Near-Optimal Cache Block Placement with Reactive Nonuniform Cache Architectures

THE GROWING CORE COUNTS AND CACHES OF MODERN PROCESSORS RESULT IN DATA ACCESS LATENCY BECOMING A FUNCTION OF THE DATA'S PHYSICAL LOCATION IN THE CACHE. THUS, THE PLACEMENT OF CACHE BLOCKS DETERMINES THE CACHE'S PERFORMANCE. REACTIVE NONUNIFORM CACHE ARCHITECTURES (R-NUCA) ACHIEVE NEAR-OPTIMAL CACHE BLOCK PLACEMENT BY CLASSIFYING BLOCKS ONLINE AND PLACING DATA CLOSE TO THE CORES THAT USE THEM.

Nikos Hardavellas
Northwestern University

Michael Ferdman
Carnegie Mellon University

Babak Falsafi
Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne

As processor manufacturers strive to deliver higher performance within the power and cooling constraints of modern chips, the number of cores on a chip rises exponentially. The exponential increase in cores results in a commensurate increase in the on-chip cache size required to supply the cores with data. However, the growing cache capacity comes at the cost of higher access latency. Although the increasing device switching speeds result in faster cache-bank accesses, the communication delay remains constant across technologies, and wire delays dominate the access latency of far-away cache blocks.[1] As a result, modern workloads spend most of their execution time on on-chip cache accesses.

To mitigate the rising data access latency, recent research advocates dividing the last-level on-chip cache (LLC) into smaller cache slices that are physically distributed throughout the die along with the cores.[2] Such designs naturally form a nonuniform cache architecture (NUCA).[1] A NUCA cache presents a range of latencies to each core, from fast access to nearby slices, to several times slower access to slices on the die's opposite side. Because the cache hit latency depends on the physical distance between the requesting core and the cached data, block placement in the aggregate cache determines the cache's performance.

Ideally, cache blocks are placed close to the requesting cores, allowing fast access. To achieve this flexibility in block placement, the block's address must be decoupled from its physical location (see the sidebar "Related work in cache organization"). However, with the block's address no longer defining its location, the cache needs a lookup mechanism to quickly and efficiently locate a cached block. Moreover, to achieve high performance and robustness, block placement must be optimized for all accesses prevalent in the workload. Unfortunately, data exhibit conflicting placement requirements, precluding a scheme that treats all blocks equally.

Published by the IEEE Computer Society

Related work in cache organization

A private cache organization offers fast local access and minimizes cross-chip communication. However, it requires slow and complex mechanisms to guarantee coherence and wastes cache capacity, leading to frequent off-chip requests. A shared cache organization maximizes the effective cache capacity and minimizes off-chip memory requests, but spreads data across the entire die and leads to high average access latency.

Recent proposals advocate hybrid mechanisms to bridge the gap between private and shared organizations. Proposals based on the private organization typically aim to minimize off-chip requests.[1] Proposals based on the shared organization rely on block migration to bring data closer to the requestors.[2-4] However, prior proposals rely on complex and high-latency lookup and coherence mechanisms, or do not optimize the placement for all block classes. CMP-NuRAPID[2] is the closest to R-NUCA, as it advocates the decoupling of a block's address from its physical location, and follows different placement policies for private, shared read-only, and shared read-write blocks. However, CMP-NuRAPID[2] also relies on complex lookup and coherence mechanisms. Other proposals advocate splitting the cache into private and shared regions,[5] but require modifying the underlying cache architecture, or enforce strict constraints on either the private or shared capacity.

In contrast, R-NUCA enables dynamic and simultaneous shared and private cache organizations, bridging the gap between the two. At the same time, unlike prior proposals, R-NUCA simplifies the cache design by using simple lookup mechanisms and obviating hardware coherence.

References

1. B.M. Beckmann, M.R. Marty, and D.A. Wood, ''ASR: Adaptive Selective Replication for CMP Caches,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 443-454.
2. Z. Chishti, M.D. Powell, and T.N. Vijaykumar, ''Optimizing Replication, Communication, and Capacity Allocation in CMPs,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 357-368.
3. S. Cho and L. Jin, ''Managing Distributed, Shared L2 Caches through OS-Level Page Allocation,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 455-468.
4. J. Huh et al., ''A NUCA Substrate for Flexible CMP Cache Sharing,'' *Proc. 19th Ann. Int'l Conf. Supercomputing* (SC 05), ACM Press, 2005, pp. 31-40.
5. L. Zhao et al., ''Towards Hybrid Last-Level Caches for Chip-Multiprocessors,'' *ACM SIGARCH Computer Architecture News,* vol. 36, no. 2, May 2008, pp. 56-63.

We observe that the cache access patterns in a range of server, multiprogrammed, and scientific workloads can be classified into classes amenable to different block placement policies. Reactive NUCA (R-NUCA) capitalizes on this observation by dynamically classifying blocks and reacting to a block's class to place it near the requesting cores. R-NUCA optimizes block placement for all cache accesses, attaining fast access while simplifying cache design. By placing the cache blocks based on their access class, R-NUCA provides performance robustness across a wide range of workloads and achieves performance close to an ideal cache.

Tiled multicores and NUCA

With both cores and cache slices physically distributed across the die, cores realize fast access to nearby slices and slower access to far away ones. From an economic, manufacturing, and physical design standpoint, it is attractive to organize such a chip as a *tiled architecture*,[2] with cores and cache slices coupled in tiles that communicate via an on-chip interconnect.

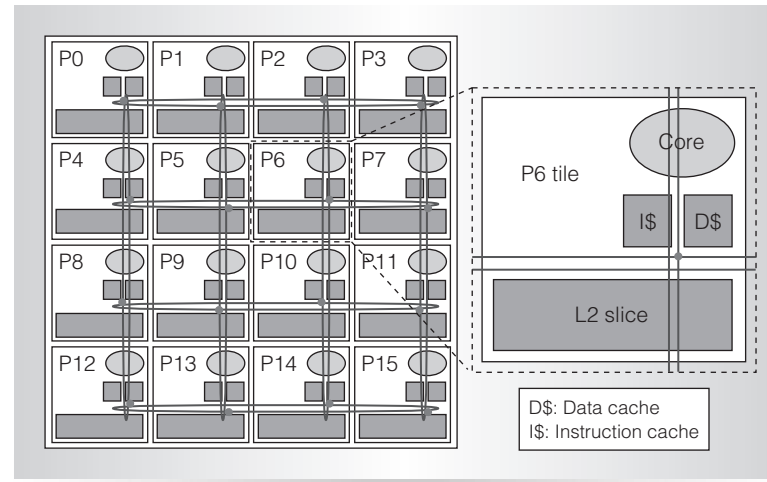

Figure 1. Typical tiled architecture. Each tile contains a core, L1 instruction and data caches, an L2 cache slice, and a router/switch. Tiles are connected through an on-chip interconnect.

Figure 1 presents a typical tiled architecture. Multiple tiles, each comprising a core, caches, and a network router/switch, are replicated to fill the die area. Each tile includes private L1 data and instruction

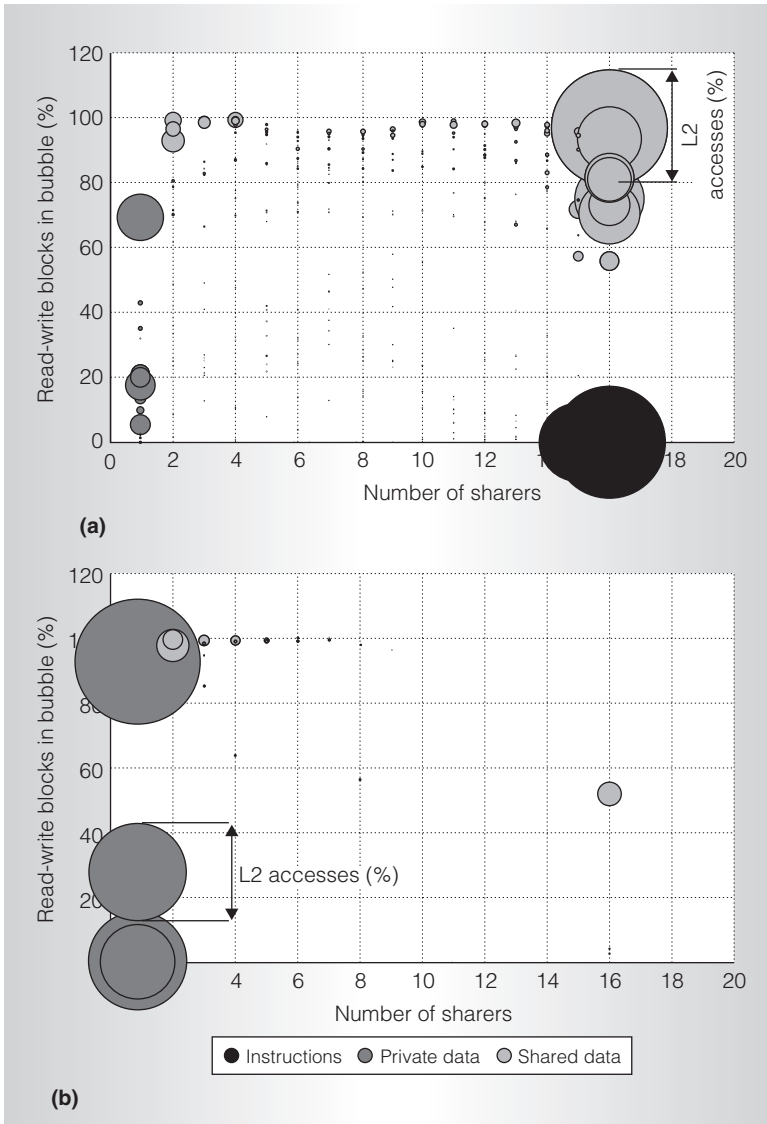

Figure 2. L2 access clustering for server workloads (a) and for scientific and multiprogrammed workloads (b). We categorize accesses to L2 blocks with respect to the blocks' number of sharers, read-write behavior, and instruction or data access request.

caches and an L2 cache slice. Each L1 miss probes an L2 slice via the on-chip interconnect. Depending on the L2 organization, the L2 slice can be a private L2 cache or a portion of a larger distributed shared L2 cache.

In the private L2 organization, each tile's L2 slice serves as a private second-level cache for the tile's core. Coherence among the L2 slices is typically maintained through a distributed directory (address-interleaved among the tiles), with coherence requests usually performed in three network traversals. Upon an L1 miss, the L1 controller probes the L2 slice in the same tile. On a read miss in the local L2 slice, the L2 controller sends a message to the distributed directory that either results in an off-chip miss, or requests an existing sharer to supply the data to the requestor. On a write miss in the local L2 slice, the L2 controller sends a similar message to the directory, which also invalidates any on-chip copies. Enforcing coherence in a private L2 organization typically requires large storage and complexity overheads (for example, a full-map directory might exceed the L2 capacity[3]). Here, we conservatively assume a private L2 organization in which each tile has a full-map directory with zero area overhead.

In the shared L2 organization, the cache blocks together with their coherence directory information are address-interleaved among the L2 slices. The L2 slices service requests from any core, with a typical request serviced in two network traversals. On an L1 miss, the miss address dictates the L2 slice responsible for caching the block, and the L1 controller sends a request directly to that slice, which replies with the data. Because each block has a fixed, unique location in the aggregate L2 cache, the coherence directory must cover only the L1 caches and is significantly smaller than in the private organization.

Characterization and placement of cache blocks

We analyze the cache access patterns of a tiled multicore processor (CMP) using trace-based and cycle-accurate full-system simulation in Flexus.[4] We simulate a tiled CMP with L2 as the LLC, executing a range of unmodified server, multiprogrammed, and scientific workloads. Details of our methodology appear elsewhere.[3]

R-NUCA classifies blocks dynamically and places them in the LLC based on the access class. We classify the blocks as *instructions*, *private data* (accessed by only one core), or *shared data* (accessed by many cores). Figure 2 shows the classification of L2 accesses for our workload suite.

We analyze the accesses along two axes: the number of cores sharing an L2 block, and the percentage of blocks with at least one write during the workload's execution (read-write blocks). Each bubble in Figure 2 represents blocks with the same number of sharers (1 to 16 for our configurations). For each workload, Figure 2 plots two bubbles for each number of sharers, one for instruction accesses 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 light gray (shared) or dark gray (private), drawing a distinction for private blocks.

The cache accesses naturally form three clusters with distinct characteristics:

- private data are accessed by only one core,
- shared data are universally shared by all cores and are mostly read-write, and
- instructions are universally shared and are read-only.

All categories are important, as different workloads issue a varying number of references within each category. At the same time, different classes present conflicting placement requirements, underscoring the need to consider the access class for cache block placement.

The axes of Figure 2 suggest the behaviors that define an appropriate L2 placement policy for each access class. Because private data blocks are always accessed by the same core, allocating them at the requesting tile achieves the lowest possible access latency.

Because shared data blocks are predominantly read-write and are shared by all cores in the system, they can benefit from migration or replication if they exhibit reuse at L2. Replication or migration can provide low-latency access to the same block from the local or nearby cores. However, our workloads typically exhibit low reuse of shared blocks, which does not justify such complex mechanisms.

In most cases, a core accesses a block only once or twice before another core writes to the same block.[3] Thus, an invalidation will occur after almost each replication or migration opportunity, eliminating the possibility of accessing the block at its new location, and rendering both techniques ineffective for shared data. Not only does the low reuse suggest a small opportunity for replicating or migrating shared blocks, but these mechanisms' complexity and overhead overshadow their benefit. Replication requires complex coherence-enforcement mechanisms to invalidate or update the replicas on each write, and migration requires complex lookup to locate the data on each access.

Instead of relying on migration or replication, R-NUCA places the shared read-write data equally close to the requestors by distributing them evenly among all participating sharers using standard address interleaving. By guaranteeing that each read-write block maps to a unique location in the aggregate cache, R-NUCA obviates the need for a hardware coherence mechanism. At the same time, a block's address determines its location, leading to a simple and fast lookup mechanism.

Finally, instruction blocks are typically written once when the operating system loads an application binary or shared library into memory. When in memory, instruction blocks remain read-only throughout the execution. Like shared read-write blocks, instruction blocks in server workloads are universally shared. All cores 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 across multiple cache slices, allowing low-latency access from multiple locations on the chip. At the same time, the blocks' read-only nature obviates coherence.

However, many server workloads have large instruction working sets that do not fit in a single cache slice.[3] Indiscriminate replication of the instruction blocks at each slice increases the cache capacity pressure and the off-chip miss rate. At the same time, replicating a block in adjacent L2 slices offers virtually no latency benefit. Thus, instruction block replication should be done at a coarser granularity: R-NUCA logically divides the L2 into clusters of neighboring slices, replicating instructions across clusters rather

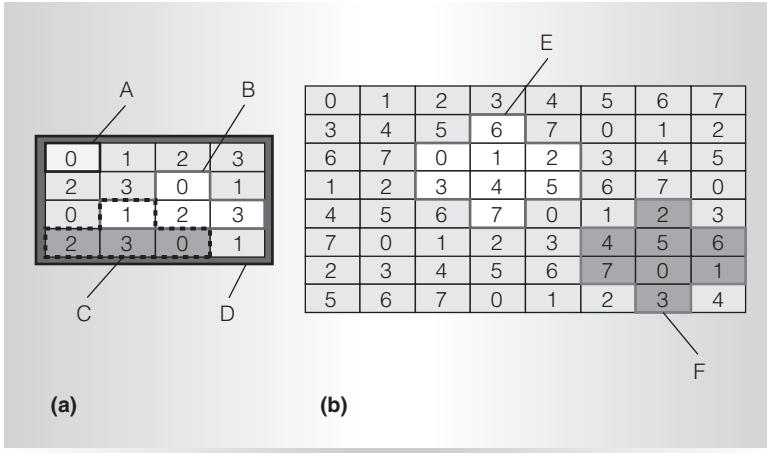

Figure 3. Example of reactive NUCA (R-NUCA) clusters and rotational interleaving for size-1, -4, and -16 fixed-center clusters (a) and for size-8 fixed-center clusters (b). The array of rectangles represents the tiles. The numbers in the rectangles denote each tile's rotational ID (RID). The lines surrounding some of the tiles are cluster boundaries.

than individual slices. The replication rotates across neighboring clusters, so that each slice participating in a size-*n* cluster stores exactly *1*/*n* of the instruction blocks, and adjacent slices never cache the same block.

Reactive NUCA design

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 B and C in Figure 3a each consist of a center tile and its neighboring tiles. Clusters can be of various power-of-2 sizes. Clusters B and C in Figure 3a are size 4. Size-1 clusters always consist of a single tile (for example, cluster A). In our example, size-16 clusters comprise all tiles (for example, cluster D). As the figure shows, clusters can overlap. Data within each cluster are interleaved among the participating L2 slices and shared among all cores participating in that cluster.

Placement

R-NUCA selects the cluster based on the cache block's classification and places the block according to the appropriate interleaving mechanism for this cluster. In our configuration, R-NUCA uses only size-1, size-4, and size-16 clusters. 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 encompass all sharers. R-NUCA allocates instructions in the most size-appropriate fixed-center cluster (size 4 for our workloads), replicating the instructions across clusters on chip. Thus, instructions are shared by neighboring cores and replicated at distant cores, ensuring low access latency while balancing capacity constraints.

Page classification

R-NUCA classifies memory accesses at the time of a translation look-aside buffer (TLB) miss. It performs classification at the operating system page granularity and communicates these classifications to the cores through the TLB. R-NUCA immediately classifies requests from L1 instruction caches as instructions and performs a lookup on the size-4 fixed-center cluster centered at the requesting core. It classifies all other requests as data requests. R-NUCA satisfies requests for private blocks from the local tile's cache slice, and sends requests for shared blocks to the address-interleaved tile.

The operating system is responsible for distinguishing between private and shared data accesses. R-NUCA extends the page table entries with a "private or shared" bit denoting the current classification, and a field to record the last requestor's core ID. Initially, R-NUCA classifies data pages as private and places them at the cache slice local to the requestor. If another process (running on another core) accesses the same page, it reclassifies the page as shared. Because the classification is performed by the operating system, which is fully aware of process state, a process that migrates from one core to another can retain its pages' private classification after migration.

Indexing and rotational interleaving

R-NUCA indexes blocks within each cluster using standard address interleaving or rotational interleaving. In standard address interleaving, R-NUCA selects an L2 slice based on the bits immediately above the accessed address's set-index bits. In rotational

Generalized rotational interleaving

Rotational IDs (RIDs) in a size-n cluster range from 0 to $n - 1$. A random tile is assigned RID 0, and 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.)

Conceptually, the Boolean function used in rotational interleaving is a compound function of three Boolean functions. Let *Addr* be the address of the block to be placed in a size-n cluster of a tiled multicore processor, where the cache index and the block offset use the lowest k bits of *Addr*. The Boolean functions are as follows.

The interleaving function, *I*, maps the address *Addr* of the block to a destination slice RID_{dest} within the cluster

$$I : Addr \rightarrow RID_{\text{dest}}. \text{ Typically, } RID_{\text{dest}} = Addr[k + \log_2(n) - 1 : k].$$

The relative-vector function, *V*, maps the tuple $<RID_{\text{center}}, RID_{\text{dest}}>$ to a destination vector **d**, denoting the location of RID_{dest} relative to the center of the cluster RID_{center}:

$$V : <RID_{\text{center}}, RID_{\text{dest}}> \rightarrow \mathbf{d}.$$

In plain English, **d** tells the requesting core ''look for the data at the slice above'' or ''at the slice to the left,'' and so on, and it is always relative to the cluster's center. In general,

$$\mathbf{d} = \left(RID_{\text{dest}} + \overline{RID_{\text{center}}} + 1\right) \wedge (n - 1).$$

The *V* function guarantees that overlapping clusters have mutually consistent interleavings.

The global-mapping function, *G*, transforms the relative destination **d** to an absolute one. It maps the tuple $<CID_{\text{center}}, \mathbf{d}>$ to the system-wide ID of the destination slice CID_{dest}. The slice CID_{dest} is where the block with address *Addr* is placed:

$$G : <CID_{\text{center}}, \mathbf{d}> \rightarrow CID_{\text{dest}}.$$

The *G* function is topology-dependent, relying on the tiles' physical locations. This function provides an additional degree of freedom in the cluster's mapping to physical tiles. For example, although a size-4 fixed-center cluster logically consists of four neighboring tiles, these tiles do not need to map to four physically neighboring tiles. This mapping freedom lets cores share blocks among any tiles in a multicore processor, provided there is a mapping of a logical cluster to a physical one.

Conventional address interleaving is a special case of rotational interleaving:

$$\forall \textit{ address } Addr,$$

$$\forall \textit{ slice } X : \begin{cases} CID_x = RID_x \\ RID_{\text{dest}} = I(Addr) = Addr[k + \log_2(n) - 1 : k] \\ V(\langle RID_x, RID_{\text{dest}} \rangle) = \mathbf{RID_{dest}} \\ G(\langle CID_x, \mathbf{RID_{dest}} \rangle) = CID_{\text{dest}} \end{cases}$$

Rotational interleaving can be generalized to any cluster of size power-of-2. The RID assignment algorithm provides correct assignments when the cluster's size is less than $n/2$. For bigger clusters (that is, size-$n/2$ and size-n), the RIDs of tiles in a column differ by m, where m is the number of tiles in a row of the tiled processor. Figure 3b in the main article shows the RID assignments and two examples of size-8 fixed-center clusters in a 64-tile processor.

interleaving, the operating system assigns a rotational ID (RID) to each core, with RIDs in a size-n cluster range from 0 to $n - 1$. The RID differs from the conventional *core ID* (CID) that the operating system assigns to each core for process bookkeeping. Figure 3a shows an example of RID assignment for size-4 fixed-center clusters, where the numbers in the rectangles denote each tile's RID.

To index a block in its size-4 fixed-center cluster, the center core uses the two address bits immediately above the set-index bits. The core evaluates a Boolean function on the address bits and the core's RID, and the outcome determines which slice caches the block. The general form of the indexing function for size-n clusters with the rotational-interleaving address bits starting at offset k is:

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

where the bar denotes the power-of-2 complement, and ^ denotes the bit-wise AND (see the sidebar ''Generalized rotational interleaving''). For size-4 clusters, the result indicates that the block is in, to the right, above, or to the left of the requesting tile, for results $R = 0$, 1, 2, and 3, respectively.

In the example of Figure 3, when the center core of cluster B accesses a block with address bits *0b01,* the core evaluates the indexing function and accesses the block in

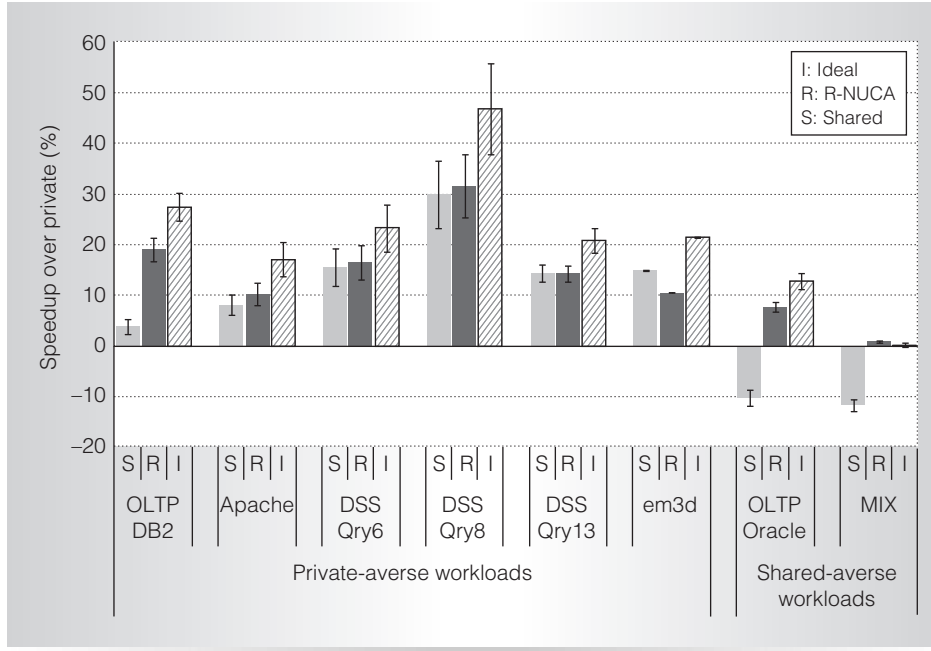

Figure 4. Performance improvement when using reactive NUCA (R-NUCA). Speedup is normalized to the private design.

the slice to its left. Similarly, when cluster C's center core accesses the same address, the indexing function indicates that the block is at the slice above. Thus, each slice stores the same $1/n$ of the data on behalf of any cluster to which it belongs.

Rotational interleaving allows clusters to replicate data in the cache, and at the same time enables fast block lookup and nearest-neighbor communication. Its implementation is simple, 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.

Evaluation

We evaluate R-NUCA by comparing it to a private (P) and a shared (S) organization of the L2 cache. Although previously we analyzed the workloads at the cache block granularity, R-NUCA classifies entire pages. Pages can simultaneously contain blocks of multiple classes (for example, private and shared data); however, a single class dominates the accesses issued to pages. By focusing on entire pages, R-NUCA correctly classifies more than 99 percent of L2 accesses.

R-NUCA improves performance by delivering the private design's fast local access and the shared design's large effective cache capacity and implementation simplicity, bridging the gap between the two. R-NUCA matches the performance of the shared organization for server workloads and provides an average speedup of 17 percent over shared for the multiprogrammed and scientific workloads. Similarly, R-NUCA matches the private organization's performance for the multiprogrammed and scientific workloads, and provides an average speedup of 17 percent over private for server workloads. Overall, R-NUCA attains an average speedup of 14 percent over the private and 6 percent over the shared designs, and a maximum speedup of 32 percent. Figure 4 shows the corresponding speedups across our workload suite, along with the 95 percent confidence intervals produced by our experimental methodology. R-NUCA achieves near-optimal block placement, as its performance is within 5 percent of an ideal design. An ideal cache simultaneously provides the fast access of the private organization and the large capacity of the shared organization.

Moore's Law will likely continue to affect chip density for at least another decade. Similarly, the number of cores on chip and the on-chip interconnect and cache will continue to grow, increasing the latency to access on-chip data. The increase in cache size will make the shared LLC organization even less attractive, as cache blocks will be spread over an ever-increasing number of cache slices. Private-based LLC designs with hardware indirection for coherence and lookup (for example, a directory) face a similar fate, as their area overhead and latency grow with increasing cache and interconnect size.

R-NUCA effectively mitigates the adverse cache effects of large-scale multicore processors. The obviation of hardware coherence mechanisms and the local and nearest-neighbor allocation allow R-NUCA to use mostly local communication. Rotational interleaving eliminates unnecessary network traversals and cache slice accesses, as only the cache slice holding the requested block is probed. These characteristics render R-NUCA relatively immune to the growing on-chip data access latencies. Therefore, we expect R-NUCA will continue to provide an ever-increasing performance benefit over the private and shared designs, while realizing significant power savings.

R-NUCA's low hardware overhead and the simplicity of its design position it among the simplest cache management schemes, while its performance and near-optimal placement characteristics position it among the fastest and most robust ones. Thus, R-NUCA is ideally suited to address the growing concern of efficient cache management for large-scale multicore processors. MICRO

Acknowledgments

We would like to thank B. Gold, S. Somogyi and S. Herbert for their technical assistance, and T. Brecht, T. Strigkos, and the anonymous reviewers for their feedback. This work was partially supported by equipment donations from Intel; a Sloan research fellowship; a European Science Foundation European Young Investigator Award; and US National Science Foundation grants CCF-0702658, CCR-0509356, CCF-0845157, CCR-0205544, IIS-0133686, and IIS-0713409.

References

1. C. Kim, D. Burger, and S.W. Keckler, ''An Adaptive, Non-uniform Cache Structure for Wire-Delay Dominated On-Chip Caches,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002 pp. 211-222.
2. M. Zhang and K. Asanovic, ''Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 336-345.
3. N. Hardavellas et al., ''Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture* (ISCA 09), ACM Press, 2009, pp. 184-195.
4. T.F. Wenisch et al., ''SimFlex: Statistical Sampling of Computer System Simulation,'' *IEEE Micro,* vol. 26, no. 4, July/Aug. 2006, pp. 18-31.

Nikos Hardavellas is the June and Donald Brewer Assistant Professor of Electrical Engineering and Computer Science at Northwestern University. His research targets architectures, memory system design, and runtime environments for parallel systems. Hardavellas has a PhD in computer science from Carnegie Mellon University. He is a member of the ACM and the IEEE.

Michael Ferdman is a PhD candidate in electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture, with an emphasis on proactive memory system design. Ferdman has an MS in electrical and computer engineering from Carnegie Mellon University. He is a student member of the ACM and the IEEE.

Babak Falsafi is a professor of computer and communication sciences at École Polytechnique Fédérale de Lausanne, where he directs the Parallel Systems Architecture Lab.

His research targets architectural support for parallel programming, resilient systems, memory systems, and computer system performance modeling and evaluation. Falsafi has a PhD in computer science from the University of Wisconsin-Madison. He is a senior member of the ACM and the IEEE.

Anastasia Ailamaki is a professor of computer science at the École Polytechnique Fédérale de Lausanne. Her research interests include optimizing database workloads for modern hardware and disks and managing large data sets for scientific applications. Ailamaki has a PhD in computer science from the University of Wisconsin-Madison.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.