Appears in Proceedings of the 36th Annual International Symposium on Computer Architecture, Austin, TX, June 2009

Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches

Nikos Hardavellas[1,2], Michael Ferdman[1,3], Babak Falsafi[3] and Anastasia Ailamaki[4,2]
[1]Computer Architecture Lab (CALCM), Carnegie Mellon University
[2]Computer Science Department, Carnegie Mellon University
[3]Parallel Systems Architecture Lab (PARSA), Ecole Polytechnique Fédérale de Lausanne
[4]Data-Intensive Application and Systems Lab (DIAS), Ecole Polytechnique Fédérale de Lausanne

ABSTRACT

Increases in on-chip communication delay and the large working sets of server 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, but they address the placement requirements of only a subset of the data accessed by the application, require complex 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 classes, where each class is amenable to different block placement policies. Based on this observation, we propose Reactive NUCA (R-NUCA), a distributed cache design which reacts to the class of each cache access and places blocks at the appropriate location in the cache. R-NUCA 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. In a range of server, scientific, and multi-programmed workloads, R-NUCA matches the performance of the best cache design for each workload, improving performance by 14% on average over competing designs and by 32% at best, while achieving performance within 5% of an ideal cache design.

Categories & Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles – *cache memories, interleaved memories, shared memory*

General Terms

Design, Experimentation, Performance

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, and major manufacturers already ship 8-core chip multiprocessors [25] with plans to scale to 100s of cores [1,32]. Specialized vendors already push the envelope further, with Cisco CRS-1 featuring 188 processing cores, and Azul Vega 3 with 54 out-of-order cores. The exponential increase in the number of cores results in a commensurate increase in the on-chip cache size required to supply all these cores with data. At the same time, physical and manufacturing considerations suggest that future processors will be tiled: the last-level on-chip cache (LLC) will be decomposed into smaller *slices,* and groups of processor cores and cache slices will be physically distributed throughout the die area [1,43]. Tiled architectures give rise to varying access latencies between the cores and the cache slices spread across the die, naturally leading to a Non-Uniform Cache Access (NUCA) organization of the 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 results 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].

From an access-latency perspective, an LLC organization where each core treats a nearby LLC slice as a private cache is desirable. Although a private organization results in fast local hits, it requires area-intensive, slow and complex mechanisms to guarantee the coherence of shared data, which are prevalent in many multicore workloads [3,20]. At the same time, the growing application working sets render private caching designs impractical due to the inefficient use of cache capacity, as cache blocks are independently replicated in each private cache slice. At the other extreme, a shared 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 LLC does not require a coherence mechanism, enabling a simple design and allowing for larger aggregate cache capacity. However, static interleaving results in a random distribution of cache blocks across the LLC slices, leading to frequent accesses to distant cache slices and high average access latency.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

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

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 [4,7,10]. The placement decisions in R-NUCA guarantee that each modifiable block is mapped to a single location in the aggregate cache, obviating the need for complex, area- and power-intensive coherence mechanisms of prior proposals [4,7,10,43]. R-NUCA utilizes *Rotational Interleaving*, a novel lookup mechanism that matches the fast speed of address-interleaved lookup, without pinning blocks to a single location in the cache [10,43]. Rotational interleaving allows read-only blocks to be shared by neighboring cores and replicated at distant ones, ensuring low access latency while balancing capacity constraints.

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 block placement in distributed caches.
- We propose rotational interleaving, a novel mechanism for fast nearest-neighbor lookup with one cache probe, enabling replication without wasted space and without coherence overheads.
- Through full-system cycle-accurate simulation of multicore systems, we show that R-NUCA provides performance stability across workloads, matching the performance of the best cache design for each workload. R-NUCA attains a maximum speedup of 32%, and an average speedup of 14% over the private design (17% for server workloads) and 6% over the shared design (15% for multi-programmed workloads), while achieving performance within 5% of an ideal cache design.

The rest of this paper is organized as follows. Section 2 presents background on distributed caches and tiled architectures. Section 3 presents our classification and offers a detailed empirical analysis of the cache-access patterns of server, scientific, and multi-programmed workloads. We describe R-NUCA in Section 4 and evaluate it in Section 5. We summarize prior work in Section 6 and conclude in Section 7.

2. BACKGROUND

2.1 Non-Uniform Cache Architectures

The exponential increase in the cache sizes of multicore processors (CMPs) renders uniform access latency impractical, as capacity increases also increase access latency [20]. To mitigate the rising access latency, recent research [24] advocates decomposing the cache into slices. Each slice may consist of multiple banks to optimize for low access latency within the slice [5], and all slices are physically distributed on the die. Thus, cores realize fast accesses to nearby slices and slower accesses to physically distant ones.

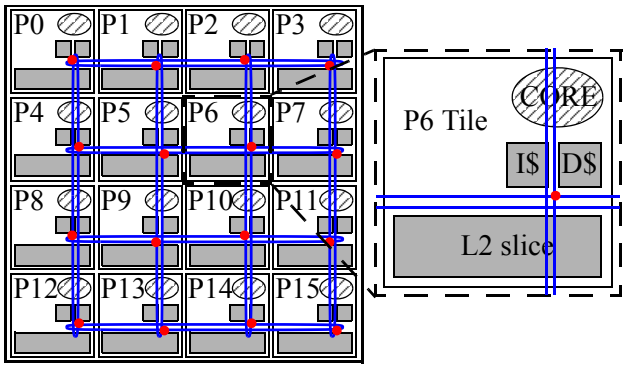


FIGURE 1. Typical tiled architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.

Just as cache slices are distributed across the entire die, processor cores are similarly distributed. Economic, manufacturing, and physical design considerations [1,43] suggest *tiled architectures*, with cores coupled together with cache slices in tiles that communicate via an on-chip interconnect. 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]. They are also economically attractive, as they can easily support families of products with varying number of tiles and power/cooling requirements. Finally, their scalability to high core counts make them suitable for large-scale CMPs.

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. Each L1 cache miss probes an on-chip L2 cache slice via an on-chip network that interconnects the tiles. 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.

Private L2 organization. Each tile's L2 slice serves as a private second-level cache for the tile's core. Upon an L1 miss, the L2 slice located in the same tile is probed. On a write miss in the local L2 slice, the coherence mechanism (a network broadcast or access to an address-interleaved distributed directory) invalidates all other on-chip copies. On a read miss, the coherence mechanism either confirms that a copy of the block is not present on chip, or it obtains the data from an existing copy. With a directory-based coherence mechanism, a typical coherence request is performed in three network traversals. A similar request in token-coherence [27] requires a broadcast followed by a response from the farthest tile.

Enforcing coherence requires large storage and complexity overheads. For example, a full-map directory for a 16-tile CMP with 64-byte blocks, 1MB L2 slices, and 64KB split I/D L1 caches requires 288K directory entries, assuming two separate hardware

structures to keep the L1 caches and the L2 slices coherent. With a 42-bit physical address space, and a 16-bit sharers bit-mask and 5-bit state per block to account for intermediate states, the directory size per tile is 1.2MB, exceeding the L2 capacity. Thus, full-map directories are impractical for the private L2 organization. Limited-directory mechanisms are smaller, but may require complex, slow, or non-scalable fall-back mechanisms such as full-chip broadcast. In this work, we optimistically assume a private L2 organization where each tile has a full-map directory with zero area overhead.

Shared L2 organization. Cache blocks are address-interleaved among the slices, which service requests from any tile through the interconnect. On an L1 cache miss, the miss address dictates the slice responsible for caching the block, and a request is sent directly to that slice. The target slice stores both the block and its coherence state. Because each block has a fixed, unique location in the aggregate L2 cache, the coherence state must cover only the L1 cache tags; following the example above, a full-map directory for a shared L2 organization requires only 152KB per tile.

2.3 Requirements for Intelligent Block Placement

A distributed cache presents a range of latencies to each core, from fast access to nearby slices, to several times slower access to slices on the opposite side of the die. Intelligent cache block placement can improve performance by placing blocks close to the requesting cores, allowing fast access.

We identify three key requirements for intelligent block placement in distributed caches. First, the block address must be decoupled from its physical location, allowing to store the block at a location independent of its address [9]. Decoupling the physical location from the block address complicates lookup on each access. Thus, the second requirement for intelligent block placement is a lookup mechanism capable of quickly and efficiently locating the cached block. Finally, intelligent block placement must optimize for all accesses prevalent in the workload. A placement policy may benefit 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 the cache access patterns using trace-based and cycle-accurate full-system simulation in *FLEXUS* [39] of a tiled CMP executing unmodified applications and operating systems. FLEXUS extends the *Virtutech Simics* functional simulator with timing models of processing tiles with out-of-order cores, NUCA cache, on-chip protocol controllers, and on-chip interconnect. We simulate a tiled CMP similar to Section 2.2, where the LLC is the L2 cache. We summarize our tiled architecture parameters in Table 1 (left).

Future server workloads are likely to run on CMPs with many cores [14], while multi-programmed desktop workloads are likely to run on CMPs with fewer cores that free die area for a larger on-chip cache [7]. We run our multi-programmed mix on an 8-core tiled CMP, and the rest of our workloads on a 16-core tiled CMP. To estimate the L2 cache size for each configuration, we assume a die size of 180mm^2 in 45nm technology and calculate component sizes following ITRS guidelines [33]. We account for the area of the system-on-chip components, allocating 65% of the die area to the tiles [14]. We estimate the area of the cores by scaling a micrograph of the Sun UltraSparc II processor. We model the 16-core configuration with 1MB of L2 cache per core, and the 8-core configuration with 3MB of L2 cache per core.

We simulate systems running the *Solaris 8* operating system and executing the workloads listed in Table 1 (right). We include a wide range of server workloads (online transaction processing, decision support systems, web server), a multiprogrammed workload, and one scientific application as a frame of reference. 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 (TPC-C on DB2 and Oracle, TPC-H queries 6, 8, 11, 13, 16, and 20 on DB2, SPECweb on Apache and Zeus), scientific workloads (em3d, moldyn, ocean, sparse), and the multi-programmed workload from Table 1.

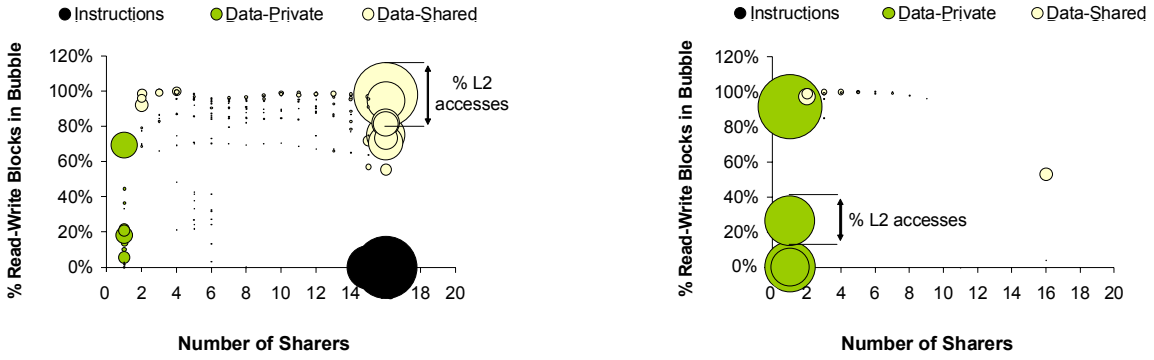
3.2 Categorization of Cache Accesses

We analyze the L2 accesses at the granularity of cache blocks along two axes: the number of cores sharing an L2 block and the percentage of blocks with at least one write request during the workload's execution (*read-write blocks*). Each bubble in Figure 2 represents blocks with the same number of sharers (1-16). For each workload, we plot two bubbles for each number of sharers, one for instruction and one for data accesses. The bubble diameter is proportional to the number of L2 accesses. We indicate instruction accesses in black and data accesses in yellow (shared) or green (private), drawing a distinction for private blocks (accessed by only one core).

TABLE 1. System and application parameters for the 8-core and 16-core CMPs.

CMP Size	16-core for server and scientific workloads 8-core for multi-programmed workloads
Processing Cores	UltraSPARC III ISA; 2GHz, OoO cores 8-stage pipeline, 4-wide dispatch/retirement 96-entry ROB and LSQ, 32-entry store buffer
L1 Caches	split I/D, 64KB 2-way, 2-cycle load-to-use, 3 ports 64-byte blocks, 32 MSHRs, 16-entry victim cache
L2 NUCA Cache	16-core CMP: 1MB per core, 16-way, 14-cycle hit 8-core CMP: 3MB per core, 12-way, 25-cycle hit 64-byte blocks, 32 MSHRs, 16-entry victim cache
Main Memory	3 GB memory, 8KB pages, 45 ns access latency
Memory Controllers	one per 4 cores, round-robin page interleaving
Interconnect	2D folded torus (4x4 for 16-core, 4x2 for 8-core) 32-byte links, 1-cycle link latency, 2-cycle router

OLTP – Online Transaction Processing (TPC-C v3.0)	
DB2	*IBM DB2 v8 ESE,* 100 warehouses (10 GB), 64 clients, 2 GB buffer pool
Oracle	*Oracle 10g Enterprise Database Server* 100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server (SPECweb99)	
Apache	*Apache HTTP Server v2.0.* 16K connections, fastCGI, worker threading model
DSS – Decision Support Systems (TPC-H)	
Qry 6, 8, 13	*IBM DB2 v8 ESE*, 480 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 inputs



(a) Server Workloads

(b) Scientific and Multi-programmed Workloads

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

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

The axes of Figure 2 suggest an appropriate L2 placement policy for each access class. Private data blocks are prime candidates for allocation near the requesting tile; placement at the requesting tile achieves the lowest possible access latency. Because private blocks are always read or written by the same core, coherence is guaranteed without requiring a hardware mechanism. Read-only universally-shared blocks (e.g., instructions) are prime candidates for replication across multiple tiles; replicas allow the blocks to be placed in the physical proximity of the requesting cores, while the blocks' read-only nature obviates coherence. Finally, read-write blocks with many sharers (shared data) may benefit from migration or replication if the blocks exhibit reuse at the L2. However, migration requires complex lookup and replication requires coherence enforcement, and the low reuse of shared blocks does not justify such complex mechanisms (Section 3.3.3). Instead, shared blocks benefit from intelligent block placement on chip.

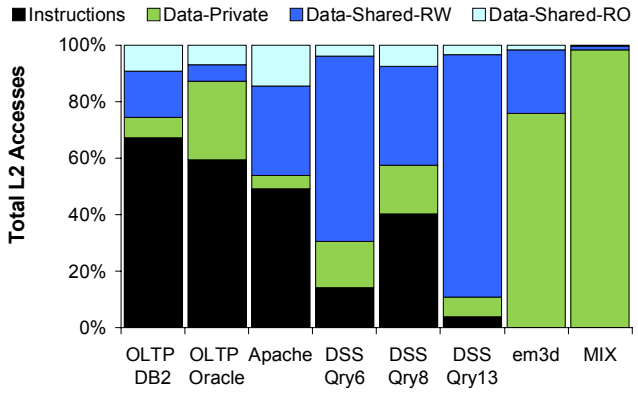


FIGURE 3. L2 Reference breakdown. Distribution of L2 references by access class.

Although server workloads are dominated by accesses to instructions and shared read-write data, a significant fraction of L2 references are to private blocks (Figure 3). The scientific and multi-programmed workloads are dominated by accesses to private data, but also exhibit some shared data accesses. The varying importance of the cache accesses categories underscores a need to react to the access class when placing blocks at L2, and emphasizes the opportunity loss of addressing only a subset of the access classes.

3.3 Characterization of Access Classes

3.3.1 Private Data

Accesses to private data, such as stack space and thread-local storage, are always initiated by the same processor core. As a result, replicating private data at multiple locations on chip only wastes cache capacity [42]. Although some private data are read-write, having only one requestor eliminates the need for a cache coherence mechanism for private blocks. Therefore, the only requirement for private data is to be placed close to its requestor, ensuring low access latency[1]. R-NUCA places private data into the L2 slice at the tile of the requesting core, ensuring minimum latency.

Although the private-data working set for OLTP and multi-programmed workloads (Figure 4, left) may fit into a single, local L2 slice, DSS workloads scan multi-gigabyte database tables and scientific workloads operate on large data sets, both exceeding any reasonable L2 capacity. To accommodate large private data working sets, prior proposals advocate migrating (spilling) these blocks to neighbors [11]. Spilling may be applicable to multi-programmed workloads composed of applications with a range of private-data working set sizes; however, 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 experiencing similar capacity pressure. Migrating private data blocks to a neighboring slice is offset by the neighboring tiles undergoing an identical operation and spilling in the opposite direction. Thus, cache pressure remains the same, but requests incur higher 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.

1. The operating system may migrate a thread from one core to another. In these cases, coherence can be enforced by the OS by shooting down the private blocks upon a thread migration.

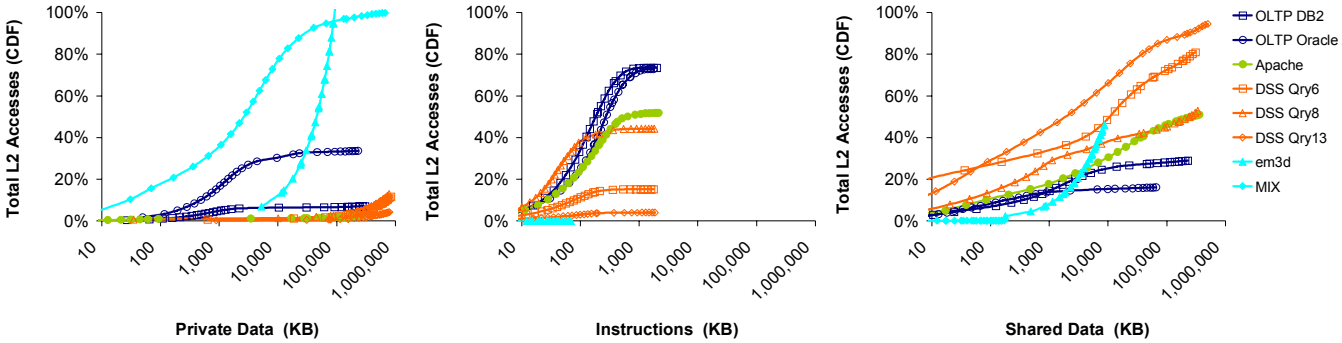


FIGURE 4. L2 working set sizes. CDF of L2 references to private data, instructions, and shared data vs. the footprint of each access class (in log-scale). References are normalized to total L2 references for each workload.

Once in memory, instruction blocks remain read-only for the duration of execution. Figure 2 indicates that instruction blocks in server workloads 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 on chip, 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 L2 accesses which constitute the 1st, 2nd, and subsequent instruction-block accesses by one core without intervening L2 accesses for the same block by a different core. The grey and higher portions of the bars represent reuse accesses that could experience a lower latency if the instruction block was migrated to the requesting core after the first access. The results indicate that accesses to L2 instruction blocks are finely interleaved between participating sharers, with minimal opportunity of instruction block migration. On the contrary, migration may reduce performance, as it increases contention in the on-chip network.

Figure 4 (middle) shows that the instruction working set size for some workloads approximates the size of a single L2 slice. Indiscriminate replication of the instruction blocks at each slice creates too many replicas and increases the capacity pressure and the off-chip miss rate. At the same time, replicating a block in adjacent L2 slices offers virtually no latency benefit, as multiple replicas are one network hop away from a core, while having just one copy nearby is enough. Thus, replication should be done at a coarser granularity: R-NUCA logically divides the L2 into clusters of neighboring slices, replicating instructions at the granularity of a cluster rather than in individual L2 slices. While an application's working set may not fit comfortably in an individual L2 slice, it 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 is possible to smoothly trade off instruction-block access latency for cache capacity: many small clusters provide low access latency while consuming a large fraction of the capacity of each participating slice; a few large clusters result in higher access latency but with a small number of replicas.

For our system configurations and workloads, clusters of 4 slices are appropriate. Clusters of size 4 ensure that instruction blocks are at most one network hop away from the requesting core while storing only a quarter of the instruction working set at each slice.

3.3.3 Shared Data

Shared data comprise predominantly read-write blocks containing data and synchronization structures. 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, on each write, complex coherence mechanisms are necessary to invalidate the replicas or to update the migrating blocks. Figure 5 (right) shows the number of L2 accesses to a shared data block issued by the same core between consecutive writes by other cores. In most cases, a core accesses a block only once or twice before a write by another core. Thus, an invalidation will occur nearly after each replication or migration opportunity, eliminating the possibility of accessing the block at its new location in most cases, and rendering both techniques ineffective for shared data.

Not only do the access characteristics shown in Figure 5 (right) indicate a small opportunity for the replication or migration of shared data, the complexity and overheads of these mechanisms entirely overshadow their benefit. The replication or migration of shared data blocks at arbitrary locations on chip require the use of directory- or broadcast-based mechanisms for lookup and coherence enforcement, as each block is likely to have different placement requirements. However, to date, there have been only few promising directions to provide fast lookup [31], while the area and latency overheads of directory-based schemes (Section 2.2) discourage their use, and broadcast-based mechanisms do not scale due to the bandwidth and power overheads of probing multiple cache slices per access. Also, replicating or migrating shared data

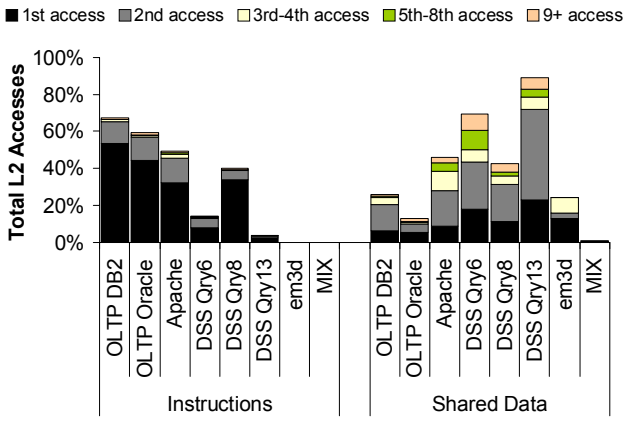


FIGURE 5. Instruction and shared data reuse. Reuse of instructions and shared data by the same core.

would increase the cache pressure and the off-chip requests due to the shared data's large working sets (Figure 4, right).

Placing shared read-write data in a NUCA cache presents a challenging problem because their coherence requirements, diverse access patterns, and large working sets render migration and replication policies undesirable for 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 their adverse behavior is detected [10].

Instead of relying to migration or replication, R-NUCA places the shared read-write data close to the requestors by distributing them evenly among all participating sharers. Shared data blocks in server workloads are universally accessed (Figure 2), with every core having the same likelihood to be the next accessor [35]. Therefore, R-NUCA distributes shared data across all tiles using standard address interleaving. By placing the blocks at the address-interleaved locations, R-NUCA avoids replication. Thus, it eliminates wasted space and obviates 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. At the same time, R-NUCA utilizes a trivial and fast lookup mechanism, as a block's address uniquely determines its location. Because the access latency 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) access latency for all cores (e.g., torus).

3.4 Characterization Conclusions

The diverse cache access patterns of server workloads make each access class amenable to a different placement policy. R-NUCA is motivated by this observation, and its design is guided by the characteristics of each class. More specifically, we find that:

- An intelligent placement policy is sufficient to achieve low access latency for the major access classes.
- L2 hardware coherence mechanisms in a tiled CMP running server workloads are unnecessary and should be avoided.
- Private blocks should be placed in the local slice of the requesting core.
- Instruction blocks should be replicated in clusters (groups) of nearby slices.
- Shared data blocks should be placed at fixed address-interleaved cache 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 not a limitation, as the mechanisms we describe apply to alternative organizations, for example, groups of cores assigned to a single L2 slice.

Conceptually, R-NUCA 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 a center tile and the neighboring tiles around it. 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. Data within each cluster are interleaved among the participating L2 slices, and shared among all cores participating in that cluster.

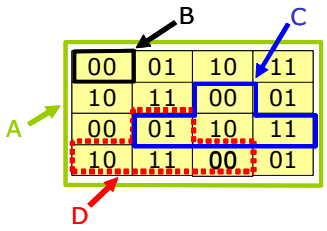


FIGURE 6. Example of R-NUCA clusters and Rotational Interleaving. The array of rectangles represents the tiles. The binary numbers in the rectangles denote each tile's RID. The lines surrounding some of the tiles are cluster boundaries.

4.1 Indexing and Rotational Interleaving

R-NUCA indexes blocks within each cluster using either standard address interleaving or rotational interleaving. In standard address interleaving, an L2 slice is selected based on the bits immediately above the set-index bits of the accessed address. In rotational interleaving, each core is assigned a *rotational ID* (RID) by the operating system. The RID is different from the conventional *core ID* (CID) that the OS assigns to each core for process bookkeeping.

RIDs in a size-*n* cluster range from *0* to *n-1*. To assign RIDs, the OS first assigns the RID *0* to a random tile. Consecutive tiles in a row receive consecutive RIDs. Similarly, consecutive tiles in a column are assigned RIDs that differ by $log_2(n)$ (along rows and columns, *n-1* wraps 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 bits with its own RID $<c_1, c_0>$ using a boolean function; the outcome of the comparison determines which slice caches the accessed block[1]. In the example of Figure 6, when the center core of cluster C accesses a block with address bits *<0, 1>*, the core evaluates the indexing function and access the block in the slice to its left. Similarly, when the center core of cluster D accesses the same block, the indexing function indicates that the block is at the slice above. Thus, each slice stores exactly the same *1/n*-th of the data on behalf of any cluster to which it belongs.

Rotational interleaving allows clusters to replicate data without increasing the capacity pressure in the cache, and at the same time enable fast nearest-neighbor communication. The implementation of rotational interleaving is trivial, requiring only that tiles have RIDs and that indexing is performed through simple boolean logic on the tile's RID and the block's address. Although for illustration

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

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

For size-4 clusters, the 2-bit result *R* indicates that the block is in, to the right, above, or to the left of the requesting tile, for binary results *<0,0>*, *<0,1>*, *<1,0>* and *<1,1>* respectively.

purposes we limit our description to size-4 clusters, rotational interleaving is simply generalized to clusters of any power-of-two.

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 cluster and places the block according to the appropriate interleaving mechanism for 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 minimal access latency. Shared data blocks are placed in size-16 clusters which are fully overlapped 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 classifies 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 L1 instruction caches are immediately classified as *instructions* and a lookup is performed on the size-4 fixed-center cluster centered at the requesting core. All other requests are classified as data requests, and the OS is responsible for distinguishing between private and shared data accesses.

To communicate the 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 requesting core examines the Private bit and looks for the block only in its own local L2 slice.

On a subsequent TLB miss to the page, 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, either the thread accessing this page has migrated to another core and the page is still private to the thread, or the page is shared by multiple threads and 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 a 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. TLB misses for this page by other cores are delayed until the poisoned state is cleared. Once the *Poisoned* bit is set, the OS shoots down the TLB entry and invalidates any cache blocks belonging to the page at the previous accessor's tile.[1]

When the shoot-down completes, the OS classifies the page as shared by clearing the Private bit in the page table entry, removes the poisoned state, and services any pending TLB requests. Because the Private bit is cleared, any core that receives a TLB entry will treat accesses to this page as shared, applying the standard address interleaving over the size-16 cluster (entire aggregate cache) to locate the shared block.

If a page is private but the thread has migrated from one core to another, a procedure similar to re-classification is employed. The only difference being that after invalidation at the previous accessor, the page retains its private classification, and the CID in the page table entry is updated to the CID of the new owner.

4.4 Extensions

Although 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 CMP 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 use standard address interleaving or rotational interleaving. The choice of interleaving depends on the block replication requirements. Rotational interleaving is appropriate for replicating blocks while balancing capacity constraints. Standard address interleaving is appropriate for disjoint clusters. By designating a *center* for a cluster and communicating it to the cores via the TLB mechanism in addition to the Private bit, both interleaving mechanisms are possible for any cluster type of any size.

Our configuration of R-NUCA employs fixed-center clusters only for instructions; however, alternative configurations are possible. For example, heterogeneous workloads with different private data capacity requirements for each thread (e.g., multi-programmed workloads) may favor 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.

5. EVALUATION

5.1 Methodology

For both CMP configurations, we evaluate four NUCA designs: private (P), ASR (A), shared (S), and R-NUCA (R). The shared and private designs are described in Section 2.2. ASR [4] is based on the private design and adds an adaptive mechanism that probabilistically allocates clean shared blocks in the local L2 slice upon their eviction from L1. If ASR chooses not to allocate a block, the

1. Block invalidation at the previous accessor is required to guarantee coherence when transitioning from a private to shared classification. The invalidation can be performed by any shoot-down mechanism available to the OS, such as scheduling a special shoot-down kernel thread at the previous accessor's core; instructions or PAL code routines to perform this operation already exist in many of today's architectures.

block is cached at an empty cache frame in another slice, or dropped if no empty L2 frame exists or another replica is found on chip. ASR was recently shown to outperform all prior proposals for on-chip cache management in a CMP[4].

Although we did a best-effort implementation of ASR, our results did not match with [4]. We believe that the assumptions of our system penalize ASR, while the assumptions of [4] penalize the shared and private cache designs. 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 six versions of ASR: an adaptive version following the guidelines in [4], and five versions that statically choose to allocate an L1 victim at the local slice with probabilities 0, 0.25, 0.5, 0.75 and 1, respectively. In our results for ASR, we report, for each workload, the highest-performing of these six versions.

For the private and ASR designs, we optimistically assume an on-chip full-map distributed directory with zero area overhead. In reality, a full-map directory occupies more area than the aggregate L2 cache, and yet-undiscovered 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 mechanisms incur no area overhead. Thus, the speedup of R-NUCA compared to a realistic implementation of the private or ASR designs will be higher than reported in this paper.

Our on-chip coherence protocol is a four-state MOSI protocol modeled after Piranha [2]. The cores perform speculative load execution and store prefetching [12,18,30]. We simulate one memory controller per four cores, each controller co-located with one tile, assuming communication with off-chip memory through flip-chip technology. Tiles communicate through the on-chip network. We list other relevant parameters in Table 1 (left).

We simulate a 2-D folded torus [13] on-chip interconnection network. While prior research typically utilizes mesh interconnects due to their simple implementation, meshes are prone to hot spots and penalize tiles at the network edges. In contrast, torus interconnects have no edges and treat nodes homogeneously, spreading the traffic across all links and avoiding hot spots. 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 of its dimensions is planar, requiring only two metal layers for the interconnect [37]. With current commercial products already featuring 11 metal layers, and favorable comparisons of tori against meshes with respect to area and power overheads [37], we believe 2-D torus interconnects are a feasible and desirable design point.

We measure performance using the SimFlex multiprocessor sampling methodology [39]. Our samples are drawn over an interval of 10 to 30 seconds for OLTP and web server applications, the complete query execution for DSS, one complete iteration for the scientific application, and the first 10 billion instructions after the start of the first main-loop iteration for MIX. We launch measurements from checkpoints with warmed caches, branch predictors, TLBs, on-chip directories, and OS page tables, 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].

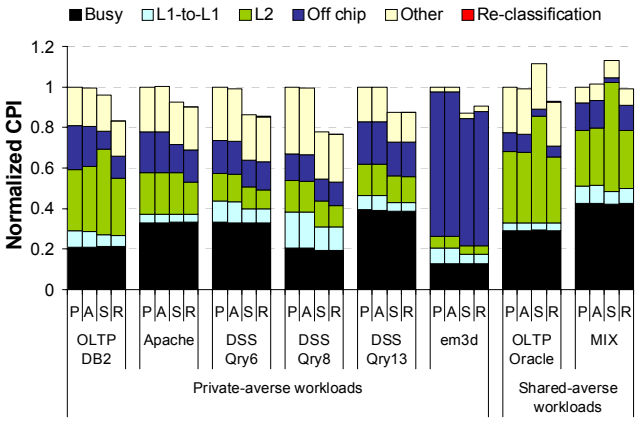


FIGURE 7. Total CPI breakdown for L2 designs. The CPI is normalized to the total CPI of the private design.

5.2 Classification Accuracy

Although in Section 3 we analyzed the workloads at the granularity of cache blocks, R-NUCA performs classification at page granularity. Pages may simultaneously contain blocks of multiple classes; part of a page may contain private data, while the rest may contain shared data. For our workloads, 6% to 26% of L2 accesses are to pages with more than one class. However, the accesses issued to these pages are dominated by a single class; if a page holds both shared and private data, accesses to shared data dominate. Classifying these pages as *shared-data* effectively captures the majority of accesses. Overall, classification at page granularity results in the misclassification of less than 0.75% of L2 accesses.

5.3 Impact of R-NUCA Mechanisms

Because different workloads favor a different cache organization, we split our workloads into two categories: private-averse and shared-averse, based on which design has a higher cycles-per-instruction (CPI). The private design may perform poorly if it increases the number of off-chip accesses, or if there is a large number of L1-to-L1 or L2 coherence requests. Such requests occur if a core misses in its private L1 and L2 slice, and the data are transferred from a remote L1 (*L1-to-L1 transfer*) or a remote L2 (*L2 coherence transfer*). The private and ASR designs penalize such requests, because each request accesses first the on-chip distributed directory, which forwards 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 accesses to L2 slices. Similarly, the shared design may perform poorly if there are many accesses to private data or instructions, which the shared design spreads across the entire chip, while the private design services through the local and fast L2 slice.

Figure 7 shows the CPI (normalized to the private design) due to useful computation (busy), L1-to-L1 transfers, L2 loads and instruction fetches (L2), off-chip requests, other delays (e.g., front-end stalls), and the CPI due to R-NUCA page re-classifications. We account for store latency in *other* due to the existence of recent proposals to minimize store latency [6,38]. Figure 7 confirms that the re-classification overhead of R-NUCA is negligible. R-NUCA outperforms the competing designs, lowering the L2 hit latency exhibited by the shared design, and eliminating the long-latency coherence operations of the private and ASR designs.

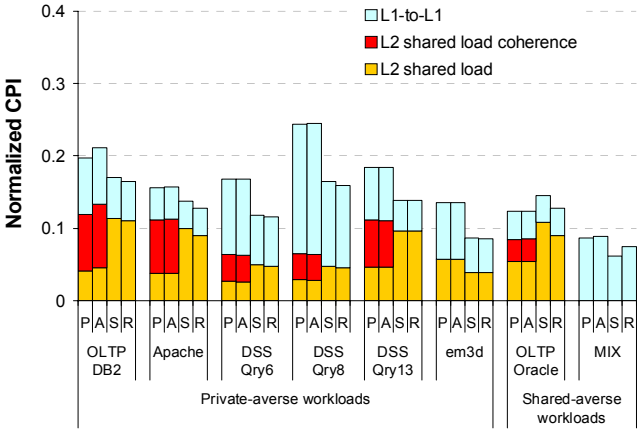


FIGURE 8. CPI breakdown of L1-to-L1 and L2 load accesses. The CPI is normalized to the total CPI of the private design.

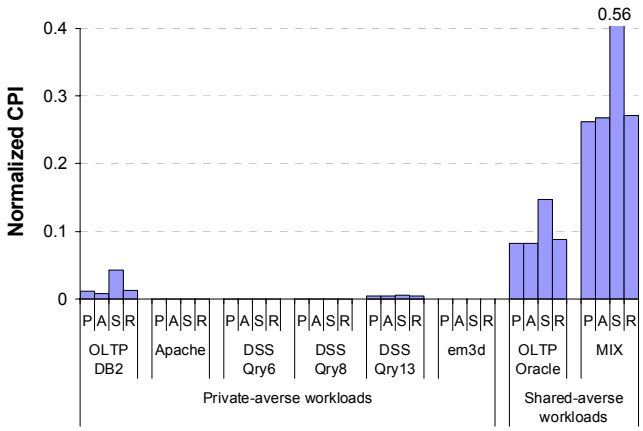


FIGURE 9. CPI contribution of L2 accesses to private data. The CPI is normalized to the total CPI of the private design.

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 in the shared design are interleaved across all L2 slices, with both designs having equal latency. The private and ASR designs replicate blocks, alternating between servicing requests from the local slice (*L2 shared load*) or a remote slice (*L2 shared load coherence*). Although local L2 slice accesses are fast, remote accesses engage the on-chip coherence mechanism, requiring an additional network traversal and two additional tile accesses compared to the shared or R-NUCA designs. Thus, the benefits of fast local reuse for shared data under the private and ASR designs are quickly outweighed by long-latency coherence operations. On average, eliminating L2 coherence requests in R-NUCA results in 18% lower CPI contribution of accesses to shared data. Similarly, the private and ASR designs require accessing both local and remote L2 slices to complete an L1-to-L1 transfer, whereas the shared and R-NUCA designs use only one access to an L2 slice before requests are sent to an L1. By eliminating the additional remote L2 slice access, R-NUCA lowers the latency for L1-to-L1 transfers by 27% on average. Overall, eliminating coherence requirements at L2 lowers the CPI due to shared data accesses by 22% on average compared to the private and ASR designs.

Impact of local allocation of private data. Similar to the private and ASR designs, R-NUCA allocates private data at the local L2 slice for fast access, while the shared design increases latency by distributing private data across all L2 slices. Figure 9 shows the impact of allocating the private data locally. R-NUCA reduces the access latency of private data by 42% compared to the shared design, matching the performance of the private design.

Impact of instruction clustering. R-NUCA's clustered replication distributes instructions among neighboring slices. Replication ensures that instruction blocks are only one hop away from the requestor, and rotational interleaving ensures fast lookup that matches the speed of a local L2 access. In contrast, the shared design spreads instruction blocks across the entire die area, requiring significantly more cycles for each instruction L2 request (Figure 10). As a result, R-NUCA obtains instruction blocks from L2 on average 40% faster than the shared design. In OLTP-Oracle and Apache, R-NUCA even outperforms the private design by 20%, as the latter accesses remote tiles to fill some requests.

While the private design enables fast instruction L2 accesses, the excessive replication of instruction blocks causes evictions and an increase in off-chip misses. Figure 11 compares the performance of instruction clusters of various sizes. We find that storing instructions only in the local L2 slice (size-1) increases the off-chip CPI component by 62% on average over a size-4 cluster, resulting in reduced performance. At the same time, clusters larger than size-4 spread instruction blocks to a larger area, increasing instruction access latency by 34% to 69% for size-8 and size-16 clusters respectively. We find that, for our workloads and system configurations, size-4 clusters offer the best balance between L2 hit latency and off-chip misses.

5.4 Performance Improvement

R-NUCA lowers the CPI contribution of L2 hits by 18% on average compared to the private design, and by 22% on average compared to the shared design. At the same time, like the shared design, R-NUCA is effective at maintaining the large aggregate capacity of the distributed L2. The CPI due to off-chip misses for R-NUCA is on average within 17% of the shared design's for server workloads, while the private design increases the off-chip CPI by 72%. Thus, R-NUCA delivers both the fast local access of the private design and the large effective cache capacity of the shared design, bridging the gap between the two. R-NUCA attains an average speedup of 14% over the private and 6% over the shared

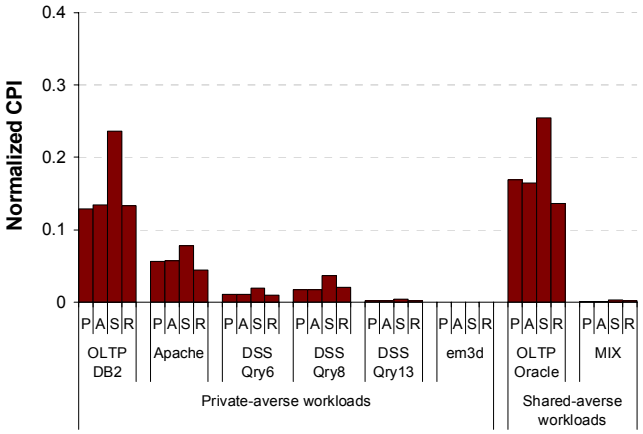


FIGURE 10. CPI contribution of L2 instruction accesses. The CPI is normalized to the total CPI of the private design.

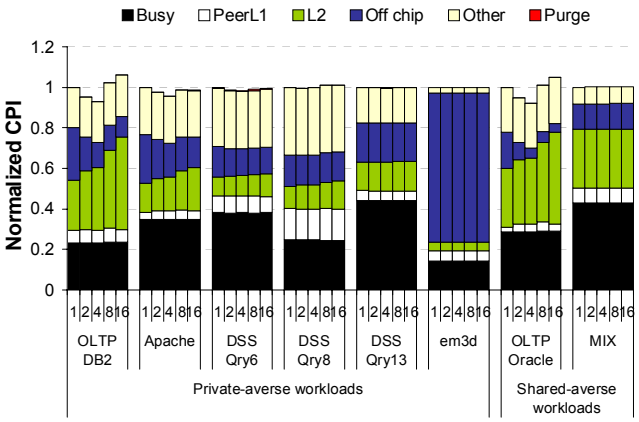


FIGURE 11. CPI breakdown of instruction clusters with various sizes. The CPI is normalized to size-1 clusters.

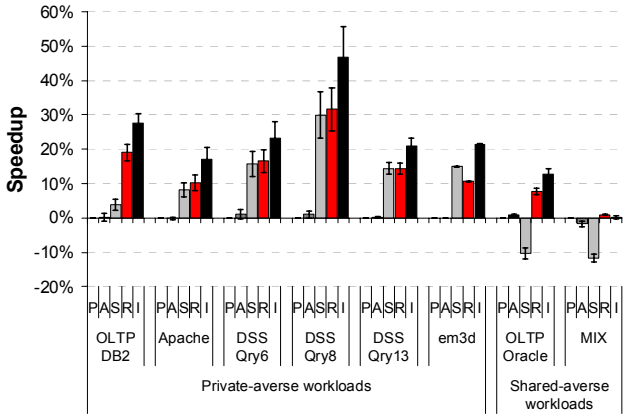


FIGURE 12. Performance Improvement. Speedup is normalized over the private design.

designs, and a maximum speedup of 32%. Figure 12 shows the corresponding speedups, along with the 95% confidence intervals produced by our sampling methodology.

In Figure 12, we also show an *Ideal* design (I) that offers to each core the capacity of the aggregate L2 cache at the access latency of the local slice. The ideal design assumes a shared organization with direct on-chip network links from every core to every L2 slice, where each slice is heavily multi-banked to eliminate contention. We find that R-NUCA achieves near-optimal block placement, as its performance is within 5% of the ideal design.

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 design 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 designs will grow with 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, avoiding the private design'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 design. Finally, we believe that the generality of R-NUCA's clustered organizations will allow for the seamless decomposition of a large-scale multicore processor 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 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. R-NUCA exploits both proposals.

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 for the placement of 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, avoiding the central contention problem and eliminating the need for complex lookup algorithms.

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 a private or shared organization by dynamically adapting the system on a 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 [19] 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 organizations, however, unlike prior proposals, 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 a large shared read-write working set which do 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 design that dynamically adjusts the probability by which read-only shared blocks are allocated at the local slice [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 design 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 the dynamic creation of overlapping clusters of slices without additional hardware, and enables the use of these clusters for dedicated private, replicated private, and shared operation. Fensch advocates the use of OS-driven placement to avoid the cache-coherence mechanism [17]. R-NUCA similarly uses OS-driven placement to avoid cache-coherence at the L2, however, R-NUCA does so without placing strict capacity limitations on the replication of read-only blocks or requiring new hardware structures and TLB ports.

7. CONCLUSIONS

Wire delays are becoming the dominant component of on-chip communication; meanwhile, the increasing device density is driving a rise in on-chip core count and cache capacity, both factors that rely on fast on-chip communication. Although distributed caches permit low-latency accesses by cores to nearby cache slices, the performance benefits depend on the logical organization of the distributed LLC. Private organizations offer fast local access at the cost of substantially lower effective cache capacity, while address-interleaved shared organizations offer large capacity at the cost of higher 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 block placement policy. Based on this observation, we propose R-NUCA, a novel cache design 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 LLC. With minimal software and hardware overheads, R-NUCA improves performance by 14% on average, and by 32% at best, while achieving performance within 5% of an ideal cache design.

8. ACKNOWLEDGEMENTS

The authors would like to thank B. Gold and S. Somogyi for their technical assistance, and T. Brecht, T. Strigkos, and the anonymous reviewers for their feedback on earlier drafts of this paper. This work was partially supported by equipment donations from Intel; a Sloan research fellowship; an ESF EurYI award; and NSF grants CCF-0702658, CCR-0509356, CCF-0845157, CCR-0205544, IIS-0133686 and IIS-0713409.

9. 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, 11(3):173-184, Aug. 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, Jun. 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, Jun. 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, Dec. 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, Dec. 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, Jun. 2007.

[7] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In Proceedings of the 33rd Annual International Symposium on Computer Architecture, Jun. 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 Proceedings of the 2005 IEEE International Symposium on Circuits and Systems, May 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, Dec. 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, Jun. 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, Dec. 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, Nov. 2005.

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

[14] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2005.

[15] H. Dybdahl and P. Stenstrom. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In Proceedings of the 13th International Symposium on High-Performance Computer Architecture, Feb. 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, Jun. 1997.

[17] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In Proceedings of the 14th International Symposium on High-Performance Computer Architecture, Feb. 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), 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, Jun. 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, Jan. 2007.

[21] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In Proceedings of the 19th Annual International Conference on Supercomputing, Jun. 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 ACM SIGPLAN Workshop on Memory System Performance and Correctness, Oct. 2006.

[23] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. A novel migration-based NUCA design for chip multiprocessors. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Nov. 2008.

[24] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 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 10th International Symposium on High-Performance Computer Architecture, Feb. 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, Jun. 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, Jun. 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, May 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, Oct. 1998.

[31] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian. Leveraging bloom filters for smart search within NUCA caches. In Proceedings of the 2006 Workshop on Complexity-Effective Design, Jun. 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. In Proceedings of the ACM SIGGRAPH 2008, Aug. 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, Jun. 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 3rd Workshop on Memory Performance Issues, Jun. 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, Jun. 2007.

[37] B. Towles and W. J. Dally. Route packets, net wires: On-chip interconnection networks. In Proceedings of the 38th Design Automation Conference, 0:684–689, Jun. 2001.

[38] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In Proceedings of the 34th International Symposium on Computer Architecture, Jun. 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.

[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, Sep. 2005.

[42] M. Zhang and K. Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical Report MIT-CSAIL-TR-2005-064, MIT, Oct. 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, Jun. 2005.

[44] Z. Zhang and J. Torrellas. Reducing remote conflict misses: Numa with remote cache versus coma. In Proceedings of the 3rd International Symposium on High-Performance Computer Architecture, Feb. 1997.

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