Optimizing Replication, Communication, and Capacity Allocation in CMPs

Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{zchishti, mdpowell, vijay}@purdue.edu

Abstract

Chip multiprocessors (CMPs) substantially increase capacity pressure on the on-chip memory hierarchy while requiring fast access. Neither private nor shared caches can provide both large capacity and fast access in CMPs. We observe that compared to symmetric multiprocessors (SMPs), CMPs change the latency-capacity tradeoff in two significant ways. We propose three novel ideas to exploit the changes: (1) Though placing copies close to requestors allows fast access for read-only sharing, the copies also reduce the already-limited on-chip capacity in CMPs. We propose controlled replication to reduce capacity pressure by not making extra copies in some cases, and obtaining the data from an existing on-chip copy. This option is not suitable for SMPs because obtaining data from another processor is expensive and capacity is not limited to on-chip storage. (2) Unlike SMPs, CMPs allow fast on-chip communication between processors for read-write sharing. Instead of incurring slow access to read-write shared data through coherence misses as do SMPs, we propose in-situ communication to provide fast access without making copies or incurring coherence misses. (3) Accessing neighbors' caches is not as expensive in CMPs as it is in SMPs. We propose capacity stealing in which private data that exceeds a core's capacity is placed in a neighboring cache with less capacity demand.

To incorporate our ideas, we use a hybrid of private, per-processor tag arrays and a shared data array. Because the shared data array is slow, we employ non-uniform access and distance associativity from previous proposals to hold frequently-accessed data in regions close to the requestor. We extend the previously-proposed Non-uniform access with Replacement And Placement usIng Distance associativity (NuRAPID) to CMPs, and call our cache CMP-NuRAPID. Our results show that for a 4-core CMP with 8 MB cache, CMP-NuRAPID improves performance by 13% over a shared cache and 8% over private caches for three commercial multithreaded workloads.

1 Introduction

CMOS scaling trends are leading to greater numbers of smaller transistors on a chip but a relative increase in wire delays. Chip multiprocessors (CMPs) are an increasingly common architecture for utilizing the numerous transistors to achieve high performance. CMP substantially increases capacity pressure on the on-chip memory hierarchy, which must now support multiple cores. At the same time, CMP also requires its processors to have fast access to data. The lowest-level on-chip cache not only needs to utilize its limited capacity effectively but also has to mitigate the increased latencies due to wire delays.

Two options for the lowest-level on-chip cache in CMPs are: shared or private caches. A shared cache has a single copy for each cache block and allows the cores to share the cache capacity. However, shared caches are slow because of the wire delays associated with large caches. In contrast, private caches are faster because they are smaller and can be located closer to each core. However private caches provide limited capacity to each core. Thus, shared or private caches can provide either capacity or fast access but not both.

In this paper we explore the possibility of achieving both goals. Symmetric multiprocessors (SMPs) and distributed shared-memory machines (DSMs) also target these goals. We make the key observation that CMPs, however, change the latency-capacity tradeoff in two significant ways. We propose three novel ideas to exploit the changes: (1) Though placing copies close to requestors allows fast access for *read-only sharing*, the copies also reduce the effective on-chip capacity in CMPs. We propose *controlled replication* which avoids extra copies in some cases, and obtains the data from an already-existing on-chip copy at the cost of some extra latency. Because the copy is on-chip, the latency penalty is small and is offset easily by the reduction in off-chip misses due to reduced capacity pressure. In SMPs and DSMs, obtaining data from another processor is expensive and capacity is not limited to on-chip storage due to off-chip caching. Therefore, on-chip capacity is a lesser concern in SMPs and DSMs, and trading off latency for on-chip capacity is inappropriate. (2) Inter-processor communication induced by *read-write sharing* is on-chip in CMPs and off-chip in SMPs and DSMs. Because on-chip communication is faster than off-chip communication, there is a new opportunity to optimize read-write sharing in CMPs. Rather than incur slow access to read-write shared data through coherence misses as do SMPs and DSMs, we propose *in-situ communication* which provides fast access to the data without making copies (via controlled replication) or incurring coherence misses. (3) SMPs and DSMs migrate private data (in the case of *no sharing*) close to the requesting cores to allow fast access. While such migration is useful for CMPs as well, it may result in inefficient use of the on-chip capacity. For example, if one core exceeds the capacity of its private cache, migrating new blocks closer to the core will cause evictions even if there is unused on-chip capacity in a neighbor's private cache. We propose *capacity stealing* which enables a core to migrate its less-frequently-accessed data to unused frames in neighboring caches with less capacity demand. Thus, capacity stealing dynamically customizes allocation of on-chip capacity. Because neighboring cores are on-chip in a CMP, accessing neighbors' caches is not expensive as is the case in SMPs and DSMs.

Neither pure private nor pure shared cache can accommodate our above ideas. Controlling replication in a pure private cache

would result in slow accesses through the bus, if the block is often reused. A pure shared cache as used in several previous designs (e.g., [19,26,12,4, 25]) has latency problems. A recent paper [6] proposes to alleviate shared cache's latency by employing non-uniform access. Non-uniform cache architecture (NUCA) [14] reduces latency in large uniprocessor caches by allowing fast access to the regions of the cache close to the processor and slow access to farther regions. To reduce latency, NUCA aims to place frequently-accessed cache blocks in the regions closest to the processor. Applying NUCA to CMPs, [6] allows migration of blocks close to the requestor. Otherwise, [6]'s design is still a pure shared cache which does not allow replication or exploit our ideas. [6] concludes that NUCA's migration is ineffective in the presence of sharing because each sharer pulls the block toward it, leaving the block in the middle, far away from all the sharers.

Because neither pure shared nor pure private cache accommodate our ideas, we propose a hybrid of private, per-processor tag arrays and a shared data array. Because the shared data array is slow, we employ non-uniform access to hold frequently-accessed data in regions close to the requestor. We extend the Non-uniform access with Replacement And Placement usIng Distance associativity (NuRAPID) [8], which improves upon NUCA, from uniprocessors to CMPs. We call our cache CMP-NuRAPID. To provide fast access to the tag, CMP-NuRAPID provides each core with its own private tag array, which snoops on a bus for coherence like SMPs. Though CMP-NuRAPID uses non-uniform access like [6] there is one key difference: CMP-NuRAPID employs replication in the shared data array to allow fast access for shared data, and customizes its replication via controlled replication and in-situ communication to exploit CMP's latency and capacity characteristics. In contrast, [6] inflexibly opts for disallowing replication altogether and relying only on migration.

Exploiting its hybrid structure, CMP-NuRAPID employs controlled replication for read-only sharing and in-situ communication for read-write sharing. For controlled replication, CMP-NuRAPID forces multiple tag arrays to point to the same copy in the data array. In contrast, controlled replication in a pure private cache would imply that the processor that does not have a copy has to incur cache miss overhead to locate the block. CMP-NuRAPID avoids the overhead by allowing the sharers to keep tag copies without making data copies.

For in-situ communication in read-write sharing, CMP-NuRAPID uses controlled replication to force only one data copy. The writer and the readers have tag copies which point to the single data copy. To prevent the writer from invalidating the readers' tag copies (like a pure private cache), CMP-NuRAPID employs a new state, called the communication state, in its invalidation-based protocol. In this state, the writer can write to the data copy and the readers can read the copy without incurring coherence misses. Though an update-based protocol could provide fast read-write sharing, it incurs not only the overhead of the updates going through the bus but also the capacity pressure of extra copies.

For capacity stealing of private data, CMP-NuRAPID exploits non-uniform access and modifies NuRAPID's promotion and demotion policies to migrate frequently-accessed blocks close to the core. These policies are especially beneficial for multiprogrammed workloads which have non-uniform capacity demands. The cores with more capacity demand can demote their less-frequently-used data to unused frames in data arrays closer to the cores with less capacity demands. Whereas pure private caches blindly migrate private data, CMP-NuRAPID enables better utilization of the shared capacity.

To summarize, the contributions of CMP-NuRAPID are:

- its hybrid private tag and shared data organization;
- its controlled replication, in-situ communication, and capacity stealing;
- the results that for a 4-core CMP with 8 MB on-chip cache, CMP-NuRAPID improves performance by 13% over a shared cache and 8% over private caches for three commercial multi-threaded workloads.

The rest of this paper is organized as follows. Section 2 describes CMP-NuRAPID's organization. Section 3 explains controlled replication, in-situ communication, and capacity stealing. Section 4 describes methodology and Section 5 presents results. Section 6 discusses related work. We conclude in Section 7.

2 Organization

Because CMP-NuRAPID relies on non-uniform cache access, we provide a brief background on previous non-uniform caches. Then we describe the changes for CMP-NuRAPID.

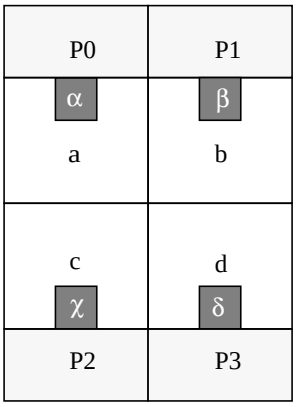
2.1 NUCA and NuRAPID

The key concept of Non-uniform cache architecture (NUCA) [14] is to place frequently-accessed information in the region closest to the core to allow fast access. NUCA distributes the tag and data arrays throughout the cache and couples tag placement with data placement. Because NUCA *explicitly couples* a cache block's set-associative way number to its distance from the processor, NUCA can place only one or two ways in each set close to the processor. However, if a "hot" set has more frequently-accessed ways, the accesses are not all fast even though the fastest region is large enough to hold all the ways of the set.

Non-uniform access with Replacement And Placement usIng Distance associativity (NuRAPID) [8] improves upon NUCA by decoupling the set-associative way number from data placement, as proposed by [13]. This decoupling allows any number of cache blocks within a set to be placed close to the processor. Like [13], NuRAPID achieves this decoupling by leveraging sequential tag-data access, a common technique to reduce energy and wiring complexity in large caches [10,28]. In sequential tag-data access, the tag array is probed *prior* to the data array, pinpointing the location of the matching set-associative way and avoiding parallel access of all the set-associative ways. Therefore, the exact location in the data array can be determined even if there is no implicit coupling between tag and data locations. NuRAPID exploits this level of indirection provided by sequential tag-data access to implement *distance associativity* where pointers kept in the tag and data array allow blocks to be placed anywhere in the data array. NuRAPID employs policies which promote frequently-accessed data closer to the processor without being restricted like NUCA only to one or two set-associative ways per set.

NuRAPID divides the cache data array into several large (hundreds of KB to a few MB) distance groups, or d-groups. Each d-group has a single uniform access latency. Using distance associativity, NuRAPID places data blocks in the appropriate d-groups.

NuRAPID uses pointers in tag and data array entries to implement distance associativity. The forward pointer is located in the tag array and points to the specific frame in the data array where the block is located. The reverse pointer is located in the data array



d-group preference	Core P0	P1	P2	P3
1	a	b	c	d
2	b	d	a	c
3	c	a	d	b
4	d	c	b	a

cores: P0-P3
tag arrays: α–δ
data d-groups: a-d

FIGURE 1: 4-core CMP with CMP-NuRAPID

and points back to the tag entry. The reverse pointer is used for replacement, which we will discuss in detail in Section 3.3.

The forward pointer does not significantly increase latency because conventional large caches already use sequential tag-data access. The reverse pointer's latency overhead is minimal because the reverse pointer is much smaller than the block. The pointers do add some overhead to the cache capacity. For example, in a 8-MB cache with 128-B blocks, 16-bit forward and reverse pointers constitute a 256-KB, or 3%, overhead [8]. [8] proposes ways to reduce the overhead at the cost of some flexibility. This overhead is offset by the fact that CMP-NuRAPID leverages the pointers to enable better capacity utilization and faster communication.

2.2 CMP-NuRAPID

Like NuRAPID, CMP-NuRAPID (1) uses sequential tag-data access; (2) divides the data array into several distance groups (d-groups) and employs distance associativity; and (3) uses forward and reverse pointers. Next, we describe how CMP-NuRAPID differs from NuRAPID.

2.2.1 Data Array

Unlike NuRAPID which specifies distances in terms of only one core, CMP-NuRAPID must specify distances for all the cores. Each data d-group has a different access latency for each core, as shown in Figure 1 for a four-core CMP. The number of d-groups need not equal the number of cores, but bandwidth considerations make it preferable to have at least one d-group per core.

To exploit non-uniform access, each core must rank the d-groups in terms of preference to place frequently-accessed blocks. Obviously, the d-groups closest and farthest to a core have the highest and lowest preference for that core. But the other rankings are not obvious as multiple d-groups may have the same distance from a core, such as d-groups *b* and *c* from *P0* in Figure 1. The rankings must avoid unnecessary contention among the cores. For example, if *P0* and *P1* use each other's first preference as their second preference (d-groups *b* and *a*), the cores will compete in these d-groups even if other d-groups (e.g., *c* and *d*) at the same distance have space. Therefore, we stagger the rankings for each d-group across the cores, as shown in Figure 1. This ranking is important for capacity stealing discussed later in Section 3.3.

Like private caches, CMP-NuRAPID employs replication of shared data in the data array to allow fast access. Each core can keep its own copy in its closest d-group. Thus, there may be multiple copies of the same block in different d-groups. We will discuss the policies to control the replication in Section 3.1.

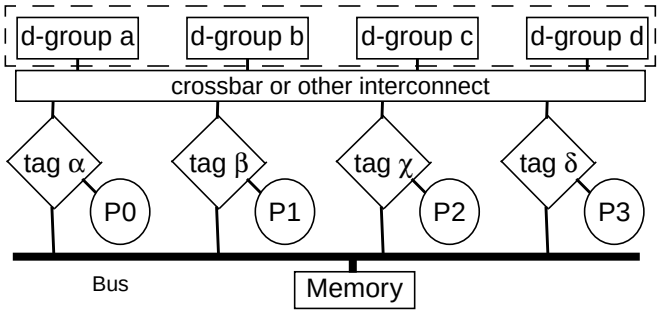


FIGURE 2: CMP-NuRAPID Tag and Data Arrays

2.2.2 Tag Arrays and Pointers

While distance-associativity in the data array brings the frequently-accessed data closer to the core, the tag also has to be close to reduce latency. Consequently, CMP-NuRAPID replaces NuRAPID's single shared tag array with private per-core tag arrays, placed close to each core. Figure 2 shows CMP-NuRAPID's organization. Like private caches, CMP-NuRAPID's tag arrays snoop on a bus to maintain coherence (discussed in Section 3). The bus has separate wires for addresses and pointers. The tag arrays access the d-groups through a crossbar (which is also used in conventional banked caches and is acceptable due to the small number of d-groups) or other interconnect.

In contrast, [6] uses NUCA's tag arrays which are distributed throughout the cache. Whereas [6] does not use any replication, CMP-NuRAPID does; and using distributed tag arrays interconnected by a switch network would require a directory-like scheme to maintain coherence. Unfortunately, directory schemes are harder to build than snoopy bus schemes. In addition to [6], several commercial CMPs (e.g., [26,12,4, 25]) employing private L1s and shared L2 use a directory scheme to keep the L1s coherent. These CMPs accept the complexity of the directory because using a snoopy bus for the L1s would overwhelm the bus due to the high cache miss rates of the small L1s. The CMPs do not have the option of using private L2s and snooping on the L2s instead of the L1s to avoid the high miss rates because of lack of transistors. In contrast, CMP-NuRAPID uses a hybrid organization which both fits within the transistor budget and allows the bus to snoop on the large L2 whose lower miss rate can be supported by the bus. Thus, CMP-NuRAPID enables the use of the simpler snoopy scheme.

CMP-NuRAPID leverages the forward and reverse pointers for controlled replication. Multiple tag arrays can share a single copy of a data block by pointing to the same block in the data array. Consequently, CMP-NuRAPID's tag arrays contain more entries than (e.g., twice as many as) the data arrays. We will describe our policy for controlled replication in Section 3.1.

In the extreme case, every data block could be shared by all the cores requiring that each private tag array be as large as the complete tag array in a shared cache. However, the disadvantages of this solution are the increased latency of the larger tag arrays and an unacceptably large capacity overhead. For an 8-MB cache with 128B blocks in a 4-core CMP, this solution would amount to quadrupling the tag capacity for each of the core resulting in a 23% increase in total cache size.

We propose a compromise by doubling each core's tag capacity, resulting in a 6% increase in total cache size. (We double the number of sets while maintaining the same set associativity.) We found that doubling the tag capacity performs almost as well as

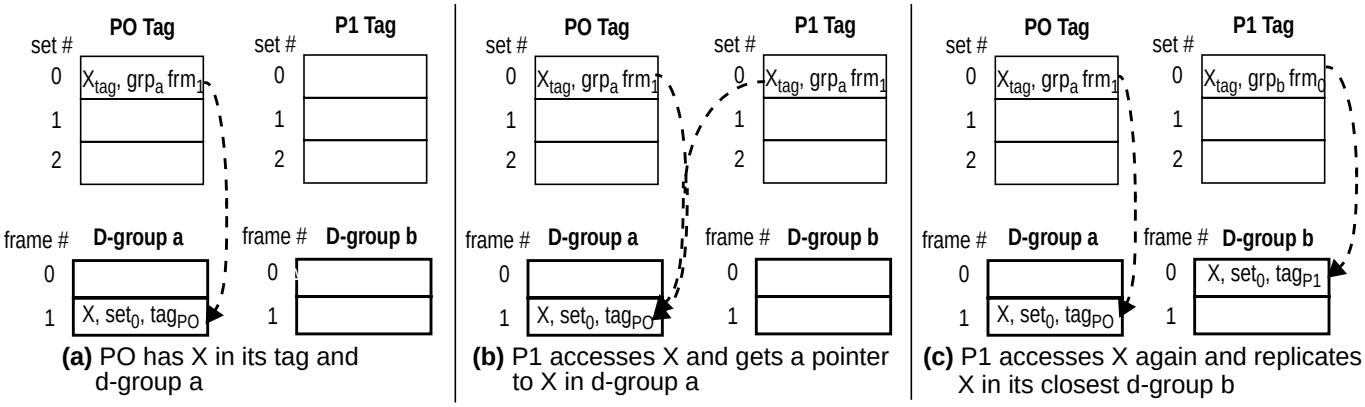


FIGURE 3: Controlled Replication Example for Block X

quadrupling. Note that the shared caches used in several previous designs [26,4, 19] also incur a capacity overhead due to storing L1 tag copies at the L2 to keep L1 caches coherent. [4] showed that the L1 tag copies in Piranha incur a 4% overhead in total on-chip cache capacity. CMP-NuRAPID's tag overhead is offset by the fact that it allows the data array capacity to be utilized more efficiently, reducing both capacity and coherence misses.

3 CMP-NuRAPID Optimizations

In this section, we discuss the CMP-NuRAPID optimizations for shared and private data. CMP-NuRAPID extends the invalidation-based 4-state MESI cache coherence protocol [21]. We first describe the extensions needed in the cache coherence protocol to implement controlled replication for read-only sharing, and in-situ communication for read-write sharing. Then we explain the CMP-NuRAPID policies to implement capacity stealing for private data.

3.1 Controlled Replication

The main goal of replication is to achieve fast access to shared data by keeping separate copies of the shared block close to each processor. Private caches attempt to achieve this goal by keeping as many copies of a shared block as readers sharing the block. Though always copying the block provides fast access on reuse, such uncontrolled replication wastes precious on-chip capacity. Private caches could save capacity by disallowing a reader from copying a block to its private cache, if an on-chip copy already exists in another processor's private cache. However when the block is re-read, the reader will incur cache miss overhead to locate the block. The savings in capacity by controlling replication in private caches may not offset the latency increases caused by the overhead on reuse.

CMP-NuRAPID exploits the hybrid structure to perform controlled replication (CR) without incurring cache misses on reuse. Because CMP-NuRAPID uses private tag arrays and shared data array, tag entries in multiple private tag arrays can point to the same block in the data array. When a reader misses on a block which is already present in the shared data array, the reader obtains the data from the already-existing on-chip copy. The reader makes a tag copy but not a data copy. We observe that many blocks brought to the cache are not reused in commercial workloads. Not copying the block in data array on the first use saves capacity for blocks which are never reused. However we observe that most of the blocks that are reused have two or more reuses. Therefore, on second use, a data copy is made in the reader's closest d-group to avoid slow accesses for future reuses. Because the block is already present in the tag array, the second use does not incur a coherence miss. Also, because the already existing copy is on-chip, the latency penalty for second use is small and is offset easily by the reduction in off-chip misses due to reduced capacity pressure.

We illustrate CR in CMP-NuRAPID by an example in Figure 3. *P0* has a copy of the data block *X* in its closest d-group *a* (Figure 3a). The tag entry for block *X* in *P0*'s tag array points to the data block. *P1* tries to access *X* and misses in its tag array. *P1* sends a read request on the bus. Because *P0* has a data copy, *P0* responds by sending the forward pointer in its tag entry on the pointer wires. This pointer return is unlike conventional cache-to-cache transfers where the actual data and *not* a pointer is returned. *P1* does not create a copy of *X* in its closest d-group *b.* Instead, the tag entry for *X* in *P1* now points to the already-existing copy of *X* in d-group *a* (Figure 3b). *P1*'s tag then accesses the d-group *a* through the crossbar (this access is direct and does not go through *P0*). The reverse pointer of *X* continues to point to *P0*'s tag array. Reverse pointers are used for replacements and because our replacement policy allows only *P0* to replace *X*, the reverse pointer need not change. Note that we do not need any new state in the coherence protocol to identify the tag entries which are pointing to a data copy in some other processor's closest d-group. The shared state suffices because the forward pointer already identifies the d-group that contains the data copy. If there is another access to *X* by *P1*, this access hits in the tag array. *P1* checks the forward pointer and finds that the block is in the farther d-group *a* (by examining the forward pointer, the tag array can determine whether the d-group is close or far). Then, *P1* makes a copy of *X* in its closest d-group *b* and updates the forward pointer in its tag entry for *X* to point to the newly-created copy (Figure 3c).

The replacement of shared blocks in CMP-NuRAPID may cause a correctness problem. When a processor decides to replace a data block which is present in the shared state in its tag array, then due to CR, there is a chance that tag array entries in other processor(s) may be pointing to the data block being replaced. If these sharers are not informed about the replacement of data block, their tag entries will contain dangling pointers to incorrect data after replacement. To solve this problem, the processor replacing the data block sends a special *BusRepl* transaction on the bus before replacing the block. The sharers that observe the *BusRepl* transaction will invalidate any tag array entry pointing to the data block being replaced. Note that if a sharer has its own copy of the data block in its closest d-group, the sharer does not need to

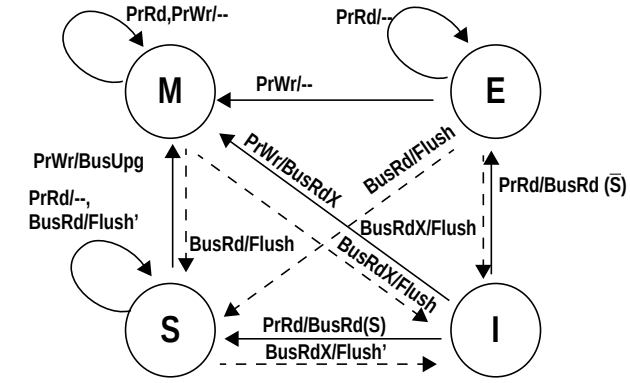
invalidate its tag array entry. Thus, unlike private caches, CMP-NuRAPID sends an invalidation on the bus every time a shared block is replaced. It is possible to avoid sending invalidations for those data blocks which are not pointed to by multiple tag arrays. However, the information about multiple tag copies of read-shared blocks is not available unless a new state is added to the coherence protocol. Because of CMP-NuRAPID's shared data array capacity, replacements are not that frequent. Consequently, such invalidations are infrequent. Instead of adding a new state, we try to minimize the number of the invalidations even further by decreasing the possibility of a shared block being replaced. We will discuss the details of our replacement policy in Section 3.3

Due to rare timing issues, simply invalidating the tag copies of the data block being replaced does not solve the problem entirely. Because readers do not advertise that they are in the process of reading (which would defeat the purpose of having the tag entry), a replacement invalidation may occur in the middle of a read. That is, there is a chance that one of the cores having a tag copy may start to read the block before the invalidation and end reading after the invalidation. Such a timing will result in a read hit in the tag array followed by an access to a farther d-group. If the invalidation is done and even the replacement finishes *before* the read completes, then the reader may read incorrect data of an unrelated block that replaced the original block. We solve this problem by requiring the tag for the block being read from a farther d-group be marked busy, implying that a read is under progress. (As mentioned before, our replacement policy allows only the core closest to a d-group to replace blocks from that d-group. Therefore, for reads to the closest d-group there are no problems with replacement because the reader and the replacer are the same). If any replacement invalidations appear on the bus, they will be inhibited until the read has completed. The busy bits are held in the cache controller as part of the controller state and not in the tags as coherence state. Such state in cache controllers is common. For instance, transient states used in cache coherence protocols are implemented in the controller.

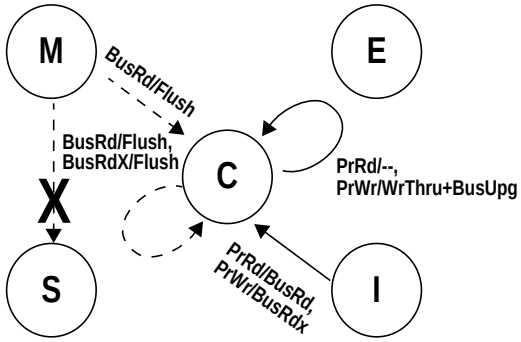
Marking the block busy solves the problem *only if* the read starts before the replacement invalidation. If the invalidation occurs before the read, there is another problem in that the invalidation is not applied instantaneously. For instance, in multi-level cache hierarchies, it may take several cycles for the invalidation to reach all the way up to the L1 cache. Even though the read starts after the invalidation appeared on the bus, the read may still go ahead and get incorrect data. Delayed application of invalidations also causes consistency-related problems for SMPs. SMPs use queues in the cache hierarchy to hold the order of the bus transactions until the transactions are applied to the caches. We solve our problem by extending the queues and putting an entry in our cache's queue before sending a read request to a farther d-group. When the read returns, the read data is returned to L1 cache in the order of the queue. Because the outstanding invalidation precedes the entry for the read, the tag for the data block being read is invalidated before the read data is returned. Accordingly, we require that the tag array be probed once more before returning the read data, so that the data is discarded if the tag has been invalidated.

3.2 In-situ Communication

Read-write sharing in multithreaded workloads involves communication of data between the writers and the readers. For a read-write-shared block, private caches using invalidation-based protocol invalidate the copy of the block in the reader's cache on every write. On a subsequent read, the reader incurs the penalty of a coherence miss to obtain the data from the writer and makes a new copy in its private cache. Thus, private caches incur slow access to read-write shared data through coherence misses, and waste capacity due to multiple copies. This approach is not suitable for CMPs where on-chip communication is fast but on-chip capacity is limited.



(a) Base Cache Coherence Protocol



(b) Modifications

FIGURE 4: Cache Coherence Protocol

CMP-NuRAPID uses in-situ communication (ISC) which provides fast access to read-write shared data without making copies or incurring coherence misses. To perform ISC, CMP-NuRAPID utilizes the hybrid structure of the cache and employs CR to force only one data copy for a read-write shared block. The writer and the readers have their private tag copies which point to the single data copy. Because we observe that each write is usually read more than once by each reader in commercial workloads, CMP-NuRAPID places the data copy close to (one of) the reader(s). To prevent the writer from invalidating the readers' tag copies (like a pure private cache), CMP-NuRAPID employs a new state, called the communication state, in its coherence protocol. In this state, the writer can write to the data copy and the readers can read the copy without incurring coherence misses. The CR used in ISC has the same capacity advantages that we discussed for read-only sharing in Section 3.1

It may seem that private caches can avoid coherence misses in read-write sharing by using an update protocol, making the communication state unnecessary. However, unlike ISC in CMP-NuRAPID, an update protocol requires the updates to go through the bus for copying the data to the reader's caches, incurring an overhead on every write. Furthermore, update protocols keep multiple copies of the read-write shared block giving rise to capacity problems similar to the ones caused by uncontrolled replication in

read-only sharing.

Next we discuss the changes in coherence protocol for ISC. Figure 4a shows the state transition diagram for a 4-state MESI protocol. (We do not include the transitions for replacements in Figure 4.) The solid arcs represent the state transitions in the tag arrays of the initiating processor, while the dotted arcs show the transitions in the tag arrays which respond to observed bus transactions. Figure 4b shows the changes in MESI for ISC. CMP-NuRAPID adds one more state to the MESI protocol, the communication state (represented by *C* in Figure 4b) to obtain the 5-state *MESIC* protocol. The *C* state allows multiple processors to share a dirty block. The M state in MESI does not suffice for ISC, because *M* indicates a dirty block with only one tag copy, while *C* represents a dirty block with multiple tag copies. Any transitions which appear in Figure 4a for MESI but are not shown in Figure 4b also take place in MESIC. The arc labelled x in Figure 4b shows the transition in MESI which does not exist in MESIC. All the other arcs represent the new transitions added to MESI for ISC. We discuss the added and deleted transitions next.

When a read miss occurs and a dirty copy (either *M* or *C*) already exists, the reader makes a new copy of the block in its closest d-group, and the previous data copy is invalidated. All the sharers enter (or remain in) *C* and their tag entries point to the new data copy. The transition from *M* to *S* does not exist in MESIC protocol, because an *M* block transits to *C*, instead of going to *S*, upon seeing a read request on the bus.

When a writer does not find the block in its tag array and the block is present in *C* in other tag arrays, the writer does not make a copy of the data block. Instead, the writer enters *C* pointing its tag entry to the already-existing data copy, and writes to the copy. Thus, the copy stays close to the reader.

The transition from *I* to *C* on a read/write miss requires that the reader/writer knows whether a dirty copy of the block exists or not. We add a *dirty* signal to detect the presence of another dirty copy (similar to the *shared* signal used in MESI protocol to detect a clean copy). The tag arrays carrying a dirty copy assert the dirty signal to inform the reader/writer about the existence of a dirty copy so that the reader/writer can decide whether to transit to *C*.

A read/write to a block in *C* does not generate any state transition. However the writer to a *C* block sends a *BusRdX* request on the bus. Whenever a sharer in *C* state observes a *BusRdX* transaction, it remains in the *C* state but invalidates the L1 copy of the block, if one exists. This invalidation is necessary because otherwise a sharer may read a stale value from the L1 cache. In a private cache, repeated writes to the same block do not result in repeated invalidations as long as the block remains in *M*. In CMP-NuRAPID, however, the block remains in *C* and repeated invalidations are needed. Nonetheless the reduction in coherence misses due to *C* outweighs the increase in the number of invalidations.

Because ISC allows multiple cores to share the same dirty block, pure write-back L1 caches cause coherence problems. If a writer writes to an L1 cache block in *C* state without writing to the L2 block, a reader reading the shared L2 copy may read the incorrect value. Therefore we use write-through for all the *C* blocks in the L1 cache. Write through is not needed for *E* or *M* blocks because no other tag copy exists. Many existing CMPs use write-through L1 caches [26,4, 19] to avoid large coherence traffic from their small L1 caches. Therefore, write through for *C* blocks is not likely to cause bandwidth problems.

The replacements for ISC work in a similar manner as the replacements for CR in read-only sharing. When a processor decides to replace a data block which is present in the *C* state in its tag array, the processor sends the *BusRepl* transaction on the bus. The sharers that observe the *BusRepl* transaction invalidate their tag copies. The timing issues that we discussed regarding replacements in Section 3.1 also exist here. We use the solutions mentioned in Section 3.1 to solve these problems.

There are no transitions out of *C* other than those due to replacements. Consequently, a read-write shared block may get stuck in the d-group closest to a processor that never reuses the block. In that case the other sharers will experience slow hits to the block. However, we note that most of the read-write shared blocks in commercial workloads are frequently read after being written, thus decreasing the possibility of a *C* block getting stuck close to a processor that never reuses it. Therefore, we adopt the simple solution of having no exits out of *C*. We leave addressing other workloads, where this issue may be a problem, to future work.

3.3 Capacity Stealing

The main goal of capacity stealing (CS) is to bring frequently-accessed data blocks close to the core. Private caches blindly migrate data by bringing a new block to the cache at the cost of evicting another block. This approach does not utilize the on-chip cache capacity efficiently. For example, if one core needs more capacity than provided by its private cache, it will incur capacity misses even if there is unused capacity in another core's private cache.

CMP-NuRAPID exploits non-uniform access and modifies NuRAPID's promotion and demotion policies to bring frequently-accessed blocks close to the core. Unlike private caches, the shared data array in CMP-NuRAPID enables better utilization of on-chip cache capacity. The cores with more capacity demand can *demote* their less-frequently-used data to unused frames in the d-groups closer to the cores with less capacity demands. Thus, capacity stealing dynamically customizes allocation of on-chip capacity. This strategy enables CMP-NuRAPID to incur fewer off-chip capacity misses than private caches.

CS is less important for multithreaded workloads because cores usually have uniform capacity demands. Due to similar working set sizes for different threads, it is unlikely that a core having no unused data blocks in its closest d-group may find unused blocks in other cores' closest d-groups. However, CS is especially beneficial for multiprogrammed workloads which usually have non-uniform capacity demands. Multiprogrammed workloads are important for multiprocessors [23].

Because distance associativity allows data to be placed in and migrated (promoted or demoted) to any d-group, we need placement, promotion, demotion, and replacement policies. Next, we discuss how blocks are placed initially in the cache and how they are promoted to closer d-groups. Then we discuss replacement and demotion to farther d-groups.

3.3.1 Placement and Promotion Policies

The placement and promotion policies are different for private (identified by the E state) and shared blocks. We discuss private blocks first.

CMP-NuRAPID initially places all private blocks in the data d-group closest to the initiating core. Space for cache misses is cleared through replacement policies discussed in the next subsec-

tion. Because each tag array is highly set-associative and placement within the d-group is not restricted by set mapping, this policy flexibly allows a large number of frequently-accessed blocks to be placed in the closest d-group. If a tag hit occurs for a private data block that is not present in the closest d-group, then we "promote" that block to a closer d-group. After promotion, we update the forward-pointer in the tag array to point to the new frame.

We examine two policies for promotion. The first, *next-fastest*, promotes the block to the next closest d-group from its current location (Section 2.2.1). The second, *fastest*, promotes the block directly to the closest d-group for the core. [8] found *next fastest* to be most effective for uniprocessors. However, the environment in a CMP is different because one core's *next-fastest* d-group is another core's *fastest* d-group, and it may be undesirable to pollute another core's fastest d-group during the promotion process. We found *fastest* to be more effective in CMPs than *next fastest.*

Shared blocks are placed as per CR or ISC, depending upon whether the block is read-only shared or read-write shared. Shared blocks do not need promotion because they are never demoted. (We discuss the reasons for disallowing demotions of shared blocks in next subsection.) Thus, shared blocks do not move around in the cache, and we avoid the problems of sharers getting incorrect data due to data movement.

3.3.2 Demotion and Replacement Policies

Distance-associative caches such as NuRAPID and CMP-NuRAPID must address two forms of replacement: data replacement and distance replacement. Data replacement is similar to conventional caches, occurs upon cache misses, and evicts a block from the cache. Distance replacement is unique to distance associative caches, occurs upon demotion, and evicts a block from a d-group but demotes it to another d-group instead of evicting it from the cache. We discuss data replacement first.

In data replacement, CMP-NuRAPID replaces a block from the same set as the cache miss. We prefer to replace in the order of invalid, private, and shared because eviction of shared blocks requires invalidations that we discussed in Section 3.1. We use LRU within each category. Replacing an invalid block or a valid (private and shared) block that points to a farther d-group creates space only for tag but not for data within the closest d-group. If it is a private block then the data block is evicted, but that creates space in a farther d-group. Some block in the closest d-group is then distance-replaced to that specific farther d-group. If it is a shared block then the data block is not evicted and it is left for the other sharers. Therefore, space needs to be created by distance-replacing some block in the closest d-group to a non-specific farther d-group. The same needs to be done for an invalid block. Replacing a valid (private and shared) block that points to the closest d-group creates space for both tag and data. Upon evicting such a shared block, the other tag copies are invalidated.

In distance replacement to a non-specific d-group, if we keep demoting from one d-group to its next neighbor then this process will go into a cycle because eventually the demotions will loop back to the first d-group. We break this cycle by choosing a d-group at random to stop the demotions. In distance replacement to a specific d-group, we stop the demotions at that d-group. From the originating d-group to the d-group where demotions stop, distance replacement simply performs repeated demotions going from one d-group to the next-fastest d-group. In each d-group, a block is chosen to be demoted. This choice is at random as well because LRU requires $O(n^2)$ hardware to track n frames and there are too many frames in a large d-group (e.g., a 1MB d-group with 128B entries has 8192 frames). The reverse pointers of the chosen blocks locate their tag entries whose forward pointers are updated to point to the new, demoted location of the block.

If the chosen block is private then the demotions proceed as mentioned, but if it is a shared block, we simply evict it instead of demoting it, as we alluded to in Section 3.3.1. We do not demote shared blocks due to the following possible scenario: suppose that *P0* in Figure 1 demotes a block *X* from its closest d-group *a* to its next closest d-group *b.* When *P0* accesses *X* next time, the tag entry indicates that *X* is a shared block present in a farther d-group. Due to the CR policy explained in Section 3.1, *P0* makes a copy of *X* in its closest d-group *a* and the tag entry for *X* now points to the new copy. The old copy in d-group *b* now contains a dangling reverse pointer. To avoid this problem, we evict shared blocks upon replacement.

Finally, we note that the demotions are not frequent enough to cause a bandwidth problem in the tag arrays or data d-groups. To validate this claim, all our experiments assume that each private tag array and data d-group is single-ported and not pipelined (the crossbar in Figure 2 is for parallel accesses to *different* d-groups). Thus, our aggregate bandwidth is the same as a single-ported private cache and an n-banked shared cache in an n-CPU CMP.

Table 1: 8 MB Cache and Bus Latencies

Cache and Component	Latency (cycles)
Shared 8 MB 32-way, 4 ports (latency of 8-way, 1-port)	
Tag (includes wire delay of central tag)	26
Data	33
Total	59
Private 2 MB 8-way, 1 port	
Tag	4
Data	6
Total	10
CMP-NuRAPID with four 2 MB d-groups	
Tag w/ extra tag space	5
Data d-groups (a,b,c,d)	6,20,20,33
Pipelined split-transaction bus (all designs with bus): 32	

4 Methodology

4.1 Simulation Environment

We use Simics [16] to perform a full-system simulation of a 4-core CMP with x86 cores. Each CPU uses in-order issue, has 64 KB, 2-way L1 I and D caches with 64-byte blocks, 3-cycle latency, and allows 1 outstanding miss. We assume 4 GB memory with a 300-cycle latency.

Our simulated system runs the Debian GNU/Linux O/S "testing" version 3.1 with SMP-enabled Linux kernel version 2.4.27 custom-compiled to interface with Simics. We compile the kernel and other applications using gcc 3.3.4.

4.2 On-chip Latencies

We perform our simulations for 70 nm technology, with a 5 GHz clock frequency. with an 8 MB on-chip L2 cache. We maintain inclusion between L1 and L2 caches. Our L2 cache configura-

Table 2: Multiprogrammed Workloads

Workload	Benchmarks
MIX1	apsi, art, equake, mesa
MIX2	ammp, swim, mesa, vortex
MIX3	apsi, mcf, gzip, mesa
MIX4	ammp, gzip, vortex, wupwise

tion is substantially more aggressive than existing CMP proposals, such as Sun Gemini and IBM Power5, which have 1 MB and 1.9 MB capacity respectively for 2-core CMPs [25,12]. Because 2 MB private caches are adequate for most SPEC2K benchmarks, comparing CMP-NuRAPID to that configuration actually makes our results conservative for multiprogrammed workloads.

Our base configuration is a 4-core CMP with 8-MB, 32-way conventional shared L2 cache (hereafter referred to as "uniform-shared cache") with 128-B blocks, and 4 ports to provide equal bandwidth to private caches. To provide a conservative base case, however, the latency of the uniform-shared cache is based on a faster 8-way cache with only 1 port. For private caches, we use four 2-MB, 8-way caches each with a single port. For CMP-NuRAPID evaluation, we assume an 8 MB, 8-way CMP-NuRAPID with 4 single-ported d-groups. We also show results for CMP-SNUCA from [6] (hereafter referred to as "non-uniform-shared cache"). CMP-SNUCA is similar to Piranha's banked cache [4]. We obtain the latencies for CMP-SNUCA from [14] and [6]. We do not evaluate CMP-DNUCA from [6], because [6] shows realistic CMP-DNUCA to perform *worse* than CMP-SNUCA. We model both the bandwidth and latency of on-chip caches carefully.

We modify Cacti [22] version 3.2 to derive the access times and wire delays for our conventional caches and for each d-group in CMP-NuRAPID. Because Cacti is not generally used for monolithic large caches (e.g., greater than 4 MB), we make the following modifications based on those proposed in [8]: 1) Treat each of our d-groups (1 to 2 MB) as independent (although tagless) caches and optimize for subarray geometry and access time; 2) Account for the wire delay to reach each d-group based on the distance to route around any closer d-groups using the RC wire-delay models in Cacti; and 3) Separately optimize our split tag arrays (or unified for the shared and private caches) for access time. We used our modified Cacti to verify CMP-SNUCA latencies from [14,6].

The computed latencies for our caches are shown in Table 1 from the perspective of core *P0* in Figure 1. (The results are symmetric for the other cores.) Note that the tag latency of the shared cache is particularly high because of RC wire delay to reach the shared tag, which must be placed centrally in the chip to minimize latency among the cores. We aggressively assume the data from the shared cache can be routed directly to the cores instead of through a central controller (such as the tag), hence the comparatively low latency for the data arrays.

We model an on-chip split transaction bus. We assume that the bus latency is the latency that would be required for a core to access the farthest tag array, which involves a large, long, global RC-wire delay. We ignore other overheads which may increase bus latency. Because private caches have more frequent bus transactions due to coherence misses than CMP-NuRAPID, ignoring overheads in bus latency helps private caches.

Table 3: Multithreaded Workloads

Online Transaction Processing (OLTP): We use the Open Source Development Labs Database Test 2 v0.23 (OSDL-DBT-2) [20, 29, 30] which is derived from the TPC-C specification revision 5.0. This test models many users performing 5 types of transactions with a wholesale supplier. We run the PostgreSQL database server version 7.4.5 compiled from source as specified for the DBT-2. We simulate 128 users, with all keying and thinking times set to 0, accessing a 1.2 GB, 10-warehouse database. We simulate 100 transactions after a 300-transaction warm-up.
Static Web Server: Apache: We run the Debian distribution of apache 1.3.31. We use SURGE [3] to generate web requests from a 30,000-file, 700MB repository. We simulate 128 users and 500 requests after a 2000-request warm-up.
Java Server: SPECjbb2000: SPECjbb is a java-server workload focusing on online transaction processing in middleware. We simulate 4-warehouses with a 30-second (simulated time) warm-up and a 10-second sample. We use the Blackdown JVM for Linux version 1.3.1.
SPLASH-2: We use two applications from SPLASH-2 [31], **ocean** and **barnes-hut**, as representative scientific workloads. These applications are compiled with a p-threads implementation of the PARMACS macros provide by [2]. We run barnes-hut (16K bodies) and ocean (514 x 514) to completion.

4.3 Workloads

Details of our commercial and scientific workloads are shown in Table 3. We account for the variability in multithreaded workloads [1] by doing multiple simulation runs for each benchmark in each configuration and injecting random perturbations in memory system timing for each run. We construct multiprogrammed workloads from 10 SPEC2K applications [27] as shown in Table 2. The applications are compiled with the default options provided with the SPEC tools. We found these workloads to be representative among a broader set of simulations. For each workload, we run until at least one core completes 1 billion instructions.

5 Results

In Section 5.1, we present the results for controlled replication (CR) and in-situ communication (ISC) using our multithreaded workloads. In Section 5.2 we evaluate the effectiveness of capacity stealing (CS) using our multiprogrammed workloads.

5.1 Multithreaded Workloads

In this section, we first analyze the sharing characteristics of our workloads and the opportunity for performance improvement. Next, we discuss the results for CR and ISC separately. Finally, we show performance results for the two together.

5.1.1 Workload Characteristics and Opportunity

In this section, we first characterize the cache access distribution in shared and private caches. Next, we quantify the opportunity for performance improvement.

Figure 5 shows the distribution of L2 cache accesses. The bars from left to right represent the different types of cache accesses as fractions of overall cache accesses for shared and private caches respectively. Note that the y-axis scale starts from 0.5 to show the

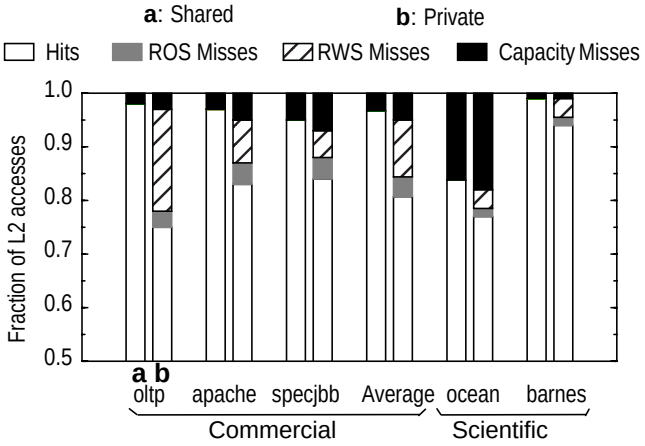


FIGURE 5: Distribution of Cache Accesses

distributions clearly. We show workloads on the X-axis in a decreasing order of sharing. Because commercial workloads have more sharing, they appear before scientific workloads. This ordering makes it easier to show the effects of sharing trends. We also show average across commercial workloads. We categorize the cache accesses into: 1) hits, 2) misses due to read-only sharing (ROS misses), 3) misses due to read-write sharing (RWS misses), and 4) capacity misses. We count a miss as a ROS miss when another copy of the block exists in shared state, and as a RWS miss when a dirty copy of the block already exists. Shared cache has only hits and capacity misses, while private caches have all four types of accesses.

Private caches incur more capacity misses than shared caches in all workloads. As mentioned in Section 3.1, the uncontrolled replication of data in private caches decreases effective capacity, causing more capacity misses than a shared cache. On average, across all commercial workloads, shared and private caches have 3% and 5% capacity misses, respectively.

Private caches experience more ROS and RWS misses in commercial workloads than in scientific workloads. The reason for this trend is that commercial workloads have more extensive data sharing. Among commercial workloads, misses in OLTP are dominated by RWS misses, while apache and specjbb have all types of misses. Because CR and ISC target ROS and RWS, they could reduce these misses a lot.

Figure 6 shows the performance of non-uniform-shared, private, and ideal caches normalized with respect to the conventional uniform-shared cache. The ideal cache is a shared cache with the same latency as that of each private cache. Thus the ideal cache has the capacity advantages of shared and latency advantages of private caches. The ideal cache results represent the upper bound on performance improvement achievable by CMP-NuRAPID. As mentioned in Section 4, the non-uniform-shared cache is like the SNUCA design from [6].

There is significant performance improvement opportunity in all the workloads. Commercial workloads exhibit more performance improvement opportunity than scientific workloads. On average, the ideal cache performs 17% better than the uniform-shared cache in commercial workloads. The corresponding performance improvements for non-uniform-shared and private caches are 4% and 5% respectively. Compared to the uniform-shared cache, both non-uniform-shared and private caches have latency advantage. However, they fail to close the gap between the uniform-shared and ideal caches significantly.

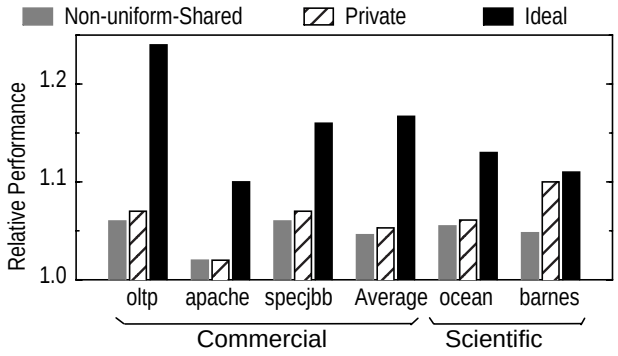


FIGURE 6: Performance Opportunity

The performance improvement for non-uniform-shared does not change significantly as we move from commercial to scientific workloads. Because non-uniform-shared only improves upon the latency of uniform-shared, differences in sharing do not impact the performance improvement for non-uniform-shared.

Private caches perform better in scientific workloads than in commercial workloads due to more frequent ROS and RWS misses as shown in Figure 5.

5.1.2 Controlled Replication and In-situ Communication

In this section, we analyze the effectiveness of CR and ISC. First, we discuss the block reuse patterns for ROS and RWS data to verify the decisions made for CR and ISC in Section 3.1 and Section 3.2. Next, we analyze the tag and data arrays' access distributions for the two optimizations.

Figure 7 shows the block reuse patterns for different types of sharing. The left bars represent the fraction of all replacements where the replaced block was brought into the private cache on an ROS miss and the block was reused *n* times before being replaced; *n* varies as 0, 1, 2 to 5, and more than 5. The right bars represent similar information for invalidations where the invalidated block was brought in on an RWS miss.

For ROS data, private caches replace many blocks before reusing them even once. On average, across commercial workloads 42% blocks are replaced without reuse. Thus, controlling replication by not copying the data on the first use (Section 3.1) saves capacity. Figure 7 also shows that 50% of the blocks are reused at least twice. This result confirms CMP-NuRAPID's approach of copying the data on second use (Section 3.1).

For RWS data, most of the blocks are invalidated before five or fewer reuses. On average, only 8% of the RWS blocks are reused more than five times before being invalidated by a writer. These

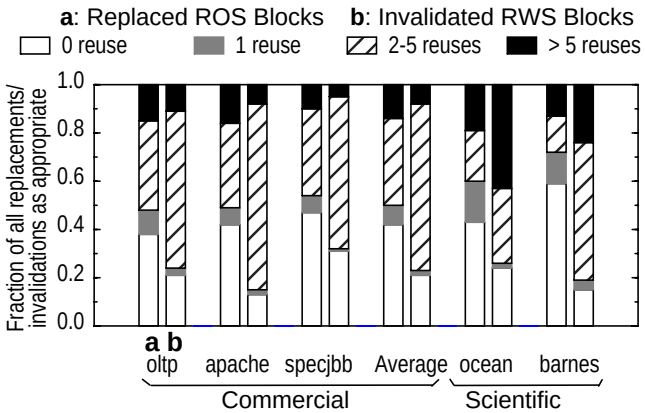


FIGURE 7: Reuse Patterns

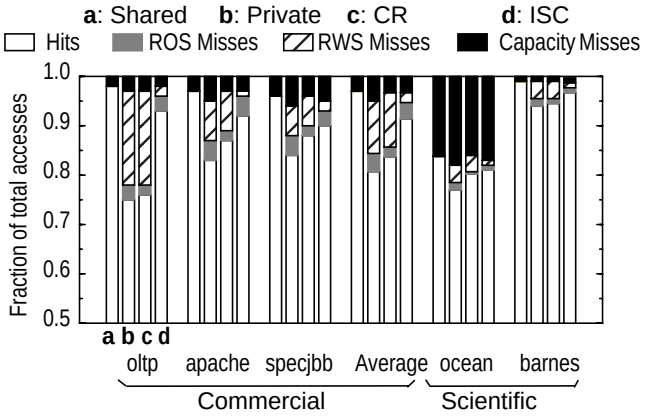


FIGURE 8: Distribution of Tag Array Accesses

invalidations cause frequent RWS misses as shown in Figure 5. On average, 69% of the blocks are reused between 2 and 5 times. CMP-NuRAPID's policy of keeping the read-write-shared block close to the reader (Section 3.2) allows fast access to these blocks.

Figure 8 shows the distribution of tag array accesses. The bars from left to right represent the access distribution as a fraction of overall accesses for shared, private, CMP-NuRAPID with CR, and CMP-NuRAPID with ISC respectively. Note that the y-axis scale starts from 0.5 to show the distributions clearly. The numbers for shared and private caches are the same as the ones already shown in Figure 5.

CR decreases both capacity misses and ROS misses. Averaging across commercial workloads, CR results in 3% capacity misses as compared to 5% capacity misses in private caches (a 40% reduction), and 2% ROS misses as compared to 4% ROS misses in private caches (a 50% reduction). Better capacity utilization allows CR to have almost the same number of capacity misses as in the shared cache.

ISC significantly decreases RWS misses. On average across all commercial workloads, private caches have 10% RWS misses, while CMP-NuRAPID with ISC has only 2%; a reduction of 80%. Because the RWS misses dominate ROS misses in most of the workloads, and ISC targets RWS misses, ISC results in more hits in the tag array than CR.

Figure 9 shows the distribution of data array accesses. as a fraction of overall cache accesses for CR and ISC respectively. The data array access distribution for private and shared caches are the same as the tag array access distributions already shown in Figure 5 and Figure 8, and are not repeated in Figure 9.

CR results in more accesses to the closest d-group than ISC.

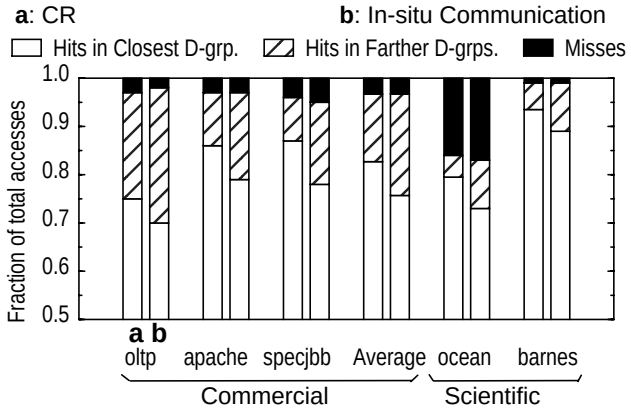


FIGURE 9: Distribution of Data Array Accesses

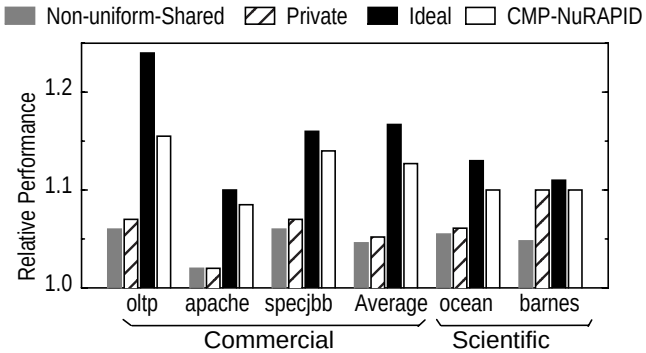


FIGURE 10: Performance

On average, CR and ISC have 83% and 76% hits to closest d-group for commercial workloads. In comparison, private caches have 84% hits in the data array as shown in Figure 8. The reason for more farther d-group accesses in ISC is that the writer needs to access the block in a farther d-group on every write to RWS data. However, this strategy results in reduced RWS misses in the tag array, as we showed in Figure 8.

In the interest of space, we do not show the tag and data array access distributions for CMP-NuRAPID with both the optimizations used together. We found that, when both the optimizations are used together, ROS misses and capacity misses are almost equal to those for CR in Figure 8, while RWS misses are equal to those for ISC in Figure 8. The data array access distribution is the same as that for ISC in Figure 9.

5.1.3 Performance

In this section, we evaluate the performance of CMP-NuRAPID with both CR and ISC. We expect CMP-NuRAPID to outperform shared and private caches because of less latency and better capacity utilization respectively.

Figure 10 shows the performance of non-uniform-shared, private, ideal, and CMP-NuRAPID normalized with respect to the performance of uniform-shared cache. We use number of transactions per second as our performance metric. We already discussed the results for non-uniform-shared, private and ideal in Section 5.1.1. We only focus on CMP-NuRAPID in this section.

CMP-NuRAPID outperforms both non-uniform-shared and private caches in all the workloads. For commercial workloads, CMP-NuRAPID performs 13% better than uniform-shared on average. In comparison, the corresponding performance improvements for non-uniform-shared and private caches are 4% and 5% respectively. Non-uniform-shared does not work well because there is no replication or migration (as mentioned before, [6] shows that migration does not improve performance). In contrast, CMP-NuRAPID employs CR. Indeed, [6]'s negative result on migration is not surprising because replication is much more important than migration for these workloads with heavy sharing. Private caches do not perform well due to limited capacity.

The maximum performance improvement for CMP-NuRAPID is in OLTP, where CMP-NuRAPID outperforms uniform-shared by 16%. Non-uniform-shared and private caches perform 6% and 7% better than uniform-shared respectively.

The performance advantage of CMP-NuRAPID relative to the private caches decreases as we move from commercial to scientific workloads. Due to less sharing in scientific workloads as shown in Figure 5, private caches have less ROS and RWS misses, decreasing the opportunity for CMP-NuRAPID. For example, in barnes,

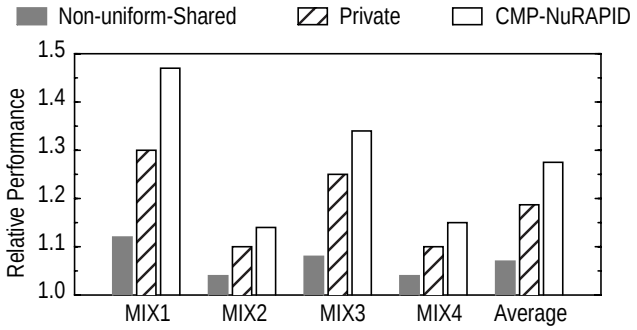


FIGURE 12: Performance

both CMP-NuRAPID and private caches perform 10% better than non-uniform-shared cache.

On average, CMP-NuRAPID performs within 3% of the ideal cache performance across the commercial workloads. The maximum performance gap between CMP-NuRAPID and ideal is in OLTP where CMP-NuRAPID performs 8% worse than ideal. The main reason for this performance gap is the high percentage of remote d-group accesses in OLTP, as was shown in Figure 9. Despite the gap between CMP-NuRAPID's and ideal's performance, CMP-NuRAPID significantly outperforms both non-uniform-shared and private caches in OLTP.

5.2 Multiprogrammed Workloads

In this section, we analyze CMP-NuRAPID for multiprogrammed workloads. First, we evaluate CMP-NuRAPID's capacity utilization. Then, we compare the performance of CMP-NuRAPID with other designs. We expect CMP-NuRAPID to have performance advantage over shared caches (both uniform and non-uniform) due to less latency and over private caches due to CS. In the interest of space, we do not show results for the individual benchmarks in the workloads and only show overall results for complete workloads.

5.2.1 Distribution of Cache Accesses

Figure 11 shows the distribution of cache accesses for shared, private, and CMP-NuRAPID caches respectively. The right most bars show the averages across all the workloads. We do not separate ROS and RWS misses (We found such misses to be insignificant in multiprogrammed workloads due to negligible sharing).

CMP-NuRAPID incurs slightly more misses then shared cache, but significantly less misses than private caches. On average, shared cache, private caches, and CMP-NuRAPID have 8.9%, 14%, and 9.7% miss rates respectively. CS and extra tag space enable CMP-NuRAPID to utilize the cache capacity more efficiently, resulting in less miss rates as compared to the private caches. CMP-NuRAPID has slightly higher miss rates than shared cache due to less tag capacity available to each core and the random choice of d-group for distance replacement (Section 3.3).

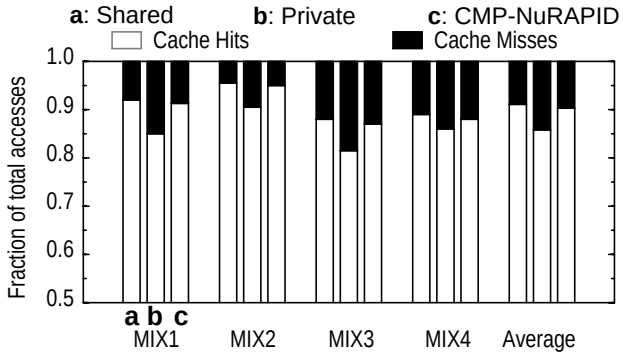


FIGURE 11: Distribution of Cache Accesses

We found that, on average, across all the multiprogrammed workloads, 85% of accesses (93% of all hits) in CMP-NuRAPID hit in the closest data d-group. We do not discuss these results in detail in interest of space. However, these results demonstrate the effectiveness of CS in keeping frequently-accessed data close to the processor.

5.2.2 Performance

Figure 12 shows the performance for different designs in terms of instructions per cycle (IPC). The bars from left to right represent the IPC for non-uniform-shared, private, and CMP-NuRAPID caches with respect to the uniform-shared cache respectively.

CMP-NuRAPID is clearly the best. On average, non-uniform-shared, private, and CMP-NuRAPID caches perform 7%, 19%, and 28% better than the uniform-shared cache.

CMP-NuRAPID outperforms non-uniform-shared by 20% on average across all the workloads. As shown in Figure 11, the miss rates for CMP-NuRAPID are only slightly higher than shared caches. But CMP-NuRAPID has significantly lower latency than a non-uniform-shared cache. This lower latency allows CMP-NuRAPID to outperform non-uniform-shared cache.

CMP-NuRAPID shows 8% performance improvement as compared to private caches. Comparing Figure 10 and Figure 12, it is worth noting that private caches have more performance advantage over shared caches in multiprogrammed than in multithreaded workloads. The reason for this trend is that negligible sharing in multiprogrammed workloads allows private caches to avoid ROS and RWS misses. However, CMP-NuRAPID still outperforms private caches in multiprogrammed workloads due to better capacity utilization.

6 Related Work

We previously discussed [6], which analyzes non-uniform shared cache designs. [19] is the first proposal of a CMP design using a shared, on-chip L2 cache. Many commercial CMPs [26,12,4, 25] use a shared L2 cache. Several papers have also examined large caches in production uniprocessors. The Itanium II uses a large, low-bandwidth L3 cache that is optimized for size and layout efficiency [28, 18].

Many papers have proposed cache coherence protocols and optimizations for SMPs (see [9] for details). Node capacity (*not* on-chip capacity) versus latency tradeoff for DSMs has been exploited by COMA[24] and R-NUMA[11].

Three recent proposals have examined uniform-access caches in CMPs. [15] compares shared caches, private caches, and a software-managed technique for partitioning capacity in a shared cache. However, the paper does not address shared cache's long latency and uses coarse-grain partitions managed by the compiler or OS. In contrast, CMP-NuRAPID uses fine-grain cache-block granularity to allow flexible sharing transparent to software. [17] evaluates migrating execution (as opposed to data) across cores to improve throughput but assumes only one program running on a 4-core CMP with private caches (i.e. 3 cores are idle). [7] proposes a conflict-predictor for multiprogrammed workloads running on a shared cache. Their technique does not address the capacity-latency tradeoff that CMP-NuRAPID exploits.

Some recent papers have proposed designs to address the wire delay problem in uniprocessor caches. We previously mentioned NUCA [14] and NuRapid [8]. Transmission Line Cache (TLC) proposes to replace long wires in large uniprocessor caches with LC transmission lines for reducing wire delay [5, 6]. TLC achieves latency reduction at the cost of substantial increase in bandwidth requirement and added complexity. CMP-TLC is orthogonal to CMP-NuRAPID.

7 Conclusions

Chip multiprocessors (CMPs) substantially increase capacity pressure on the on-chip cache hierarchy, while requiring fast access. Neither private nor shared caches can provide both large capacity and fast access. CMPs create a new opportunity for exploring capacity-latency trade-off which does not exist in SMPs and DSMs. We proposed three novel ideas to exploit this opportunity: (1) Though placing a copy close to each requestor allows fast data access for read-only sharing, the copies increase pressure on the already-limited on-chip capacity in CMPs. We propose controlled replication to conserve capacity by not making copies in some cases. (2) CMPs allow fast on-chip communication between processors for read-write sharing. While private caches incur slow accesses to read-write shared data through coherence misses, we propose in-situ communication to provide fast access without making copies or incurring coherence misses. (3) Accessing neighbors' caches is not expensive in CMPs. We propose capacity stealing to place private data that exceeds a core's capacity in a neighboring cache with less capacity demand.

We proposed CMP-NuRAPID, a hybrid of private, per-processor tag arrays and a shared data array to incorporate the two ideas. To mitigate the slow access to shared data array, we employ non-uniform cache access and distance associativity from previous proposals to keep frequently-accessed data in regions close to the processor. Our results showed that for a 4-core CMP with 8 MB on-chip capacity, CMP-NuRAPID improves performance by 13% over a shared cache and by 8% over private caches for three commercial multithreaded workloads. For four multiprogrammed workloads, CMP-NuRAPID performs 28% better than a shared cache and 8% better than a private cache.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA 9*, pp 7–18, Feb. 2003.

[2] E. Artiaga, X. Martorell, Y. Becerra, and N. Navarro. Experiences on implementing Parmacs macros to run the Splash-2 suite on multiprocessors. Technical Report UPC-DAC-1998-1, Department of Computer Architecture Universittat Politecnica de Catalunya, Jan. 1998.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[4] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *the 27th ISCA*, pages 282–293, June 2000.

[5] B. M. Beckmann and D. A. Wood. TLC: Transmission line caches. In *MICRO 36*, pages 43–54, Dec. 2003.

[6] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37*, pages 319-330, Dec. 2004.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting the inter-thread cache contention on a chip multiprocessor architecture. In *HPCA 11*, pages 340-351, Feb. 2005.

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *MICRO 36*, pages 55–66, Dec. 2004.

[9] J. P. Singh, D Culler, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[10] J. H. Edmondson and et al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1), 1995.

[11] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *the 24th ISCA*, pages 229–240, June 1997.

[12] S. Finnes. *iseries.myseries*. http://www-1.ibm.com/servers/uk/media/iseries_skillbuilder/POWER5DeliverWith% outDisruption1.pdf, 2004.

[13] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *the 27th ISCA*, pages 107–116, June 2000.

[14] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS X*, pages 211–222, Oct. 2002.

[15] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA 10*, pages 176-185, Feb. 2004.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, and G. Hallberg. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[17] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA 10*, pages 186-197, Feb. 2004.

[18] S. D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. Sullivan, and T. Grutkowski. The implementation of the Itanium 2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1448–1460, Nov. 2002.

[19] K. Olukotun, B. A Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS VII*, pages 2-11, 1996.

[20] Open Source Development Labs. Open source development labs database test 2. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/o% sdl_dbt-2/.

[21] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *the 11th ISCA 84*, pages 348–354, 1984.

[22] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, Aug. 2001.

[23] P. Stenstrom, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal. Trends in shared memory multiprocessing. *IEEE Computer*, 30(12):44–50, Dec. 1997.

[24] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *the 19th ISCA*, pages 80–91, 1992.

[25] Sun Microsystems. Sun's 64-bit gemini chip. *Sunflash*, 66(4), Aug. 2003.

[26] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. IBM eserver Power4 System Microarchitecture. IBM White Paper, Oct. 2001.

[27] The Standard Performance Evaluation Corporation. Spec CPU2000 suite. http://www.specbench.org/osg/cpu2000/.

[28] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1523–1529, Nov. 2002.

[29] M. Wong. Stressing linux with real-world workloads. In *Linux Symposium*, pages 495–504, July 2003.

[30] M. Wong, J. Zhang, C. Thomas, B. Olmstead, and C. White. Open source development labs database test 2 differences from the tpc-c, version 0.15. http://www.osdl.org/docs/dbt_2_differences_from_tpc_c.pdf, June 2002.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *the 22nd ISCA*, pages 24–36, July 1995.