in the local bank, cannot help these apps: they suffer either many misses (with replication) or many network traversals (without).

From these observations, it is clear that replication is not always beneficial, and even when it is, a fixed replication degree cannot work well for all cases. The best replication degree depends on system size, LLC size, application behavior, input size, and even interactions between applications. It is unreasonable for programmers to reason about all of these

factors. Instead, *the system should transparently optimize the replication degree*, freeing the user of this burden.

Replication degrees are stable: We find that workloads quickly converge to a stable replication degree that rarely changes (although it depends on application and input). Hence, while Nexus does support dynamic adaptation and this is important for real-world systems where apps come and go, this feature is not emphasized in our evaluation.

B. Classification in directory-less D-NUCAs

Coarse-grain data classification suffices: Nexus builds on top of directory-less D-NUCAs, which let cores share replicas and find the closest one with a single lookup. The cost of this flexibility is that these schemes classify data at page granularity, unlike directory-based D-NUCAs that replicate at line granularity. Similar to prior work [7, 14, 19], we find there is little performance penalty with 4 KB pages. Fig. 8 shows the performance of modified R-NUCA with different page sizes, from 64 B to 4 KB, when using the best degree for each workload and perfect TLBs. Using 64 B pages improves performance by only 3% over 4 KB pages.
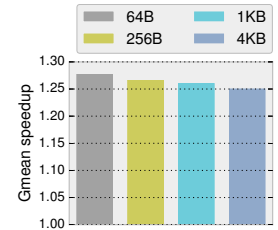

Figure 8: Performance of modified R-NUCA with different page sizes.

Data classifications are stable over time: Nexus only replicates read-only data, so the amount of data classified as read-only can strongly influence Nexus's performance. In designing Nexus, we originally believed that dynamic data classification would be a critical feature. However, to our surprise, we find that it is not: across sixty server and scientific benchmarks, only one benchmark (`barnes`) benefits significantly from dynamic reclassification. We thus present Nexus with one-way data classification for simplicity and clarity. Its limitations can be addressed by dynamically reclassifying pages, as in recent work [17, 41], which we evaluate as a case study (Sec. VI-E).

IV. NEXUS ON R-NUCA (NEXUS-R)

Any prior directory-less D-NUCA can benefit from adapting the replication degree. The first implementation we describe is Nexus-R, which extends R-NUCA [19] to replicate all read-only data at an adaptive degree. Our design goal is a simple scheme that works well on single multithreaded applications. Nexus-R monitors the performance of different replication degrees using set sampling [40] and adopts the degree that performs best. Since all threads in a process should agree on their replication degree, we introduce simple OS support to coordinate degrees within a process.

A. Background: Reactive NUCA

Reactive NUCA (R-NUCA [19]) classifies pages into three categories (instructions, private data, or shared data) and uses

different placement policies for each. Fig. 9 illustrates how R-NUCA places data by showing which banks are possible locations for data in each category, starting from the core in the center of the chip. **Private** data is mapped to the local bank to minimize access latency, and **shared** data is striped across banks to keep a single copy for coherence. **Instructions**, on the other hand, are replicated

Figure 9: Data placement in R-NUCA.

at a fixed degree with 1 replica for every 4 tiles. This is because the server workloads studied in R-NUCA [19] have instruction footprints that cannot fit in local bank, but do fit in a cluster of 4 banks. R-NUCA also introduces *rotational interleaving*, a specialized lookup mechanism that finds the closest replica from every core, by labeling banks 1 through 4 and using the closest bank of each label (see Fig. 9). The replication degree in R-NUCA is determined by the cluster size (e.g., at 36 cores, cluster size 4 gives degree $36/4 = 9$).

R-NUCA updates data classification by leveraging the virtual memory system. Pages start as **private** to the thread that first touches them, and any change in their usage is captured on a TLB miss. For example, when another thread first accesses an address, it triggers a TLB miss and the page is reclassified to **shared**. This changes its location (i.e., it is now striped across banks throughout the chip), so the OS shoots down the TLB entry in the original core, after which both cores now access the shared page normally. While TLB shootdowns are expensive, page reclassifications are rare.

B. Data classification in Nexus-R

Nexus-R extends R-NUCA to replicate *all* read-only pages. Fig. 10 shows the resulting per-page state transition diagram. All read-only pages, not just instructions, are mapped to clusters and replicated.
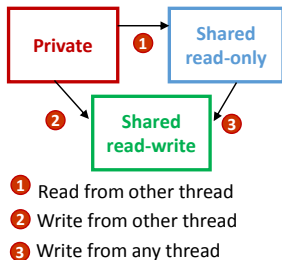

Figure 10: Page state transition diagram in Nexus-R.

Figure 11: Example data mapping in Nexus-R.

A page starts as **private**, a read access from other core upgrades it to **shared read-only**, and a write access from other core upgrades it to a **shared read-write** page.
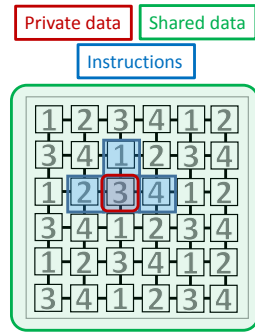
C. Replication in Nexus-R

Nexus-R monitors the latency of different replication degrees to select the best one. We use set sampling [40], but differ from prior work in that *(i)* we monitor fine-grain latencies, vs. counting discrete events; and *(ii)* we choose from several options, vs. making a binary choice.

Supporting multiple degrees: The replication degree in R-NUCA is controlled by the cluster size, but R-NUCA only supports power-of-2 cluster sizes. In fact, it is straightforward to support arbitrary sizes with a simple indexing function. For example, to support size of 9 (degree of 4) in a 6x6 chip, we label tiles as in Fig. 11. Each tile accesses the nearest bank for each label. Thus, each tile's cluster consists of the 9 closest banks with different labels (computed ahead-of-time). In each core, Nexus-R adds a register to store the current cluster size and a circuit that finds the label for a given address, which does not need complex arithmetic [43]. This generalization supports arbitrary cluster sizes, and thus arbitrary replication degrees.

Monitoring degrees: Each degree is assigned to a small number of *sampling sets* that always use that degree, as shown in Fig. 12. Nexus-R compares the average memory access latency of each degree by monitoring the latency of accesses to each sampling set.

Specifically, Nexus-R maintains a long-run cumulative latency difference between all pairs of degrees. We denote the latency difference between degrees a and b as Δ_a^b. By convention, positive Δ_a^b indicates that b is higher latency, and negative Δ_a^b indicates that a is higher latency. With four supported degrees, Nexus-R must maintain six counters (Δ_1^4, Δ_1^9, Δ_1^{36}, Δ_4^9, Δ_4^{36}, and Δ_9^{36}).

When a sampled access completes, Nexus-R updates the counters by adding or subtracting its latency as appropriate, as shown in Fig. 13. For instance, when a sampled access to degree 4 completes, we add its latency to Δ_1^4 and subtract its latency from Δ_4^9 and Δ_4^{36}. These counters all saturate at some maximum absolute value (2^{17} in our evaluation).

For the latency differences to be meaningful, it is important that they capture all interactions with other threads in the system. We therefore assign sampling sets to avoid overlapping with each other. That is, process 0 samples in sets 0 to 3, process 1 in sets 4 to 7, and so on, as shown in Fig. 12c. This is implemented by determining the sampling sets from the local core's active process id.

Choosing a degree: Counters then "vote" on the best replication degree. Positive values of Δ_a^b indicate that a is better than b, and negative values the opposite. If Δ_a^b's value exceeds half the maximum (i.e., $\Delta_a^b > 2^{16}$), then it votes for degree a; otherwise, if less than half the minimum, then it votes for b. This mechanism requires just a simple combinational circuit that takes MSBs of counters to 4 AND gates and votes with a 4-1 decoder. If any degree achieves consensus (3 votes with 4 supported degrees), it becomes the active replication degree.
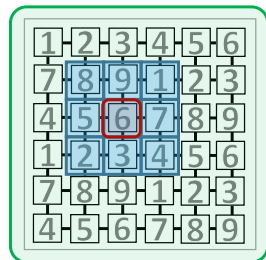
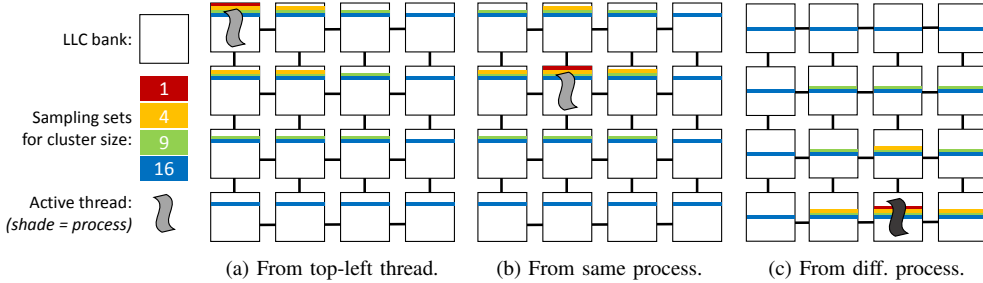(a) From top-left thread.   (b) From same process.   (c) From diff. process.

Figure 12: Set sampling of different replication degrees. Cores spread accesses to read-only data across nearby banks and monitor their latency. Cores access different banks, processes access non-overlapping sampling sets.

Figure 13: Core sampling access to degree 4. Δ-counters record latency difference between two degrees.

Changing degrees: Read-only data does not require coherence, so no special coherence actions are required when the replication degree changes. LLC inclusion is preserved: every sharer in a private cache is tracked by some LLC directory—even if it is not the LLC directory that the private cache is currently mapped to. This requires no coherence changes if the protocol performs silent drops, a common optimization (private caches evict clean lines without notifying the LLC directory, so they will not evict to the wrong directory when replication degree changes). This means that hardware can change degrees whenever counters indicate it is beneficial.

Finally, since adaptive replication degree increases the number of banks where data may reside, the OS needs to invalidate replicas in more locations when pages transition from read-only to read-write shared.

D. Coordinated replication degree

The simplest implementation of Nexus-R would let each core choose its replication degree independently, but this performs poorly—sacrificing half of the possible gains (Sec. VI-E). The problem is that there is systematic pressure towards more replication: Each core is selfish, wanting its neighbors to replicate as little as possible and itself to replicate as much as possible. When a core chooses a higher replication degree, it places additional pressure on the nearby banks where its data is replicated. This can make its neighbors' replicated data no longer fit, reducing the benefits of replication. Neighboring cores thus prefer to replicate more aggressively: since their data no longer fits, the best policy is to *miss quickly* by replicating in nearby banks.

Hence, there is a tendency towards full replication, even on applications where the read-only footprint does not fit. This is a classic coordination problem. A simple solution is to coordinate decisions among actors, i.e., all threads in the same process. Since processes are an OS-level construct, we leverage the OS to coordinate replication degrees.

OS support: We achieve this by exposing the latency-difference counters as part of the thread context. The OS then tracks *per-process* latency differences that are used to decide the replication degree for the entire process.

The OS is responsible for initializing a core's local counters to the process counter values upon thread migration, creation,

or context switch. The OS also records this value (e.g., on the stack). Then, on each OS scheduler tick or context switch, the OS computes the difference between the current value and the last recorded value. It uses this *second-order difference* to update the per-process difference counters. Since changes to the best degree are rare, the OS tick rate is frequent enough.

E. Overheads

Nexus-R introduces small overheads on top of R-NUCA. We find that, with 32-way caches, a single sampling set per degree suffices. (With a single multi-threaded process, the total number of sampling sets per degree equals the number of cores.) Using 512 KB banks, 1.5% of accesses are sampled, and only three-quarters of these (1.1%) use the "wrong" degree. Counters add $17 \times 6 = 102$ bits per core, and the combinational logic for indexing banks and voting adds small overheads. The OS support is tens of instructions per context switch, a small addition to R-NUCA's existing support for page reclassification and thread migration.

F. Related work

Qureshi et al. [40] originally proposed set dueling, which been widely applied in caching techniques. Our design is similar to TADIP-F [24] in that each process monitors its behavior against a background of the decisions of other processes. However, it differs in that *(i)* it chooses among 4 options for each process, rather than a binary choice, *(ii)* sampling plays out across several banks, rather than within a single bank, and *(iii)* decisions must be coordinated, rather than independent.

V. NEXUS ON JIGSAW (NEXUS-J)

Nexus-R works well for a single multithreaded application. However, prior work [2, 6, 7, 32] has shown that, in multi-programmed workloads, managing capacity among applications is critical to improve system throughput and fairness. Nexus-J extends Jigsaw [6, 7], which already manages LLC capacity, to support adaptive replication of read-only data. Nexus-J adds a few extra hardware monitors to those already used in Jigsaw, and enhances the Jigsaw software runtime to choose the best replication degree during capacity allocation. Nexus-J thus handles more complicated workloads with modest added complexity.
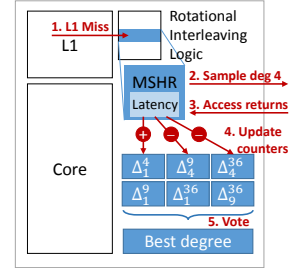
A. Background: Jigsaw

Jigsaw is a partitioned, directory-less D-NUCA. Jigsaw builds *virtual caches* (VCs) by combining partitions of physical cache banks, as shown in Fig. 14 (colors represent different VCs). Pages are mapped to a specific VC through the TLB, as in R-NUCA, but Jigsaw adapts the placement of VCs by adding a further layer of indirection. Jigsaw uses three types of VCs: thread-private, process-shared, and globally-shared. Pages start as private to the thread that allocates them, and are upgraded lazily: e.g., an access from another thread upgrades the page to the process VC, and an access from another process upgrades the page to the global VC.
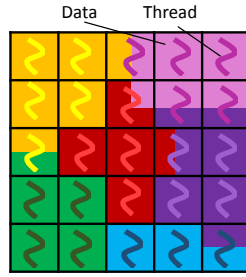

Figure 14: Jigsaw divides LLC banks into *virtual caches* (VCs).

Jigsaw has hardware and software components. In hardware, Jigsaw augments the TLB with the page's VC id and adds a small structure to each core, called the *VC translation buffer (VTB)*, that maps each VC's accesses to its allocated banks. (Since each thread accesses only three VCs, the VTB needs only three entries.)

Jigsaw also uses hardware monitors (GMONs [7]) to produces *miss rate curves* for each VC (i.e., the number of expected misses at different cache sizes). Other than monitors, the hardware also needs to expose basic information (e.g., the number of cores and the network topology) to the software so that Jigsaw's runtime can optimize VC configuration by modeling *latency curves*.

In software, an OS runtime periodically (e.g., every 50 ms) reallocates LLC capacity among VCs to minimize data movement. Jigsaw's runtime models the expected total access latency of each VC at different sizes (Fig. 15), including network traversals (on-chip latency, from topology information) and cache misses (off-chip latency, from GMONs). It then allocates capacity among VCs and places them across LLC banks while trying to minimize total system latency (see prior work for further details [6, 7]). The runtime essentially takes latency curves as the input and produces VC configuration as the output. Jigsaw does not replicate data.
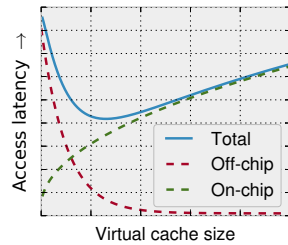

Figure 15: Access latency.

B. Supporting replication in Nexus-J

Nexus-J extends Jigsaw to create additional *read-only* VCs for each process. Pages transition between VCs similar to Nexus-R: they start as thread-private, but a read from another thread upgrades the page to the read-only VC first, and writes update it to the (read-write) process VC.

Nexus-J supports replication by creating multiple read-only VCs and directing accesses from different *core groups* to different replicas. Nexus-J adds another entry for read-only VCs to each core's VTB and configures them to use the closest replica. For example, at degree 4, Nexus-J assigns one read-only VC replica to each chip quadrant. Hence, threads running in different quadrants will access different read-only VCs, but the same read-write VC (see Fig. 16).
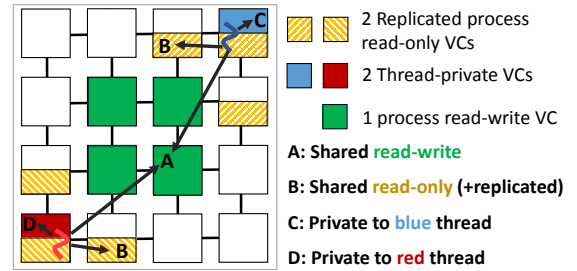

Figure 16: Nexus-J snapshot with 2 threads running on a 16-core system. Shared read-only data (e.g., B) is replicated, but shared read-write data (e.g., A) is not.

When a read-only page is upgraded to read-write, Nexus-J invalidates the page in every read-only VC to maintain coherence. Finally, Nexus-J monitors each core group separately, letting it capture heterogeneous behavior across groups, unlike Nexus-R. This helps on some apps (e.g., freqmine).

C. Selecting replication degree in Nexus-J

One of the main benefits of applying Nexus to Jigsaw is that we can leverage its existing optimization runtime. Jigsaw uses each VC's latency curve to allocate capacity among VCs. Nexus-J changes these latency curves so that Jigsaw's runtime automatically chooses the best replication degree.

Latency curves: First, Nexus-J produces the latency curve for each read-only VC by combining curves for each replication degree [36, §B], as illustrated in Fig. 17. For each replication degree, Nexus-J computes the latency curve for read-only data from each corresponding core group. The result is intuitive: increasing the replication degree achieves lower latency, but takes more capacity (e.g., compare degrees 1 and 36). Nexus-J then combines these curves by taking the minimum (Fig. 17b).
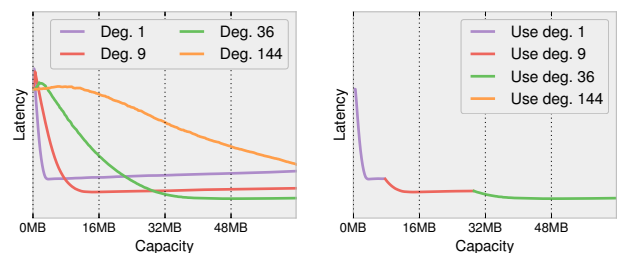
(a) Curves for each degree.   (b) Combined curve.

Figure 17: Nexus-J produces latency curves for read-only VCs by taking the minimum latency across all supported degrees.

Selecting the replication degree: The combined curve serves two purposes: for every VC size, it encodes both the best achievable latency and the corresponding replication degree (color in Fig. 17b). Hence, when Jigsaw's optimization runtime chooses the read-only VC's size, it is also implicitly choosing its replication degree. For example, if the read-only VC in Fig. 17 is allocated 48 MB, that means the best replication degree is 36 (i.e., it is green at 48 MB in Fig. 17b).

Nexus-J therefore adds a small step to Jigsaw's optimization runtime after sizing VCs that: finds each read-only VC's replication degree (from its combined latency curve), creates that many VCs (one per core group), and allocates the capacity among the created VCs (using per-group latency curves). Jigsaw's placement step then proceeds normally.

D. Overheads

Nexus-J adds small overheads in hardware and software. Nexus-J adds one VC accessible from each core with an extra entry in the VTB, adding 272 B per core (0.05% of the local bank). Nexus-J monitors each core group for each replication degree. With 144 cores and 4 supported replication degrees, this gives amortized overheads of $1 + 1/4 + 1/36 + 1/144$ monitors per core, 4.2 KB per tile (0.8% of LLC capacity). Software overheads are small: each reconfiguration takes 50 M cycles, less than 0.4% of system cycles with reconfigurations every 50 ms.

E. Related work

Like Nexus-J, Jenga [47] leverages Jigsaw's optimization runtime by modifying its input latency curves. However, these systems target different system parameters: Jenga configures the depth and configuration of the cache hierarchy, whereas Nexus-J configures replication degree. We will consider simultaneously optimizing hierarchy, replication, and thread placement [7] in future work.

VI. EVALUATION

We now evaluate Nexus against state-of-the-art D-NUCAs to demonstrate the benefits of adaptive replication in directory-less D-NUCAs (Sec. VI-B). We show that adapting replication degree is necessary to fully exploit the latency-capacity tradeoff, and that simply extending R-NUCA and Jigsaw to replicate read-only data with fixed degrees does not yield the same benefits as Nexus (Sec. VI-C).

A. Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [42], and model a 144-core tiled CMP with a 12×12 mesh network. Each tile has one lean 2-way OOO core similar to Silvermont [28] with private L1 caches and an LLC bank. We configure the system to match commercial many-core chips (e.g., Knights Landing [45] and TILE-Gx [46]) by using a mesh NoC and a shallow cache hierarchy. Table II details the system's configuration, which

TABLE II: CONFIGURATION OF THE SIMULATED 144-CORE CMP.

Cores	144 cores, x86-64 ISA, 2 GHz, Silvermont-like OOO [28]: 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
Prefetchers	16-entry stream prefetchers modeled after and validated against Nehalem [18, 42]
Coherence	MESI, 64 B lines; sequential consistency
Global NoC	12×12 mesh, 128-bit flits and links, X-Y routing, 1-cycle pipelined routers, 1-cycle links
L2 caches	72 MB, 512 KB bank per tile, 32-way set-associative cache, 9-cycle bank latency, LRU replacement
Main memory	8 MCUs, 1 channel/MCU, 120 cycles zero-load latency, 19.2GB/s per channel (DDR4-2400)

is also similar to prior work in adaptive replication [19, 31]. Sec. VI-E studies the impact of different system parameters.
LLC schemes: Our baseline is a S-NUCA LLC. We compare five schemes against the baseline. *(i)* R-NUCA [19], which only replicates instructions. *(ii)* Jigsaw [6, 7], which never replicates data. *(iii)* Locality-aware replication (LAR [31]), a state-of-art, directory-based adaptive replication scheme. LAR has a complete classifier (K=144) and uses the reported best replication threshold (RT=3). LAR also uses R-NUCA's private/shared classification to place private data in local banks. Finally, we evaluate *(iv)* Nexus-R and *(v)* Nexus-J as described earlier.
Metrics: Since IPC is not a valid measure of work in multithreaded workloads [1], to perform a fixed amount of work we instrument each app with heartbeats that report global progress (e.g., when each timestep or transaction finishes) and run each app for as many heartbeats as S-NUCA completes in 2 B cycles after the serial region.

We report speedup over S-NUCA and dynamic data movement energy breakdown. To achieve statistically significant results, we perform enough runs to achieve 95% confidence intervals ≤1%. We use McPAT 1.1 [33] to derive the energy numbers of chip components (cores, caches, NoC, and memory controller) at 22 nm and Micron datasheets [35] for memory. We report energy consumed to perform a fixed amount of work. We focus on dynamic data movement energy, as this is the part of system energy affected by LLC scheme (static energy is affected by performance, which is evaluated separately). We present total system energy in text; overall, dynamic data movement energy consumes 25% of total S-NUCA system energy in our workloads.
Workloads: We simulate the 60 multithreaded benchmarks from five diverse suites, using their medium and large input sets: scientific workloads from SPECOMP2012, PARSEC [8], SPLASH-2 [48], and BioParallel [25], and server workloads from TailBench [29].[1] Fig. 18 plots the performance of each workload, comparing Nexus-R's improvement over R-NUCA

[1] These include all applications in these suites except the three PARSEC pipeline-parallel benchmarks (which do not work in our infrastructure), and `choleksy` and `radiosity` from SPLASH-2 due to short execution times.

on the *y*-axis and LAR's improvement over R-NUCA on the *x*-axis. Nexus-R outperforms LAR on most of these workloads (i.e., those above the dashed line in Fig. 18).
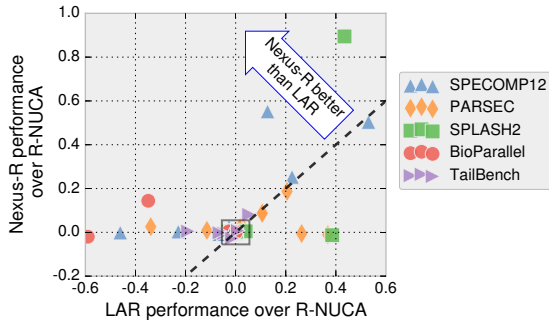

Figure 18: Performance of Nexus-R and LAR over R-NUCA on the 60 apps in SPECOMP12, PARSEC, SPLASH2, BioParallel, and TailBench.

We focus our evaluation on the 20 *replication-sensitive* workloads from this full set, i.e., those workloads with at least 5% difference in performance between Nexus, LAR, and R-NUCA (shown outside the box in Fig. 18). Table III details these 20 applications and their input sets.

TABLE III: REPLICATION-SENSITIVE WORKLOADS AND INPUTS USED.

Suite	Benchmark and input
SPECOMP2012	botsspar, kdtree (ref/train) bt331, imagick (train)
PARSEC	canneal, freqmine, swaptions (native/simlarge) streamcluster (simlarge)
SPLASH2	barnes (1M particle), raytrace (balls4)
BioParallel	svm (253/30 cases)
TailBench	SpecJBB (TPC-C 144 warehouses) Shore (TPC-C 10 warehouses) Masstree (mycsb-a)

B. Nexus outperforms prior D-NUCAs

Fig. 19 shows the performance and dynamic data movement energy of the different LLC organizations. We normalize all results to S-NUCA. Nexus-R and Nexus-J both outperform prior schemes, showing the wide applicability of adaptive replication in directory-less D-NUCAs.

We classify applications into four categories: those that prefer *(i)* high, *(ii)* medium, and *(iii)* low replication degrees, and those where *(iv)* LAR is better than Nexus-R.

For the first group (7 out of 20 workloads), both Nexus and LAR outperform R-NUCA and Jigsaw (Fig. 19a). These workloads have small read-only data footprints that fit in the local bank. Therefore, full replication is best. Both Nexus and LAR achieve this, improving performance of applications by eliminating on-chip network traversals (Fig. 19b).

However, when the read-only data footprint becomes larger, as in the second group (4 out of 20 workloads), replicating in the local bank is suboptimal. In these workloads (e.g.,

btspa-r and raytrace), the read-only data does not fit in the local bank, but fits in a cluster of banks. LAR, which selects between replicating in the local bank or not, causes more off-chip misses than Nexus. In `svm-s`, these extra misses make LAR slower than even S-NUCA. Nexus significantly outperforms LAR by choosing intermediate degrees and letting cores share replicas. Nexus thus finds the best latency-capacity tradeoff for these applications.

When the footprint is even larger, as in the third group (6 out of 20 workloads), LAR performs worse than S-NUCA because excessive replication adds unnecessary misses, while Nexus chooses not to replicate and avoids these misses.

For the last group (3 out of 20 workloads), LAR performs better than Nexus-R. `barnes` has infrequently written data, and Nexus's one-way classification cannot capture the read-only phases between writes. We study this benchmark in detail later (Sec. VI-E). For `freqmine`, when using 144 threads, its performance is determined by a single, dominant thread that accesses shared read-write data. Since R-NUCA and Nexus-R treat all threads equally, they cannot improve the performance of this thread. In contrast, Jigsaw and Nexus-J outperform LAR by placing shared read-write data closer to this thread, and Nexus-J outperforms Jigsaw by distinguishing between read-only and read-write data.

Overall, for replication-sensitive workloads, Nexus-J/R significantly outperform LAR, by 20%/16% on average and by up to 2.8× (`svm-l`). Nexus-J/R improve performance over S-NUCA by 23%/18% on average, while prior work outperforms S-NUCA by less than 10%. Nexus also achieves the greatest savings in dynamic data movement energy (40%) and full system energy (15%). Over all 60 workloads, Nexus-J/R improve performance over S-NUCA by 11/9% on average, while others achieve less than 5%.

The original LAR paper did not find these pathologies because it did not evaluate applications with large read-only data footprints, such as `svm` and `canneal`. To correct these pathologies, we sweep the replication threshold (RT) at 1, 3, 8, 20, 50, 150, 500, and 2000, and use the threshold that achieves the highest gmean speedup for our workloads, $RT = 150$. With this threshold, LAR achieves only 10% speedup over S-NUCA, and Nexus-J/R still outperform LAR by 12%/7% on average.

Finally, Nexus-R outperforms Nexus-J in some workloads, such as `raytrace` and `botsspar-r`. This is due to rotational interleaving, which further reduces the access latency in R-NUCA. With more extensive changes to Jigsaw, a similar technique could provide similar benefits in Nexus-J.

C. Adaptive replication is essential

To show that Nexus's benefits come from adapting replication degree, we compare Nexus-R and -J to their unmodified baseline schemes (i.e., R-NUCA and Jigsaw) and against themselves replicating read-only data but at fixed degrees. Fig. 20 shows the performance improvement over S-NUCA.

(a) Speedup.

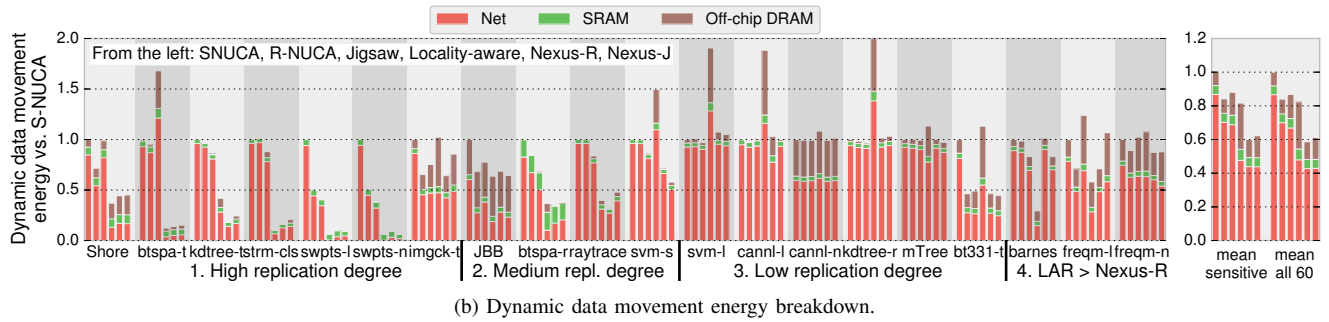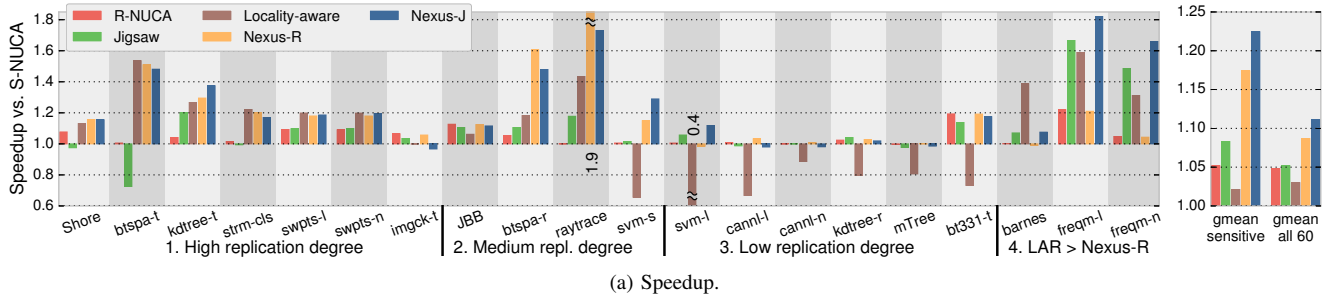(b) Dynamic data movement energy breakdown.

Figure 19: Simulation results for server and HPC applications on several NUCA schemes and Nexus.

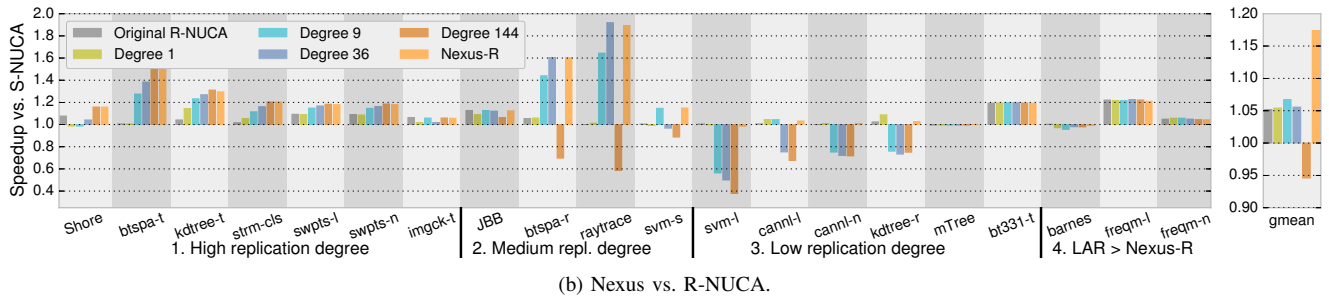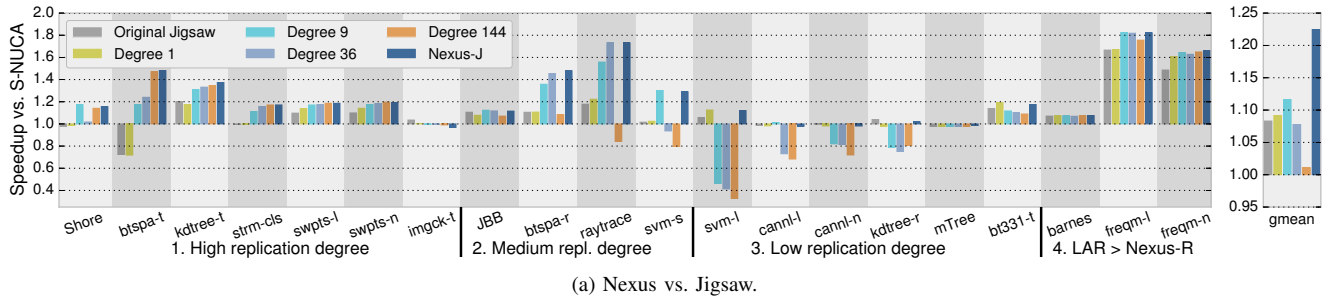(a) Nexus vs. Jigsaw.

(b) Nexus vs. R-NUCA.

Figure 20: Performance of Nexus-J/R vs. R-NUCA and Jigsaw using fixed replication degrees.

As we saw in Sec. III, applications prefer widely different replication degrees. `shore`, `streamcluster`, `kdtree-t`, and `botsspar-s` have a small read-only footprint and little cache pressure from private or read-write data, so replication degree 144 (full replication) is best. For JBB, `botsspar-l`, `raytrace`, the best replication degree is 36 to balance on-chip and off-chip latencies. And for `svm-s`, the best replication degree is 9. Nexus-J/R find the right replication degree on all applications, matching the performance of the best static choice.

These results show that *Nexus's improvements come from adapting replication degree, not other changes*. Any fixed replication degree gives little to no average improvement over the baseline. Compare the gmean bars on the right of Fig. 20: the leftmost bar shows the baseline, the next four bars show Nexus with fixed degree, and the rightmost bar shows Nexus with adaptive degree. Overall, no single fixed replication degree works well, whereas Nexus-J/R improve gmean performance over the best fixed degree by 11%.
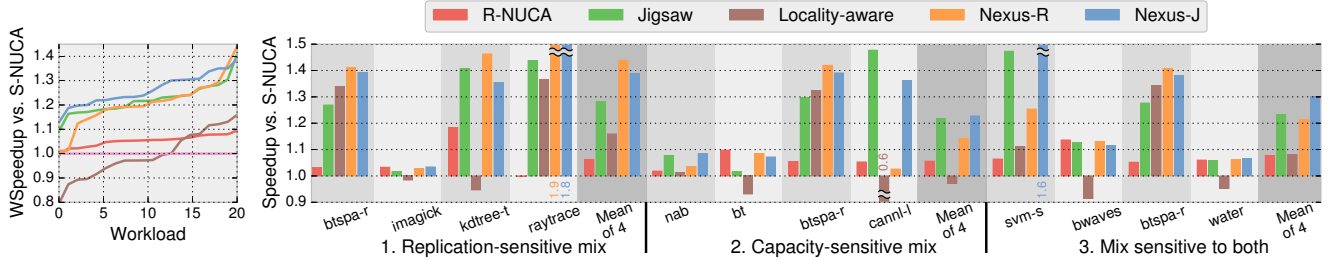
Figure 21: Performance on multi-programed workloads. Left: Distribution of weighted speedups over 20 mixes. Right: Per-application speedups of 3 mixes.

D. Nexus-J outperforms other schemes in multi-programmed workloads

Next, we evaluate Nexus with 20 multi-programmed workloads. Each workload is a mix of four 36-thread apps that are randomly selected from our benchmark suites. Applications are clustered in quadrants of the 144-core chip. We use weighted speedup as the performance metric, which accounts for throughput and fairness [39, 44].

The left of Fig. 21 shows the distribution of weighted speedups over S-NUCA for the schemes we consider. Each line shows the performance over all 20 mixes for a single scheme, sorted from worst to best along the x-axis. Overall, gmean weighted speedups are 26% for Nexus-J, 21% for Jigsaw, 20% for Nexus-R, 5% for R-NUCA, and 1% for LAR. Hence, even without replication, Jigsaw outperforms LAR and Nexus-R, which perform adaptive replication.

To better understand these results, Fig. 21 presents 3 representative mixes and shows the performance gain for each program in the mix. In the first mix, where most apps benefit from replication, Nexus-J and -R significantly improve performance by replicating read-only data. Jigsaw improves performance by placing shared read-write data in the middle of each quadrant instead of spreading it across the chip. LAR helps some apps, but not all, because it does not consider how apps interfere with each other in the LLC. R-NUCA gets no improvement since it spreads all shared data (including read-only) across the chip, like S-NUCA. For this mix, Nexus-J and -R improve weighted speedup by 40%/43%, while Jigsaw, LAR, and R-NUCA only improve it by 29%, 17%, and 5%.

In the second mix, where apps are less sensitive to replication, Nexus-J and Jigsaw improve performance the most by carefully allocating capacity. Nexus-R also improves performance, but only `botsspar-r` benefits from replication. LAR also improves `botsspar-r`, but by sacrificing `canneal-l`'s performance. R-NUCA again performs similarly to S-NUCA. For this mix, Nexus-J and Jigsaw improve weighted speedup by 22% and 21%, but Nexus-R and R-NUCA only improve it by 15% and 6%, and LAR hurts it by 2%.

In the third mix, which contains both replication-sensitive and capacity-sensitive apps, Nexus-R improves performance by 21% through adaptive replication, Jigsaw by 23% through careful capacity allocation, and Nexus-J by 30%—the highest speedup—by combining both techniques.

In summary, when several apps run concurrently, they compete for LLC capacity, complicating the tradeoffs in adaptive replication. This makes Nexus-J attractive, since its software runtime carefully weighs the latency-capacity tradeoffs when deciding how to allocate capacity and how much to replicate.

E. Nexus sensitivity studies

Sensitivity to system size: We evaluate Nexus on systems with different core counts and network topologies. Fig. 22a shows the performance improvement of NUCA schemes at different system sizes. Nexus's benefits increase as the diameter of the NoC increases, from gmean 12% vs. S-NUCA for an 8×8 mesh to 17% for a 12×12 mesh. Also, we evaluate Nexus on a latency-optimized 12×12 mesh with 2-cycle express links that connect tiles four hops away. Nexus-J/R still improve gmean performance by 18%/14%.

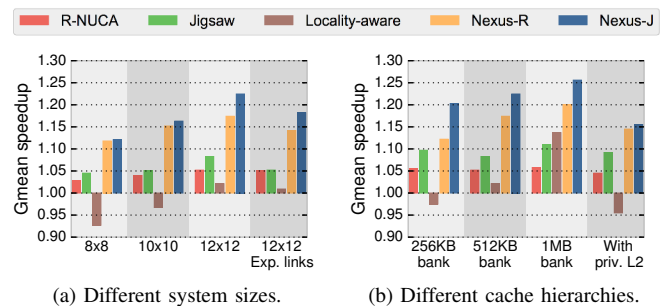(a) Different system sizes. (b) Different cache hierarchies.

Figure 22: Sensitivity study of performance of NUCA schemes with various system parameters.

Sensitivity to different cache hierarchies: Fig. 22b shows the performance of different schemes using LLC banks with half (128 KB) or twice (1 MB) their original capacity, and when using a 128KB, 6-cycle private L2 in each tile.

Smaller LLC banks make capacity scarcer, so replication is less beneficial (Sec. III). In this case, Nexus-R and Nexus-J replicate less, but still improve performance by 12% and 20%. Conversely, larger banks make replication more beneficial. LAR, Nexus-R, and Nexus-J improve performance further with 1MB banks, by 14%, 20%, and 26%.

With private L2s, a small amount of read-only data is replicated locally. Therefore, the performance advantage of replication is reduced. With private L2s, Nexus-R/J improves

performance by 9/6% over R-NUCA and Jigsaw, and by 15% over S-NUCA. LAR performs worse since private L2s capture most of the benefit of replication in the local bank.

In summary, these experiments show that Nexus offers consistent benefits across different system parameters.

Dynamic reclassification for `barnes`: In one benchmark, `barnes` from SPLASH-2, shared data is read without being written for long phases, but is written infrequently. Fig. 23 shows the sharing behavior (thread-private, shared read-only, shared read-write) of different pages in `barnes` over time. More than 50% of the pages are *intermittently read-write*— these pages are written in some phases, but read-only in others. LAR performs well in this benchmark, with 40% improvement over S-NUCA, while Nexus only achieves 8% improvement due to its one-way classification.
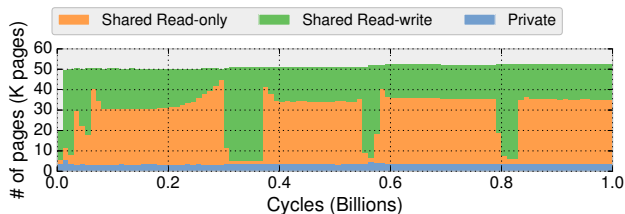

Figure 23: Trace showing page usage in `barnes` over time. Many pages are temporarily read-write but read-only otherwise.

This issue can be addressed by dynamically reclassifying pages [17, 41]. Nexus with an idealized page reclassification technique [41], which makes page classification periodically "decay", increases Nexus's performance improvement on `barnes` to 37%. However, this pathology occurs in just one out of sixty evaluated workloads.

Uncoordinated Nexus-R often performs poorly: In Nexus-R, all threads in a process agree upon a single replication degree. To study the impact of this coordination, we evaluate Nexus-R without this support: cores make local decisions based on local latency counters. Performance degrades on five apps (`svm-l`, `canneal-l`, `kdtree-t`, `raytrace`, and `swaptions`), and gmean performance decreases by 7%.

VII. Conclusion and Future Work

Data replication significantly improves performance and efficiency in systems with distributed caches. Unlike prior adaptive replication techniques, Nexus adapts *how much* to replicate data, not *which* data to replicate. To achieve this, Nexus builds on recent, directory-less D-NUCAs that allow cores to share replicas across the chip. We have presented two implementations of this idea, Nexus-R and Nexus-J, each of which add small overheads and significantly outperform the state-of-the-art adaptive replication scheme.

One question left unresolved is how to combine Nexus with selective replication, i.e., how to choose both *how much* and *which* data to replicate. Currently, Nexus uses a single replication degree for all read-only data, but adapting replication degree to different data could be more beneficial. We believe this is an interesting and important direction for future work in distributed caches.

Appendix: A Simple Analytical Latency Model for Data Replication

Fig. 3 uses a simple analytical cache model to calculate the average access latency of different replication schemes:

$$\text{Latency} = \text{Hit latency} + (1 - \text{Hit ratio}) \times \text{Miss penalty}$$

We use ℓ to denote latencies, c to denote cache capacity, and s for the application's working set size. We assume an optimal replacement policy, i.e., the hit ratio is:

$$h(c,s) = \min\{c/s, 1\}$$

Therefore, the latency for full replication is:

$$\text{Latency}_{\text{Full}} = \ell_{\text{Bank}} + (1 - h(c_{\text{Bank}}, s)) \times \ell_{\text{Memory}}$$

For no replication, it is similar, but with different parameters:

$$\text{Latency}_{\text{None}} = \ell_{\text{LLC}} + (1 - h(c_{\text{LLC}}, s)) \times \ell_{\text{Memory}}$$

Selective replication checks the local bank first, then the whole LLC, and finally memory if it misses in both. This policy essentially combines the previous two and creates a two-level cache hierarchy. We optimistically assume that the full LLC capacity is available to both replicated and non-replicated data:

$$\text{Latency}_{\text{Selective}} = \ell_{\text{Bank}} + (1 - h(c_{\text{Bank}}, s)) \times \ell_{\text{LLC}}$$
$$+ (1 - h(c_{\text{LLC}}, s)) \times \ell_{\text{Memory}}$$

Finally, Nexus uses the optimal replication degree $d = c_{\text{LLC}}/s$ and replicates data d times across the chip. Each core accesses a replica in the closest cluster, which has size c_{LLC}/d. The latency of this cluster consists of the bank latency plus the average network latency, which is determined by the number of banks in the cluster and the network topology. On a mesh, the network latency grows with the square root of the size of the cluster. Also, Nexus's hit ratio equals that of a cache without replication, since Nexus will not replicate uniformly accessed data when it does not fit in the LLC. Nexus's latency is:

$$\text{Latency}_{\text{Nexus}} = \ell_{\text{Cluster}}(d) + (1 - h(c_{\text{LLC}}, s)) \times \ell_{\text{Memory}}$$

REFERENCES

[1] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[2] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. HPCA-15*, 2009.

[3] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of SPLASH-2 and PARSEC," in *Proc. IISWC*, 2009.

[4] B. Beckmann, M. Marty, and D. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.

[5] B. Beckmann and D. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proc. ASPLOS-XI*, 2004.

[6] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.

[7] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *Proc. HPCA-21*, 2015.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. PACT-17*, 2008.

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

[10] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *Proc. HPCA-15*, 2009.

[11] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *Proc. ISCA-32*, 2005.

[12] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. MICRO-39*, 2006.

[13] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *Proc. EuroSys*, 2013.

[14] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proc. ISCA-38*, 2011.

[15] W. J. Dally, "GPU computing: To exascale and beyond," *SC10 Keynote*, 2010.

[16] H. Dybdahl and P. Stenstrom, "An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors," in *Proc. HPCA-13*, 2007.

[17] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "TokenTLB: A token-based page classification approach," in *Proc. ICS'16*, 2016.

[18] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, 1992.

[19] 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.

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

[21] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors," in *Proc. ISCA-37*, 2010.

[22] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence protocol optimization for both private and shared data," in *Proc. PACT-20*, 2011.

[23] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S. Keckler, "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Par. Dist. Sys.*, vol. 18, no. 8, 2007.

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

[25] A. Jaleel, M. Mattina, and B. Jacob, "Last level cache (LLC) performance of data mining workloads on a CMP," in *Proc. HPCA-12*, 2006.

[26] L. Jin and S. Cho, "SOS: A software-oriented distributed shared cache management approach for chip multiprocessors," in *Proc. PACT-18*, 2009.

[27] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proc. ISCA-42*, 2015.

[28] D. Kanter, "Silvermont, Intel's low power architecture," *Real World Tech*, 2013.

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

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

[31] G. Kurian, S. Devadas, and O. Khan, "Locality-aware data replication in the last-level cache," in *Proc. HPCA-20*, 2014.

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

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO-42*, 2009.

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

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

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

[37] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked DRAM," in *Proc. PACT-24*, 2015.

[38] M. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proc. HPCA-15*, 2009.

[39] 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.

[40] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. ISCA-34*, 2007.

[41] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *Proc. ICPP-42*, 2013.

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

[43] A. Seznec, "Bank-interleaved cache or memory indexing does not require euclidean division," in *WDDD-11*, 2015.

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

[45] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, 2016.

[46] Tilera, "TILE-Gx 3000 Series Overview," Tech. Rep., 2011.

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

[48] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. ISCA-22*, 1995.

[49] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proc. ISCA-32*, 2005.