Computer Architecture Lab at Carnegie Mellon Technical Report CALCM-TR-2008-001
Submitted to ISCA 2009

R-NUCA: Data Placement in Distributed Shared Caches

Nikos Hardavellas[1], Michael Ferdman[1,2], Babak Falsafi[1,2] and Anastasia Ailamaki[3,1]

[1]Computer Architecture Lab (CALCM), Carnegie Mellon University, Pittsburgh, PA, USA
[2]Parallel Systems Architecture Lab (PARSA), École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[3]École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Abstract

Increases in on-chip communication delay and the large working sets of commercial and scientific workloads complicate the design of the on-chip last-level cache for multicore processors. The large working sets favor a shared cache design that maximizes the aggregate cache capacity and minimizes off-chip memory requests. At the same time, the growing on-chip communication delay favors core-private caches that replicate data to minimize delays on global wires. Recent hybrid proposals offer lower average latency than conventional designs. However, they either address the placement requirements of only a subset of the data accessed by the application, require complicated lookup and coherence mechanisms that increase latency, or fail to scale to high core counts.

In this work, we observe that the cache access patterns of a range of server and scientific workloads can be classified into distinct categories, where each class is amenable to different data placement policies. Based on this observation, we propose Reactive NUCA (R-NUCA), a distributed shared cache design which reacts to the class of each cache access and places blocks at the appropriate location in the cache. Our design cooperates with the operating system to support intelligent placement, migration, and replication without the overhead of an explicit coherence mechanism for the on-chip last-level cache. We evaluate R-NUCA on a range of server, scientific and multi-programmed workloads and find that its performance matches the best alternative design, providing a speedup of 17% on average against the competing alternative, and up to 26% at best.

1 Introduction

In recent years, processor manufacturers have shifted towards producing multicore processors to remain within the power and cooling constraints of modern chips while maintaining the expected performance advances with each new processor generation. Increasing device density enables exponentially more cores on a single die. Major manufacturers already ship 8-core chip multiprocessors [25] with plans to scale to 100s of cores [1, 32], while specialized vendors already push the envelope further (e.g., Cisco's CRS-1 with 192 Tensilica network-processing cores, Azul's Vega 3 with 54 out-of-order cores). The exponential increase in the number of cores results in the commensurate increase in the on-chip cache size required to supply these cores with data. Physical and manufacturing considerations suggest that future processors will be tiled, where groups of processor cores and banks of on-chip cache will be physically distributed throughout the chip area [1,43]. Tiled architectures give rise to varying access latencies between the cores and the cache slices spread around the die, naturally leading to a Non-Uniform Cache Access (NUCA) organization of the on-chip last-level cache (LLC), where the latency of a cache hit depends on the physical distance between the requesting core and the location of the cached data.

However, growing cache capacity comes at the cost of access latency. As a result, modern workloads already spend most of their execution time on on-chip cache accesses. Recent research shows that server workloads lose as much as half of the potential performance due to the high latency of on-chip cache hits [20]. Although increasing device switching speeds result in faster cache-bank access times, communication delay remains constant across technologies [8], and access latency of far away cache slices becomes dominated by wire delays and on-chip communication [24]. Thus, from an access-latency perspective, an LLC organization where each core treats a nearby LLC slice as a private cache is desirable.

While a private distributed LLC organization results in fast local hits, it requires area-intensive, slow and complex mechanisms to guarantee coherence. In turn, coherence mechanisms reduce the available cache area and penalize data sharing, which is prevalent in many multicore workloads [3,20]. At the same time, growing application working sets render private caching schemes impractical due inefficient use of cache capacity, as cache blocks are replicated between private cache slices and waste space. At the other extreme, a shared distributed LLC organization where blocks are statically address-interleaved in the aggregate cache offers maximum capacity by ensuring that no two cache frames are used to store the same block. Because static interleaving defines a single, fixed location for each block, a shared distributed LLC does not require a coherence mechanism, enabling a simple LLC design and allowing for larger aggregate cache capacity. However, static interleaving results in a random distribution of cache blocks across the L2 slices, leading to frequent accesses to distant cache slices and high access latency.

An ideal LLC organization enables the fast access times of the private LLC and the design simplicity and large capacity of the shared LLC. Recent research advocates hybrid and adaptive designs to bridge the gap between private and shared organizations. However, prior proposals require complex, area-intensive, and high-latency lookup and coherence mechanisms [4, 10, 7, 43], waste cache capacity [4, 43], do not scale to high core counts [7, 19], or optimize only for a subset of cache accesses [4, 7, 11].

In this paper we propose Reactive NUCA (R-NUCA), a scalable, low-overhead and low-complexity cache architecture that optimizes data placement for all cache accesses, while at the same time attaining the fast local access of the private organization and the large aggregate capacity of the shared scheme.

R-NUCA cooperates with the operating system to classify accesses at the page granularity, achieving negligible hardware overhead and avoiding complex heuristics that are prone to error, oscillation, or slow convergence [4, 10, 7]. The placement decisions in R-NUCA guarantee that each modifiable block is mapped to a single location in the aggregate cache, thereby obviating the need for complex, area- and power-intensive coherence mechanisms that are commonplace in other proposals [7, 4, 10, 43]. At the same time, R-NUCA allows read-only blocks to be shared by neighboring cores and replicated at distant ones, ensuring low access latency for surrounding cores while balancing capacity constraints. In the process of doing so, R-NUCA utilizes rotational interleaving, a novel lookup mechanism that matches the fast speed of address-interleaved lookup without pinning the block to a single location in the cache, thereby allowing the block's replication while avoiding expensive lookup operations [43, 10].

More specifically, in this paper we make the following contributions:

- Through execution trace analysis, we show that cache accesses for instructions, private data, and shared data exhibit distinct characteristics leading to different replication, migration, and placement policies.
- We leverage the characteristics of each access class to design R-NUCA, a novel, low-overhead, low-latency mechanism for data placement in the NUCA cache of large-scale multicore chips.
- We propose rotational interleaving, a novel mechanism to allow fast lookup of nearest-neighbor caches, eliminating multiple cache probes and allowing replication without wasted space and without coherence overheads.
- We use full-system cycle-accurate simulation of multicore systems to evaluate R-NUCA and find that its performance always exceeds the best of private or shared design for each workload, attaining a speedup of 20% on average against the competing alternative when running server workloads (17% on average when including scientific and multi-programmed workloads) and up to 26% speedup at best.

The rest of this paper is organized as follows. Section 2 presents background on distributed caches and tiled architectures, and provides insight into the data-access patterns of modern workloads. Section 3 presents our classification and offers a detailed empirical analysis of the cache-access classes of commercial, scientific, and multi-programmed workloads. We detail the R-NUCA design in Section 4 and evaluate it in Section 5 using cycle-accurate full-system simulation. We summarize prior work in this area in Section 6 and conclude in Section 7.

While our techniques are applicable to any last-level cache, we assume a 2-level cache hierarchy in our evaluation. Thus, in the interest of clarity, we refer to our last-level cache as L2 in the remainder of this work.

2 Background

2.1 Non-Uniform Cache Architectures

Growing wire delays have necessitated a departure from conventional cache architectures that present each core with a uniform cache access latency. The exponential increase in cache sizes required for multicore processors renders caches with uniform access impractical, as increases in capacity simultaneously increase access latency [20]. To mitigate this problem, recent research [24] proposes to decompose the cache into multiple slices. Each slice may consist of multiple cache banks to optimize for low access latency [5], with all slices physically distributed across the entire chip. Thus, cores realize fast accesses to the cache slices in their physical proximity and slower accesses to physically remote slices.

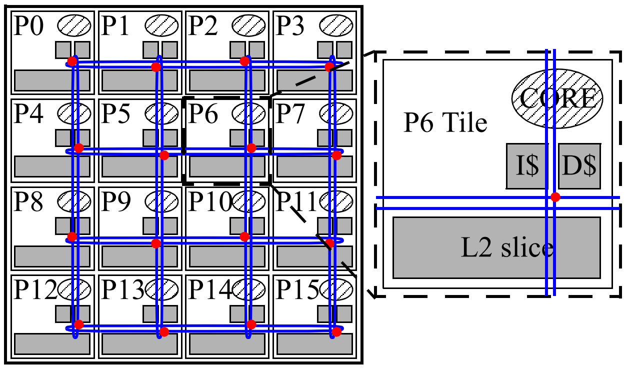


FIGURE 1. Typical tiled architecture. Each tile contains a core, L1 instruction and data caches, and a shared-L2 cache slice, interconnected into a 2-D folded torus.

Just as cache slices are distributed across the entire die, processor cores are similarly distributed. Thus, it is natural to couple cores and cache slices together and allow each core to have a "local" slice that affords fast access. Furthermore, economic, manufacturing, and physical design considerations [1, 43] suggest "tiled" architectures that co-locate distributed cores with distributed cache slices in tiles communicating via an on-chip interconnect.

2.2 Tiled architectures

Figure 1 presents a typical tiled architecture. Multiple tiles, each comprising a processor core, caches, and network router/switch, are replicated to fill the die area. Each tile includes private L1 data and instruction caches and an L2 cache slice. L1 cache misses probe the on-chip L2 caches via an on-chip network that interconnects the tiles (typically a 2D mesh). Depending on the L2 organization, the L2 slice can be either a private L2 cache or a portion of a larger distributed shared L2 cache. Also depending on the cache architecture, the tile may include structures to support cache coherence such as L1 duplicate tags [2] or sections of the L2-cache distributed directory.

Tiled architectures scale well to large processor counts, with a number of commercial implementations already in existence (e.g., Tilera's Tile64, Intel's Teraflops Research Chip). Tiled architectures are attractive from a design and manufacturing perspective, enabling developers to concentrate on the design of a single tile and then replicate it across the die [1]. Moreover, tiled architectures are beneficial from an economic standpoint, as they can easily support families of products with varying number of tiles and power/cooling requirements.

Private L2 organization. Each tile's L2 slice serves as a private second-level cache for the tile's core. On an L1 cache miss, only the L2 slice located in the same tile is probed. On a read miss in the local L2 slice, the coherence mechanism (a network broadcast or access to a statistically address-interleaved distributed directory) confirms that a copy of the block is not present on chip. On a write miss, the coherence mechanism invalidates all other on-chip copies. With a directory-based coherence mechanism, a typical coherence request is performed in three network hops. With token-coherence [27], a broadcast must be performed and a response must be received from the farthest tile.

Enforcing coherence requires large storage or complexity overheads. For example, a full-map directory for a 16-tile multicore processor with 1MB per L2 slice and 64-byte blocks requires 256K entries. Assuming a 42-bit physical address space, the directory size per tile is 1.3MB, exceeding the L2 capacity of the slice. Thus, full-map directories are impractical for the private L2 organization. Limited-directory mechanisms use smaller directories, but require complex, slow, and non-scalable fall-back mechanisms such as full-chip broadcast. In the rest of this work, we optimistically assume a private L2 organization where each tile has a full-map directory with zero area overhead.

Shared L2 organization. Each L2 slice is statically dedicated to caching a part of the address space, servicing requesting from any tile through the interconnect. On an L1 cache miss, the miss address dictates the tile responsible for caching the block, and a request is sent directly to that tile. The target tile stores both the cache block and its coherence state. Because each block has a unique location in the aggregate L2 cache, the coherence state must cover only the L1 cache tags; following the example for the private L2 organization and assuming 64KB split I/D L1 caches per core, the directory size is 168KB per tile.

2.3 Requirements for Intelligent Cache Block Placement

While the private and shared L2 organizations present two extremes in the design space, the latency characteristics of the distributed cache allow hybrid designs to strike a balance between these organizations. A distributed cache presents a range of latencies to a core, from fast access to slices near the core, to several times slower access to slices at the opposite side of the die. Intelligent cache block placement can improve performance by bringing data close to the requesting cores, allowing fast access.

We identify three key requirements to enable high performance operation of distributed NUCA caches through intelligent block placement. First, the address of a block must be decoupled from its physical location, enabling to store the block at a location independent of its address [9]. Placement of blocks in physical proximity of the requesting core allows fast access to these blocks; however, decoupling the physical location from the block address complicates searching for the block on each access. Thus, the second requirement for intelligent data placement is an effective cache lookup mechanism, capable of quickly and efficiently locating the cached block. Finally, intelligent data placement must optimize for all accesses prevalent in the workload. Different placement policies lend themselves to some access classes while penalizing others [44]. To achieve high performance, an intelligent placement algorithm must react appropriately to each access class.

3 Characterization of L2 References

3.1 Methodology

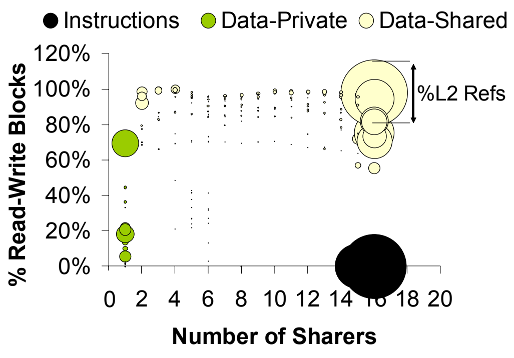
We analyze cache access patterns using trace-based and cycle-accurate full-system simulation of a chip multiprocessor (CMP) using *FLEXUS* [39]. FLEXUS models the SPARC v9 ISA and can execute unmodified commercial applications and operating systems. FLEXUS extends the *Virtutech Simics* functional simulator with models of processing tiles with out-of-order and in-order cores, NUCA cache, on-chip protocol controllers, and on-chip interconnect. We model a tiled CMP architecture summarized in Table 1 (left), running the *Solaris 8* operating system and executing workloads shown in Table 1 (right).

Future server workloads are likely to run on CMPs with a large number of small in-order cores [14] while multi-programmed desktop workloads are likely to run on CMPs with a small number of large out-of-order cores [7], allowing for more cache on chip. We run our multi-programmed mix on an 8-core tiled CMP with out-

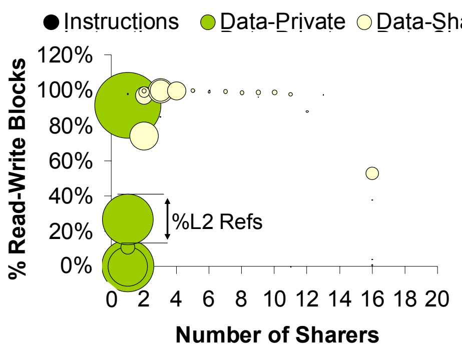
TABLE 1. System and application parameters for 16-core in-order and 8-core out-of-order CMPs.

CMP Size	16-core for server and scientific workloads 8-core for multi-programmed workloads
Processing Cores	UltraSPARC III ISA; 2GHz 8-stage pipeline 4-wide dispatch / retirement 32-entry conventional store buffer 16-core CMP: in-order cores 8-core CMP: OoO cores, 96-entry ROB, LSQ
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 3 ports, 32 MSHRs, 16-entry victim cache
L2 NUCA Cache	16-core CMP: 1MB / core, 14-cycle hit latency 8-core CMP: 3MB / core, 25-cycle hit latency 16-way set-associative, 64-byte lines 1 port, 32 MSHRs, 16-entry victim cache
Main Memory	3 GB total memory, 45 ns access latency
Memory Controller	one per 4 cores, round-robin page interleaving
Interconnect	2D torus (4x4 for 16-core CMP, 4x2 for 8-core) 32-byte links, 1-cycle link latency 2-cycle router latency

Online Transaction Processing (TPC-C)	
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server (SPECweb)	
Apache	16K connections, fastCGI, worker threading model
Decision Support (TPC-H)	
Qry 8	DB2, 450 MB buffer pool, 1GB database
Scientific	
em3d	768K nodes, degree 2, span 5, 15% remote
Multi-programmed (SPEC CPU2000)	
MIX	2 copies from each of gcc, twolf, mcf, art; reference input



(a) Server Workloads



(b) Scientific and multi-programmed Workloads

FIGURE 2. L2 Reference Clustering. Categorization of references to L2 blocks with respect the blocks' number of sharers, read-write behavior and instruction or data access class.

of-order cores, and the rest of our workloads on a 16-core tiled CMP with in-order cores.

To estimate the L2 cache size for each configuration, we assume a die size of 210mm^2 using 45nm technology and estimate the sizes of each component on chip following ITRS guidelines [33]. We allocate one memory controller per 4 tiles. We account for the area occupancy of the various system-on-chip components and allocate 65% and 75% of the chip area [14] for the processors and the NUCA cache in our 16-core and 8-core CMP respectively, taking into account that the larger-scale CMP requires more components (e.g., more memory controllers and a larger network). We estimate the area of the out-of-order and in-order cores by scaling the micrographs of the IBM Power5 and Sun UltraSparcT1 processors, using 1MB of L2 cache per core for the 16-core CMP and 3MB of L2 per core for the 8-core CMP.

Table 1 (right) enumerates our commercial and scientific application suite. We include the TPC-C v3.0 OLTP workload on *IBM DB2 v8 ESE* and *Oracle 10g Enterprise Database Server*. We run one query from the TPC-H DSS workload on DB2. We evaluate web server performance with the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. We drive the web servers using a separate client system (client activity is not included in timing results). We run one multi-programmed workload composed of SPEC CPU2000 applications running the reference input. Finally, we include one scientific application as a frame of reference for our server workload results.

With one exception, we focus our study on the workloads described in Table 1. To show the wide applicability of our L2 reference clustering observations, Figure 2 includes statistics gathered using a larger number of server workloads (OLTP on DB2 and Oracle, SPECweb99 on Apache and Zeus, TPC-H queries 6, 8, 11, 13, 16, and 20 on DB2), scientific workloads (em3d, moldyn, ocean, sparse), and the multi-programmed workload from Table 1.

3.2 Categorization of Cache Accesses

We analyze the L2 memory accesses at the granularity of a single cache block along two axes: the number of cores sharing the block and the percentage of read-write blocks. We plot the results in Figure 2. For every workload, we plot two bubbles for each number of sharers (1-16), one bubble for instruction and another for data accesses. The bubble size corresponds to the relative frequency of the accesses for that workload. We indicate instruction-access bubbles in black and data-access bubbles in white (shared) or green (private), drawing a distinction for private blocks (accessed by only one core).

We observe that, in server workloads, L2 references naturally form three clusters with distinct characteristics: (1) instructions are shared by all cores and are entirely read-only, (2) shared data are generally read-write and shared among all cores, (3) private data exhibit a varying degree of read-write blocks. We further observe that scientific and multi-programmed workloads mostly access private data, with a small fraction of shared accesses in data-parallel scientific codes exhibiting producer-consumer or nearest-neighbor communication. The instruction footprints of scientific and multi-programmed workloads are effectively captured by the L1-I cache.

We present the normalized breakdown of L2 references in Figure 3. Although server workloads are dominated by accesses to instructions and shared read-write data, a significant fraction of L2 references are to private blocks. The scientific and multi-programmed workloads are dominated by accesses to private data, but also exhibit shared data accesses. With the exception of the multi-programmed workload (SPEC CPU2000), our workloads underscore the need to react to the access class when implementing the L2 placement scheme, and emphasize the opportunity loss of addressing only some of the access classes.

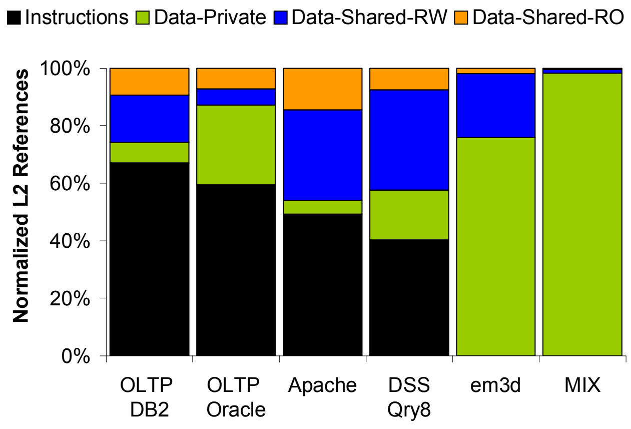


FIGURE 3. L2 Reference breakdown. Distribution of L2 references by access class.

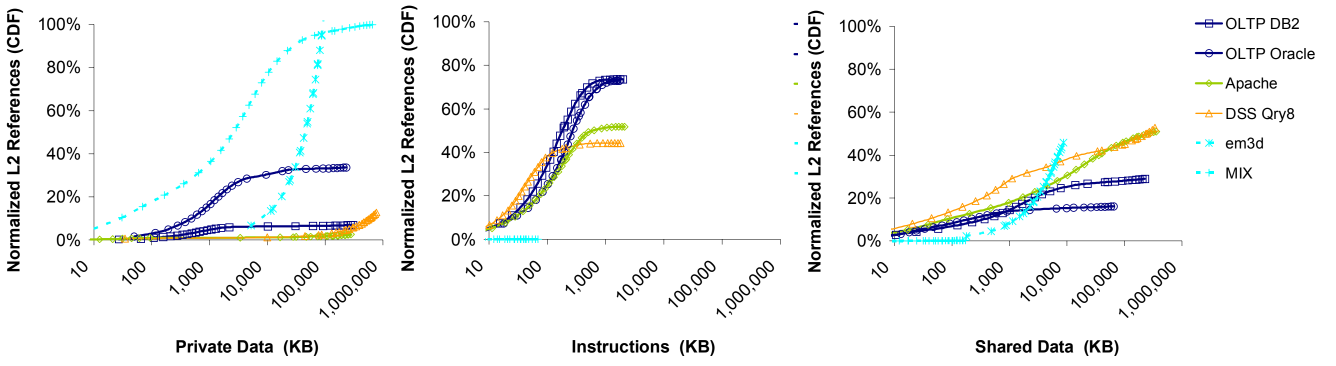


FIGURE 4. L2 working set sizes. CDF of L2 references to private data, instructions, and shared data vs. the footprint of each access class (in log-scale). References are normalized to total L2 references for each workload.

The categorization axes of Figure 2 suggest an appropriate L2 placement scheme for each access class. Blocks accessed by a single core (private data) are prime candidates for allocation at the requesting core; placement at the requesting tile achieves lowest possible access latency, avoiding the need for coherence because private blocks are always read or written by the same core. Read-only and universally shared blocks (e.g., instructions) are prime candidates for replication on chip; replication allows locating blocks in close proximity to the requesting core, while their read-only nature does not require coherence. Finally, read-write blocks with many sharers (shared data) may benefit from migration or replication if the shared data blocks exhibit reuse by one or a group of cores. However, migration requires complex lookup mechanisms and replication requires coherence enforcement, underscoring the need for an intelligent mechanism to find an appropriate location for the block on chip.

3.3 Characterization of Access Classes

3.3.1 Private Data

Accesses to private data, such as stack space and thread-local storage, are characterized by always being initiated from the same processor core. Because requests are always initiated by the same core, replicating private data to make them available at multiple locations on chip only leads to wasted cache capacity [42]. Therefore, despite private data comprising read-only and read-write blocks, a cache coherence mechanism is not necessary, leading to a single placement goal for private data: to be cached at a slice close to the requestor, so that private data references may be satisfied with minimal latency.[1] The R-NUCA placement policy satisfies the private-data placement goal by always placing private data into the L2 slice within the same tile as the requesting core.

We analyze the private-data working sets of our workloads in Figure 4 (left). Although the private-data working set for OLTP and multi-programmed workloads may fit into a single (local) L2 slice, DSS workloads scan multi-gigabyte database tables and scientific workloads operate on large data sets, making their private-data working sets exceed any reasonable L2 capacity. To accommodate large private working sets, prior proposals advocate migrating (spilling) these blocks to neighbors [11]. Although spilling may be applicable to some multi-programmed workloads composed of applications with a range of private-data working sets, it is inapplicable to server or scientific workloads. All cores in a typical server or balanced scientific workload run similar threads, with each L2 slice having similar capacity pressure. Migrating private data blocks to a neighboring slice to relieve cache pressure on the local slice is offset by the neighboring slices undergoing an identical operation and spilling blocks in the opposite direction. As a result, cache pressure remains the same, but private data references incur greater access latency.

3.3.2 Instructions

Instruction blocks are typically written once when the operating system loads an application binary or shared library into memory from disk. Once in memory, instruction blocks remain read-only for the duration of execution. Figure 2 indicates an important characteristic of instruction blocks in server workloads: instruction blocks are universally shared among the processor cores. All cores in server workloads typically exercise the same instruction working set, with all cores requiring low-latency access to the instruction blocks with equal probability. Instruction blocks are therefore amenable to replication. By caching multiple copies of the blocks, replication enables low-latency access to the instruction blocks from multiple locations on chip.

In addition to replication, in Figure 5 (left) we examine the utility of instruction-block migration toward a requesting core. We present the percentage of all L2 references constituting the 1st, 2nd, and subsequent instruction-block accesses by a single core without intervening L2 accesses for the same block by a different core. Thus, the grey and higher portions of the bars represent reuse accesses that could experience a lower access latency if the instruction block was migrated toward the requesting core after the first access. Based on these results, we observe that accesses to L2 instruction blocks are finely interleaved between participating sharers, yielding minimal potential benefit in migration of instruction blocks. On the contrary, allowing for migration may be detrimental to performance, as migration may increase contention in the on-chip network.

A potential down-side to instruction-block replication arises due to excessive replication. Replication leads to a reduction of aggregate

1. In some situations, the operating system may migrate a thread from one core to another, with all subsequent accesses to the thread's private data being initiated by the destination core. In these cases, coherence can be preserved, by the operating system, through shoot-down of the private blocks at the time of thread migration.

L2 capacity because the same block simultaneously occupies multiple frames in the cache, leading to a higher aggregate off-chip miss rate. Additionally, careful placement of replicas is required to avoid caching multiple copies of a block in L2 slices at close physical proximity to each other; for example, there is virtually no access latency benefit to caching the same instruction block in two adjacent L2 slices.

Figure 4 (middle) shows that the instruction working set of some server workloads exceeds the size of a single L2 slice (e.g., OLTP on Oracle has a working set of 1.2MB). Indiscriminately replicating instruction blocks for these workloads in every L2 slice causes excessive cache pressure; even in case the instruction working set fits into an L2 slice, the instruction blocks strongly compete for cache capacity with data blocks. We therefore conclude that instruction blocks benefit most from a placement policy that divides the L2 into clusters of neighboring slices, replicating instructions at the granularity of a cluster rather than individual L2 slices. While the applications' instruction working set is too large to be cached in individual L2 slices, the working set fits into the aggregate capacity of a cluster. Each slice participating in a cluster of size *n* should store *1/n* of the instruction working set. By controlling the cluster size, it becomes possible to smoothly trade off instruction-block access latency for cache capacity; whereas a large number of small clusters will offer minimal access latency while consuming a large fraction of capacity of each participating slice, a small number of large clusters will result in larger average access latency but fewer overall replicas.

We find that for the workloads we study in our CMP configurations, a cluster of size 4 is sufficient for replicating instruction blocks. By each caching a quarter of the instruction working set, clusters of 4 neighboring slices ensure that instruction blocks are at most one network hop away from the requesting core.

3.3.3 Shared Data

Shared data comprises predominantly read-write data blocks, containing both data and synchronization or locking mechanisms to protect that data. Replication or migration of shared blocks can provide low-latency access for the subsequent references to the same block from the local or nearby cores. However, both replication and migration of read-write blocks require complex coherence mechanisms to invalidate the replicated or migrating L2 blocks on every write to the data. Figure 5 (right) shows the count of accesses to a shared data block by the same core after a write by a different core (writes by the same core hit in the L1-D cache and are not visible at the L2). We observe that, for shared data, half of the accesses are the "1st access" after a write, and the vast majority of accesses are either the "1st access" or "2nd access" after a write. We therefore conclude that an invalidation will occur nearly after every replication or migration opportunity, eliminating the possibility of accessing the block at its new location in most cases, and rendering both techniques ineffective. Furthermore, due to frequent invalidations, replication of shared blocks reduces the effective aggregate cache capacity not only due to storage of live replicas, but also due to storage of a large number of invalidated frames.

While the shared data access-pattern characteristics shown in Figure 5 (right) and the large working set size shown in Figure 4 (right) indicate minimal opportunity for replication or migration of shared data, the implementation complexity and overheads of these mechanisms entirely overshadow their potential benefit. Although

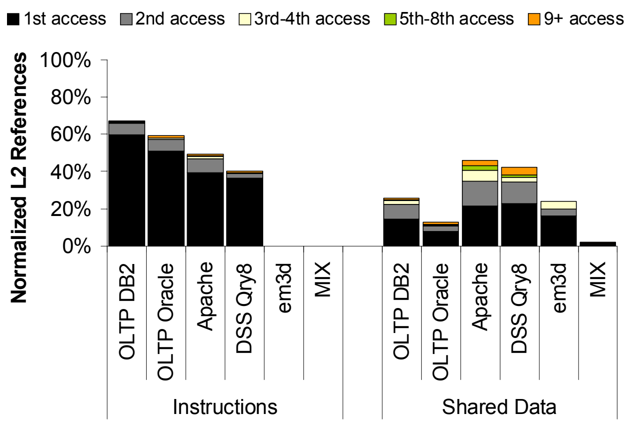


FIGURE 5. Instruction and shared data reuse. Reuse of instructions and shared data by the same core. References are normalized to total L2 references for each workload.

in our discussion of the benefits of replication and migration we assume the existence of a fast lookup mechanism, to date there have been only few promising directions discovered in this domain [31]. To support read-write blocks at arbitrary locations on chip, either directory-based or broadcast-based coherence mechanism must be implemented. The area costs of a directory-based L2 coherence scheme were estimated in Section 2.2, showing that a directory-based scheme imposes large area overheads that drastically reduce the on-chip area dedicated to L2 capacity, while the bandwidth and power overheads of a broadcast-based mechanism that probes multiple cache slices per access do not scale well even up to 16 slices.

Handling of shared read-write data in a NUCA cache presents a challenging problem due to the coherence requirements, diverse access patterns, and large working set of these data. The challenge has been recognized by prior studies in NUCA architectures, however the problem remained largely unaddressed, with the best proposals completely ignoring shared read-write blocks [4] or ignoring them once the adverse behavior of shared read-write blocks is detected [10].

On the other hand, we find that shared data can be directly handled by placing them at a fixed location in the cache. Because shared data blocks in server workloads are universally accessed (Figure 2) and the next sharer is not known *a priori* [35], every core accessing the shared block has the same likelihood to be the next accessor, and the average latency to access shared data in L2 remains the same, no matter in which L2 slice these blocks reside. We determine the placement of shared cache blocks by static address interleaving over the entire aggregate L2. Placing these blocks at their appropriate address-interleaved location allows us to define a single, fixed location for each block, utilize a trivial and fast lookup mechanism (the address bits uniquely determine the location), forego coherence among the L2 slices, and eliminate wasted space. Sharers request such blocks through the interconnection network, and they are allowed to cache them at their local L1 cache but not their local L2 slice. Because the latency to access shared data depends on the network topology, accesses to statically placed shared data benefit most from a topology that avoids hot spots and affords best-case (average) read and write latency for all cores (e.g., a torus interconnect).

3.4 Characterization Conclusions

In Section 4 we present the necessary mechanisms to support Reactive NUCA, a low complexity mechanism to achieve low-latency access in NUCA caches. Based on the characterization of private-data, instruction, and shared-data access patterns, we summarize the conclusions which motivate and guide the R-NUCA design:

- An intelligent placement policy is sufficient to achieve low access-latency for the major access classes
- L2 hardware coherence mechanisms in a tiled CMP architecture are unnecessary and should be avoided
- Private blocks should be placed into the local slice of the requesting core
- Instruction blocks should be replicated in non-overlapping clusters (groups) of slices
- Shared data blocks should be placed at fixed address-interleaved on-chip locations

4 R-NUCA Design

We base our design on a CMP with private split L1 I/D caches and a distributed shared L2 cache. The L2 cache is partitioned into "slices," which are interconnected by an on-chip 2-D folded torus network. We assume that cores and L2 slices are distributed on the chip in tiles, forming a tiled architecture similar to the one described in Section 2.2. This assumption is made to simplify the explanation of our design, and is not a limitation. The mechanisms we describe apply to alternative organizations, for example, groups of cores assigned to a single L2 slice.

Conceptually, the R-NUCA placement scheme operates on overlapping "clusters" of one or more tiles. R-NUCA introduces "fixed-center" clusters, which consist of the tiles logically surrounding a core. Each core defines its own fixed-center cluster. For example, clusters C and D in Figure 6 each consist of the tile at the center and the neighboring tiles around it. Because fixed-center clusters logically surround a core, they allow for fast nearest-neighbor communication.

Clusters can be of various power-of-2 sizes. Clusters C and D in Figure 6 are size-4. Size-1 clusters always consist of a single tile (e.g., cluster B). In our example, size-16 clusters comprise all tiles (e.g., cluster A). As shown in Figure 6, clusters may overlap each other. Data within each cluster are interleaved among the participating L2 slices, and shared among all cores participating in that cluster.

4.1 Indexing and Rotational Interleaving

R-NUCA indexes blocks within each cluster using the standard address interleaving and the rotational interleaving indexing schemes. In standard address interleaving, an L2 slice is selected based on the bits immediately above the set-index bits of the accessed address. R-NUCA uses standard address interleaving for the size-16 and size-1 clusters (in the size-1 cluster, the scheme degenerates to a single interleave: the only slice in the cluster).

To index blocks in a size-4 cluster, R-NUCA utilizes rotational interleaving. Under rotational interleaving, each core is assigned an ID by the operating system, e.g., at boot time. This ID may be different from the conventional core ID that the OS gives to each core. To avoid confusion, in the remainder of this paper we refer to the rotational ID as RID, and to the conventional one as CID. For illustration purposes, we assume that a tile, its core, and its slice share the same RID and CID.

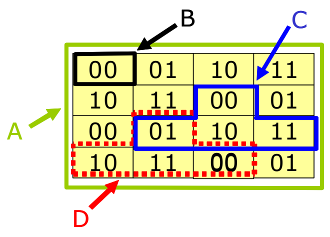


FIGURE 6. Example of R-NUCA clusters and Rotational Interleaving. The array of rectangles represents the tiles. The binary numbers in the rectangles denote the tile's RID. The lines surrounding some of the tiles are clusters.

RIDs in a size-*n* cluster range from *0* to *n-1*. To assign RIDs, the OS first assigns to a random tile the RID 0. Consecutive tiles in a row receive consecutive RIDs (*n-1* wraps around to *0*, and the two are considered consecutive). Similarly, consecutive tiles in a column are assigned RIDs that differ by $log_2(n)$, again with *n-1* wrapping around to *0*. An example of RID assignment for size-4 fixed-center clusters is shown in Figure 6.

To index a block in its size-4 fixed-center cluster, the center core uses the 2 address bits $<a_1,a_0>$ immediately above the set-index bits. The core compares the address bits with its own RID $<c_1, c_0>$ using boolean logic[1], and the outcome of the comparison determines whether the core indexes the block at its local slice, the slice above it in the cluster, the slice on the left, or the slice on the right. In our example in Figure 6, if the center core in cluster C accesses a block with address bits *<0, 1>*, the core will evaluate the boolean function and look for the block at the slice on its left. Similarly, when the center core at cluster D accesses the same block, it will attempt to find it at the same slice (above the center of cluster D). Thus, each slice stores exactly the same *1/n* of the data on behalf of any cluster it belongs to. This property of rotational interleaving allows clusters to replicate data without increasing cache pressure, and at the same time affording nearest-neighbor communication.

The implementation of rotational interleaving is trivial. It requires only that tiles have RIDs, and indexing is performed through simple boolean logic on the tile's RID and the block's address. The rotational-interleaving scheme can be generalized to clusters of any power-of-two size, however, for illustration purposes, we only describe it for size-4 clusters.

4.2 Placement

Depending on the access latency requirements, the working set, the user-specified configuration, or other factors available to the OS, the system can smoothly trade off latency, capacity, and replication degree by varying the cluster sizes. Based on the cache block's classification presented in Section 3.2, R-NUCA selects the appro-

1. The general form of the indexing function for size-*n* clusters with the address interleaving bits starting at offset *k* is:

$$R = (Addr[k + \log_2(n) : k] + \overline{RID} + 1) \wedge (n - 1)$$

For size-4 clusters, the 2-bit result instructs the core to send the request to the slice that is local to the core, to the right, above or to the left, for binary results *<0,0>*, *<0,1>*, *<1,0>* and *<1,1>* respectively.

priate cluster and places the block according to the address interleaving of the slices within this cluster.

In our configuration, R-NUCA utilizes only clusters of size-1, size-4 and size-16. R-NUCA places core-private data in the size-1 cluster encompassing the core, ensuring lowest access latency. Shared data blocks are placed in size-16 clusters which are fully overlapped by all sharers. The shared data placement policy avoids replication, obviating the need for a coherence mechanism by ensuring that, for each shared block, there is a unique slice to which that block is mapped by all sharers. Instructions are allocated in the most size-appropriate fixed-center cluster (size-4 for our workloads), and are replicated across clusters on chip. Thus, instructions are shared by neighboring cores and replicated at distant ones, ensuring low access latency for surrounding cores while balancing capacity constraints. Although R-NUCA forces an instruction cluster to experience an off-chip miss rather than retrieving blocks from other on-chip replicas, the performance impact of these “compulsory” misses is negligible.

4.3 Page Classification

R-NUCA performs classification of memory accesses at the time of a TLB miss. Classification is performed at the OS-page granularity, and communicated to the processor cores using the standard TLB mechanism. Requests from the L1 instruction cache are immediately classified as “instructions” and a lookup is performed assuming a size-4 fixed-center cluster centered at the requesting core. All other requests are classified as data requests, and the operating system is responsible for distinguishing between private and shared data accesses.

To determine a private or shared classification for data pages, the operating system extends the page table entries with a bit that denotes the current classification, and a field to record the CID of the last core to access the page. Upon the first access, a core encounters a TLB miss and traps to the OS. The OS marks the faulting page as private and the CID of the accessor is recorded. The accessor receives a TLB fill with an additional Private bit set. On any subsequent request, during the virtual-to-physical translation, the requestor also examines the Private bit and looks for the block only in its own local slice.

On a subsequent TLB miss, the OS compares the CID in the page table entry with the CID of the core encountering the TLB miss. In the case of a mismatch, two situations are possible. Either the thread accessing this page has been migrated to another core and the page is still private to the thread, or the page is actively shared by multiple cores and it must be re-classified as shared. Because the OS is fully aware of thread scheduling, it can precisely determine whether or not thread migration took place, and correctly classify a page as private or shared.

If the page is actively shared, the OS must re-classify the page from private to shared. Upon a re-classification, the OS first sets the page to a poisoned state. Subsequent requests for the page are held until the poisoned state is cleared. Then, the OS shoots down the TLB entry and invalidates any cache blocks belonging to this page at the tile CID marked as the previous accessor’s.[1] When the shoot-down operation completes, the OS classifies the page as shared by clears the Private bit, removes the poisoned state from the page table entry, and services any pending TLB requests. Because the private bit is now cleared, any core that receives a TLB entry will treat accesses to this page as shared, applying the standard address interleaving over the entire aggregate cache to locate the shared block.

The case for thread migration from one core to another is handled in the same manner. The only difference being that the page retains its private classification, and the CID in the page table entry is updated to the CID of the new owner of the page.

The implementation of R-NUCA’s placement scheme is simple. Placement is achieved through an interaction of the OS and a boolean-logic index-remapping mechanism. The OS extends each page table entry with $log_2(n)+1$ bits, performs trivial classification at page granularity upon a TLB miss, and communicates the classification to the cores via the standard TLB fill operation. Each TLB entry is extended with a bit to indicate whether the page holds private or shared data. Instruction references are immediately recognized, as they come from the instruction cache. On an L1-D miss, the request’s original (I or D cache) and the additional Private TLB bit guides the index-remapping logic in the selection of the L2 slice to which the request should be sent. Both the software and hardware mechanisms are inherently simple: the operating system has precise knowledge of all the information added to the TLB entry; the index-remapping hardware applies simple boolean logic on the TLB bit, the requested address, and the core’s RID.

4.4 Extensions

While our configuration of R-NUCA utilizes only clusters of size-1, size-4 and size-16, the techniques can be applied to clusters of different types and sizes. For example, R-NUCA can utilize fixed-boundary clusters, which have a fixed rectangular boundary and all cores within the rectangle share the same data. The regular shapes of these clusters make them appropriate for partitioning a multicore processor into equal-size non-overlapping partitions, which may not always be possible with fixed-center clusters. The regular shapes come at the cost of allowing a smaller degree of nearest-neighbor communication, as tiles in the corners of the rectangle are farther away from the other tiles in the cluster.

The indexing policy is orthogonal to the cluster type. Indexing within a cluster can be performed using either standard address interleaving or rotational interleaving. The choice of interleaving policy depends on the replication requirements of the data. Rotational interleaving is appropriate for replicating data while balancing capacity constraints. Standard address interleaving is appropriate when clusters are disjoint. By designating a “center” for a cluster and communicating it to the cores via the TLB mechanism, both interleaving mechanisms are possible for any cluster type of any size.

Although we evaluate R-NUCA with fixed-center clusters only for instructions, different configurations are possible. For example, when running heterogeneous workloads where the threads at each core have different private cache capacity requirements, it is possible to use a fixed-center cluster of appropriate size for private data, effectively spilling blocks to the neighboring slices to lower cache capacity pressure while retaining fast lookup.

1. This step is required to guarantee coherence and can be performed by any shoot-down mechanism such as scheduling a special shoot-down kernel thread at the previous accessor’s core; special instructions or PAL code routines to perform this operation already exist in many of today’s architectures.

5 Evaluation

5.1 Methodology

For each CMP configuration we implement four mechanisms to manage the NUCA cache: shared, private, ASR, and R-NUCA. The shared and private organizations are similar to the ones described in Section 2.2. ASR [4] is based on the private scheme and adds an adaptive mechanism which, upon an L1 eviction of a clean shared block, it decides with some probability whether to allocate that block in the private L2 slice. If it decides not to allocate it, the block is allocated at an empty cache frame at another L2 slice, or it is dropped if no empty L2 frame exists or there are other sharers on chip. We select ASR for our comparison because it has been shown to outperform all prior proposals for on-chip cache management in a multicore processor.

Although we did a best-effort implementation for ASR, our results did not match with [4]. We believe that the assumptions of our system penalize the ASR implementation, while the assumptions of [4] penalize the shared cache implementation that we use as our baseline. The relatively fast memory system (90 cycles vs. 500 cycles in [4]) and the long-latency coherence operations due to our directory-based implementation ([4] utilizes token broadcast) leave ASR with a small opportunity for improvement. We implemented three versions for ASR: an adaptive version following the guidelines in [4], a scheme that always chooses to allocate an L1 victim at the local slice, and a scheme that always sends the L1 evicts through the logical write-back ring. In our results for ASR, we report the highest-performing mechanism of these three implementations for each workload.

For the private and ASR schemes we assume optimistically an on-chip full-map distributed directory with zero area overhead. In reality, a full-map directory will require more area than the aggregate L2 cache, and novel approaches are required to maintain coherence among the tiles with a lower overhead. Such techniques are beyond the scope of this paper. Similarly, we assume that ASR has no area overhead. Thus, the speedup of R-NUCA against a realistic private and ASR organization based on distributed directories will be higher than reported in this paper.

Our on-chip coherence protocol is a four-state MOSI protocol modeled after Piranha [2]. Our cores perform speculative load execution and store prefetching as described in [12,18]. Hence, our base system is similar to the system described in [30]. We simulate one memory controller per four cores. Each memory controller is co-located with one tile and communicates with memory through flip-chip transistors. The controllers communicate with the tiles through the on-chip interconnection network and pages are interleaved among all controllers in a round-robin fashion. The page size in our system is 8KB. We list other relevant parameters in Table 1 (left).

For the interconnection network we simulate a 2-D folded torus [13]. While prior research typically utilizes mesh interconnects due to their simple implementation, meshes allow hot spots to form in the middle of the network and penalize tiles at the edges with detrimental effects to performance. In contrast, torus interconnects have no "edges" and treat nodes homogeneously, spreading the traffic across all links and avoiding hot spots. We believe that 2-D tori can be built efficiently in modern VLSI by following a folded topology [37] which eliminates long links. While a 2-D torus is not planar, each one of its dimensions is, requiring only two metal layers for the interconnect [37]. With current commercial products already featuring 11 metal layers we believe 2-D torus interconnects are a feasible design point. Moreover, tori have compared favorably against meshes with respect to area and power overhead [37]. All cache organization alternatives we evaluate assume a 2-D torus topology, thus all alternatives receive exactly the same benefits from the interconnect.

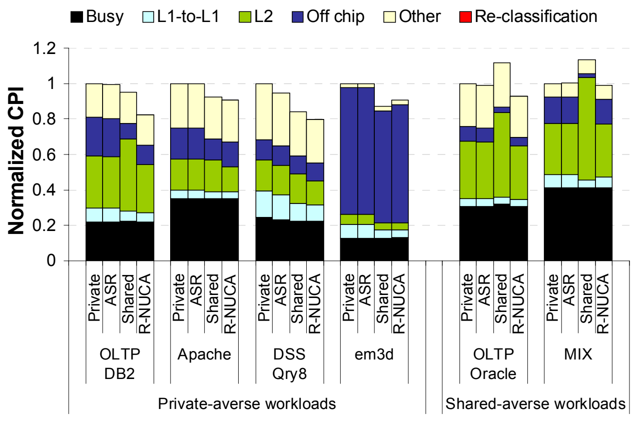


FIGURE 7. Total CPI breakdown for L2 organizations. CPI is normalized to the private organization.

We measure performance using the SimFlex multiprocessor sampling methodology [39]. The SimFlex methodology extends the SMARTS [40] statistical sampling framework to multiprocessor simulation. Our samples are drawn over an interval of 10s to 30s of simulated time for OLTP and web server applications, over the complete query execution for DSS, over a complete single iteration for the scientific application and over the first 10 billion instructions for the multi-programmed workload after all applications have entered their loop iterations. We launch measurements from checkpoints with warmed caches, branch predictors, TLBs, on-chip directory and OS page table, then warm queue and interconnect state for 100,000 cycles prior to measuring 50,000 cycles. We use the aggregate number of user instructions committed per cycle (i.e., committed user instructions summed over all cores divided by total elapsed cycles) as our performance metric, which is proportional to overall system throughput [39].

5.2 Classification Accuracy

While we performed the workload analysis in Section 3 at the granularity of cache blocks, R-NUCA classifies entire pages. Pages may contain blocks of a different class, for example part of a page may contain private data, while the remaining may contain shared data. For our workloads, we found that between 10% - 27% of L2 references are to pages with more than one class. However, the references issued to these pages are dominated by a single class. If a page services both shared and private data, accesses to shared data dominate. By designating such a page as "shared-data" we effectively capture the majority of the references and miss-classify only a small portion of them. As a result, classification at page granularity results in the miss-classification of only 0.64% - 0.75% of the L2 references, which is a very small cost to pay for the benefits awarded by using pages. Thus, we conclude our page classification is accurate.

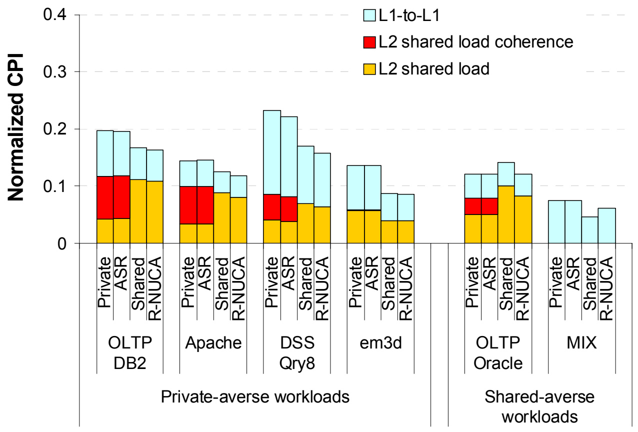


FIGURE 8. Impact of L2 coherence elimination. The CPI is normalized to the total CPI of the private organization.

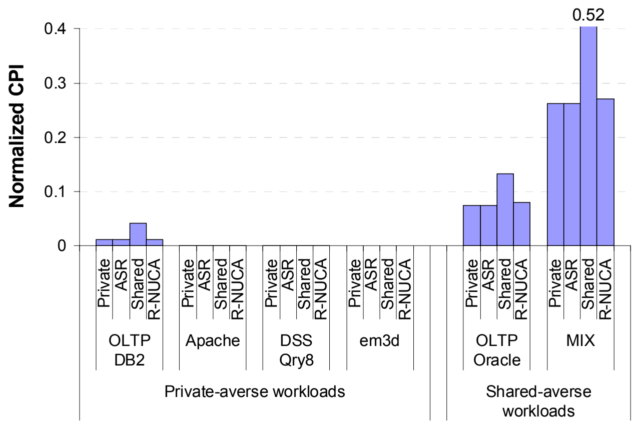


FIGURE 9. Impact of local allocation of private data. The CPI is normalized to the total CPI of the private organization.

5.3 Impact of R-NUCA Mechanisms

We compare R-NUCA against the shared, private and ASR organizations on CMPs running our workload suite. Because different workloads favor a different cache organization, we split our workloads into two categories: private-averse and shared-averse based on which organization has a higher CPI. Private may perform poorly when it increases the number of off-chip accesses, or when there is a large number of L1-to-L1 or L2 coherence requests. An L1-to-L1 request occurs when a core misses on its private L1 and L2 slice, and the data are transferred from a remote L1. An L2 coherence request occurs when a core misses on its private L1 and L2 slice, and the data are transferred from a remote L2. The private and ASR organizations penalize such requests, because the request has to be sent first to the on-chip distributed directory, which will forward the request to the remote tile, which then probes its L2 slice and (if needed) its L1 and replies with the data. Thus, such requests incur additional network traversals and additional accesses to remote L2 slices. Similarly, the shared organization may perform poorly when there are a lot of accesses to private data or instructions, which the shared scheme spreads across the entire chip, while the private scheme services through the local and fast L2 slice.

Figure 7, shows a breakdown of the cycles-per-instruction (CPI) normalized to the private organization. We measure the CPI due to useful computation (busy), L1-to-L1 transfers, L2 loads and instruction fetches (L2), off-chip accesses, other delays (e.g., front-end and L1 stalls), and the CPI due to page re-classifications in R-NUCA. We account for loads separately from stores, as read latency is difficult to overlap, while recent techniques minimize store latency [38,6]. Thus, we account for store latency in the "other" category. Figure 7 shows that re-classifications result in negligible overhead due to their infrequency. Overall, R-NUCA delivers on its promise and outperforms the competing organizations, as it lowers the L2 hit latency exhibited by the shared organization, and eliminates the long-latency coherence operations of the private and ASR schemes.

Impact of L2 coherence elimination. Figure 8 shows the portion of the total CPI due to accesses to shared data, which may engage the coherence mechanism. Shared data in R-NUCA and the shared organization are interleaved across all L2 slices, thus such requests are of equal latency for both. The private and ASR schemes replicate data locally, so sometimes a request is serviced from the local L2 slice ("L2 shared load") while other times it is serviced from a remote one ("L2 shared load coherence"). While accesses to the local L2 slice are fast, accesses to a remote tile engage the on-chip coherence mechanism, and require one more network traversal and one more L2 slice access than shared or R-NUCA. Thus, the benefits of fast local reuse for shared data under the private and ASR schemes are quickly outweighed by the long-latency coherence operations. On average, the elimination of L2 coherence requests allow R-NUCA to exhibit a 11% lower CPI contribution of accesses to shared data. Similarly, L1-to-L1 requests require an additional remote L2 slice access in the private and ASR schemes, as the coherence mechanism works at the granularity of tiles. By eliminating the additional remote L2 slice access, R-NUCA lowers the latency for L1-to-L1 requests by 27% on average. Overall, eliminating the coherence requirements at L2 lowers the CPI due to shared data accesses by 21% on average against the private and ASR schemes.

Impact of local allocation of private data. Similar to the private and ASR schemes, R-NUCA allocates private data at the local L2 slice for fast access, while the shared scheme distributes them across all L2 slices, requiring more cycles per request. Figure 9 shows the impact of allocating the private data locally. Overall, R-NUCA lowers the latency of accessing private data by 48% against shared, matching the performance of the private scheme's when accessing private data.

Impact of instruction clustering. While R-NUCA's clustered replication scheme spreads instructions between neighbors, they are only one hop away from the requestor and the fast lookup afforded by the rotational interleaving matches the speed of a local L2 access. In contrast, the shared scheme spreads instruction blocks across the entire chip area, therefore requiring significantly more cycles for each instruction L2 request (Figure 10). As a result, R-NUCA obtains instruction blocks from L2 on average 38% faster than shared. In OLTP-Oracle, it obtains instructions even faster than the private scheme, as the latter accesses remote tiles to fill some of its requests.

While the private scheme affords fast instruction L2 accesses, the excessive replication of the instruction stream causes the eviction

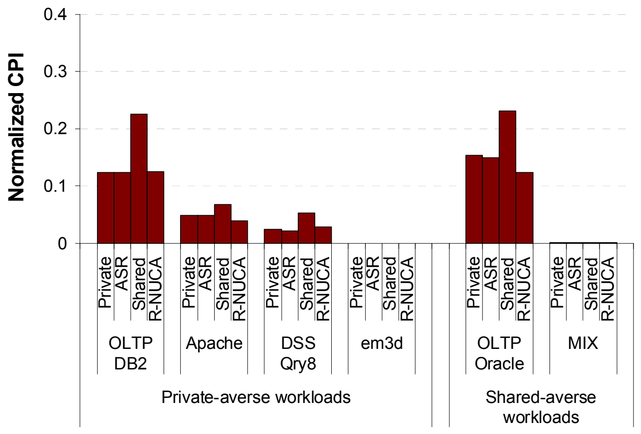


FIGURE 10. CPI contribution of L2 instruction accesses. The CPI is normalized to the total CPI of the private scheme.

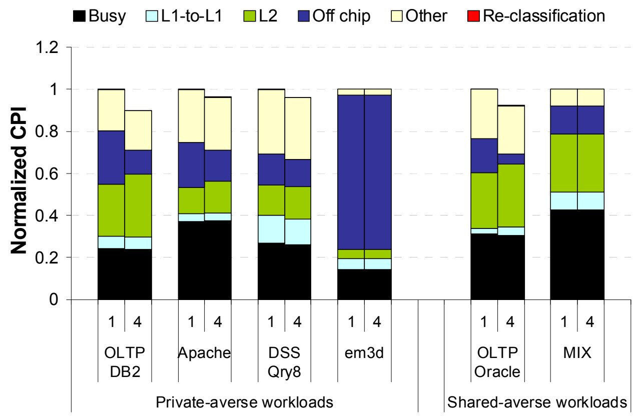


FIGURE 11. CPI breakdown with size-1 and size-4 instruction clusters. The CPI is normalized to size-1.

of data blocks and a subsequent increase in off-chip misses. Figure 11 compares size-1 instruction clusters, where instructions are stored in the local L2 slice, with size-4 clusters, similar to the ones in our configuration of R-NUCA. We find that storing instructions only in the local L2 slice increases the off-chip CPI component by 70% on average over a size-4 cluster, with detrimental effects to performance. At the same time, as Figure 10 suggests for the shared organization, clusters larger than size-4 spread instruction blocks to a larger area, increasing the latency to access them by almost 40%. We find that, for our workloads, a cluster size of four gives the best balance between L2 hit latency and off-chip misses.

5.4 Performance Improvement

Overall, we find that R-NUCA lowers the CPI contribution of L2 hits by 15% on average against private, and 26% on average against shared. At the same time, R-NUCA is effective in maintaining the large aggregate capacity of the distributed L2 cache much like the shared organization does. The CPI contribution of off-chip misses for R-NUCA is on average within 20% of the shared scheme's, while the private organization increases the off-chip CPI by 80% on average. Thus, R-NUCA delivers both the fast local access of the private scheme, as well as the large effective cache capacity of the shared scheme, therefore bridging the gap between the two organizations. Even more, it avoids the long-latency coherence operations of the private and ASR schemes, achieving even better performance. As a result, R-NUCA provides an average speedup of 17% against private on private-averse workloads, and 17% against shared on shared-averse workloads. The corresponding speedups are shown at Figure 12, along with the 95% confidence intervals produced by our sampling methodology. The results are even more encouraging for server workloads alone, where R-NUCA attains an average speedup of 20% against either alternative.

5.5 Impact of Technology

As Moore's Law continues and the number of cores on chip continue to grow, the on-chip interconnect and the aggregate cache will grow commensurately. This will make the shared organization even less attractive, as cache blocks will be spread over an ever increasing number of tiles. At the same time, the coherence demands of the private and private-based schemes will grow by the size of the aggregate cache, increasing the area and latency overhead for accesses to shared data. R-NUCA eliminates coherence among the L2 slices, therefore avoiding the private organization's overheads, while still exhibiting fast L2 access times. Moreover, by allowing for the local and nearest-neighbor allocation of blocks, R-NUCA will continue to provide an ever-increasing performance benefit over the shared scheme. Finally, we believe that the generality of R-NUCA's clustering scheme will allow for the seamless decomposition of a large-scale multicore chip into virtual domains, each one with its own subset of the cache, where each domain will experience fast and trivial cache lookup through rotational interleaving with minimal hardware and operating system involvement.

6 Related work

To mitigate the access latency of large on-chip caches, Kim proposed Non-Uniform Cache Architectures (NUCA) [24], showing that a network of independent cache banks can be used to reduce average access latency. Chishti proposed to decouple physical placement from logical organization [9] to add flexibility to the NUCA design.

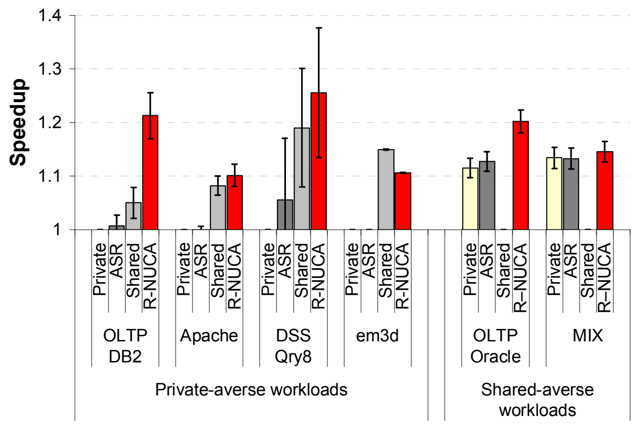


FIGURE 12. Performance Improvement. Speedup over the private organization for private-averse workloads, and over the shared organization for shared-averse workloads.

Beckmann evaluated NUCA architectures in the context of CMPs [5], concluding that dynamic migration of blocks within a NUCA can benefit performance but requires smart lookup algorithms and may cause contention in the physical center of the cache. Kandemir proposed migration algorithms to achieve near-optimal location for each cache block [23], and Ricci proposed smart lookup mechanisms using Bloom filters [31]. In contrast to these works, R-NUCA avoids block migration in favor of intelligent block placement, thus avoiding the central contention problem and eliminating the need for an intelligent lookup algorithm.

Zhang observed that different classes of accesses benefit from either a private or shared system organization [44] in multi-chip multi-processors. Falsafi proposed to apply either private or shared organization by dynamically adapting the system on a per-page granularity [16]. R-NUCA similarly applies either a private or shared organization at page granularity, however we leverage the OS to properly classify the pages, avoiding reliance on heuristics.

Huh extended the NUCA work to CMPs [21], investigating the effect of sharing policies. Yeh [41] and Merino [29] proposed coarse-grain approaches of splitting the cache into private and shared slices. Guz advocated building separate but exclusive shared and private regions of cache. R-NUCA similarly treats data blocks as private until accesses from multiple cores are detected. Finer-grained dynamic partitioning approaches have also been investigated. Dybdahl proposed a dynamic algorithm to partition the cache into private and shared regions [15], while Zhao proposed partitioning by dedicating some cache ways to private operation [45]. R-NUCA enables dynamic and simultaneous shared and private, however unlike prior proposals, this is achieved without modification of the underlying cache architecture and without enforcing strict constraints on either the private or shared capacity.

Chang proposed a private organization which steals capacity from neighboring private slices, relying on a centralized structure to keep track of sharing. Liu used bits from the requesting-core ID to select the set of L2 slices to probe first [26], using a table-based mechanism to perform a mapping between the core ID and cache slices. R-NUCA applies a mapping based on the requesting-core ID, however this mapping is performed through boolean operations on the ID without an indirection mechanism. Additionally, prior approaches generally advocate performing lookup through multiple serial or parallel probes or indirection through a directory structure; R-NUCA is able to perform exactly one probe to one cache slice to look up any block or to detect a cache miss.

Zhang advocated the use of a tiled architecture, coupling cache slices to processing cores [43]. Starting with a shared substrate, [43] creates local replicas to reduce access latency, requiring a directory structure to keep track of the replicas. As proposed, [43] wastes capacity because locally allocated private blocks are duplicated at the home node, and offers minimal benefit to workloads with large shared read-write working set which does not benefit from replication. R-NUCA assumes a tiled architecture with a shared cache substrate, but avoids the need for a directory mechanism by only replicating blocks known to be read-only. Zhang improves on the design of [43] by migrating private blocks to avoid wasting capacity at the home node [42], however this design still can not benefit shared data blocks.

Beckmann proposed an adaptive scheme that dynamically adjusts the degree to which the local slice is used for replication of read-only blocks [4]. Unlike [4], R-NUCA is not limited to replicating blocks to a single cache slice, allowing for clusters of nearby slices to share capacity for replication. Furthermore, the heuristics employed in [4] require fine-tuning and adjustment, being highly sensitive to the underlying architecture and workloads, whereas R-NUCA offers a direct ability to smoothly trade off replicated capacity for access latency.

Marty studied the benefits of partitioning a cache for multiple simultaneously-executing workloads [28] and proposed a hierarchical structure to simplify handling of coherence between the workloads. The R-NUCA organization can be similarly applied to achieve run-time partitioning of the cache while still preserving the R-NUCA access latency benefits within each partition.

OS-driven cache placement has been studied in a number of contexts. Sherwood proposed to guide cache placement in software [34], suggesting the use of the TLB to map addresses to cache regions. Tam used similar techniques to reduce destructive interference for multi-programmed workloads [36]. Jin advocated the use of the OS to control cache placement in a shared NUCA cache, suggesting that limited replication is possible through this approach [22]. Cho used the same placement mechanism to partition the cache slices into groups [11]. R-NUCA leverages the work of [22] and [11], using the OS-driven approach to guide placement in the cache. Unlike prior proposals, R-NUCA enables dynamic creation of overlapping clusters of slices without additional hardware, and enables use of these clusters for dedicated to private, replicated private, and shared operation. Fensch advocates the use of OS-driven placement to avoid a cache-coherence mechanism [17]. R-NUCA similarly uses the OS-driven placement to avoid cache-coherence at the L2, however R-NUCA does so without placing strict capacity limitations on replication of read-only blocks or on moving of private data when threads are migrated.

7 Conclusions

Wire delays are becoming the dominant component of on-chip communication; meanwhile, increased device density is driving a rise in on-chip core count and cache capacity, both factors that rely on fast on-chip communication. Although the physical organization of distributed caches permits low-latency access by cores to nearby cache slices, the logical organization of the distributed L2 remains an open research topic. Private L2 organizations offer fast local accesses at the cost of substantially lower effective cache capacity, while statically-interleaved shared organizations offer large capacity at the cost of higher average access latency. Prior research proposes hybrid designs that strike a balance between latency and capacity, but fail to optimize for all accesses, or rely on complex, area-intensive and high-latency lookup and coherence mechanisms.

In this work, we observe that accesses can be classified into distinct classes, where each class is amenable to a different placement policy. Based on this observation, we propose R-NUCA, a novel data placement scheme that optimizes the placement of each access class. By utilizing novel rotational interleaving mechanisms and cluster organizations, R-NUCA offers fast local access while maintaining high aggregate capacity, and simplifies the design of the multicore processor by obviating the need for coherence at the L2 cache. R-NUCA has minimal software and hardware overheads, and improves performance by 17% on average against competing designs, and by 26% at best.

References

[1] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya. Integration challenges and trade-offs for tera-scale architectures. *Intel Technology Journal*, August 2007.

[2] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[4] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive selective replication for CMP caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pages 443–454, 2006.

[5] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, pages 319–330, Washington, DC, USA, 2004.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[7] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006.

[8] G. Chen, H. Chen, M. Haurylau, N. Nelson, P. M. Fauchet, E. G. Friedman, and D. H. Albonesi. Electrical and optical on-chip interconnects in scaled microprocessors. In *IEEE International Symposium on Circuits and Systems*, pages 2514–2517, 2005.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, page 55, Washington, DC, USA, 2003.

[10] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 357–368, Washington, DC, USA, 2005.

[11] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pages 455–468, 2006.

[12] Y. Chou, L. Spracklen, and S. G. Abraham. Store memory-level parallelism optimizations for commercial applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*, pages 183–196, Washington, DC, USA, 2005.

[13] W. J. Dally and C. L. Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, 1986.

[14] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *Proceedings of the Thirteenth International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Washington, DC, USA, 2005.

[15] H. Dybdahl and P. Stenstrom. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Computer Architecture*, pages 2–12, Washington, DC, USA, 2007.

[16] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[17] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *Proceedings of the 14th IEEE Symposium on High-Performance Computer Architecture*, 2008.

[18] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–355–364, Aug. 1991.

[19] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser. Utilizing shared data in chip multiprocessors with the Nahalal architecture. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–10, New York, NY, USA, 2008.

[20] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: limitations and opportunities. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research,* pages 79–87, Asilomar, CA, USA, 2007.

[21] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. pages 31–40, New York, NY, USA, 2005.

[22] L. Jin, H. Lee, and S. Cho. A flexible data to L2 cache mapping approach for future multicore processors. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC'06)*, pages 92–101, New York, NY, USA, 2006.

[23] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. A novel migration-based NUCA design for chip multiprocessors. pages 1–12, Piscataway, NJ, USA, 2008.

[24] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *ACM SIGPLAN Not.*, 37(10):211–222, 2002.

[25] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Mar-Apr 2005.

[26] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the Tenth IEEE Symposium on High-Performance Computer Architecture*, page 176, Washington, DC, USA, 2004.

[27] M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[28] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 46–56, New York, NY, USA, 2007.

[29] J. Merino, V. Puente, P. Prieto, and J. 'Angel Gregorio. SP-NUCA: a cost effective dynamic non-uniform cache architecture. *ACM SIGARCH Computer Architecture News*, 36(2):64–71, 2008.

[30] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 307–318, Oct. 1998.

[31] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian. Leveraging bloom filters for smart search within NUCA caches. In *Proceedings of WCED*, June 2006.

[32] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[33] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net/, 2007 Edition.

[34] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings*

of the 13th Annual International Conference on Supercomputing, pages 155–164, 1999.

[35] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *Proceedings of the Third Workshop on Memory Performance Issues (WMPI-2004)*, June 2004.

[36] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[37] B. Towles and W. J. Dally. Route packets, net wires: On-chip interconnection networks. *Design Automation Conference*, 0:684–689, 2001.

[38] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):266–277, 2007.

[39] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, Jul-Aug 2006.

[40] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation through rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[41] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*, pages 237–248, New York, NY, USA, 2005.

[42] M. Zhang and K. Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical report, MIT, 2005.

[43] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, 2005.

[44] Z. Zhang and J. Torrellas. Reducing remote conflict misses: Numa with remote cache versus coma. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, page 272, Washington, DC, USA, 1997.

[45] L. Zhao, R. Iyer, M. Upton, and D. Newell. Towards hybrid last-level caches for chip-multiprocessors. *SIGARCH Computer Architecture News*, 36(2):56–63, 2008.