Cooperative Caching for Chip Multiprocessors

Jichuan Chang and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin-Madison
E-mail: {chang, sohi}@cs.wisc.edu

Abstract

This paper presents CMP Cooperative Caching*, a unified framework to manage a CMP's aggregate on-chip cache resources. Cooperative caching combines the strengths of private and shared cache organizations by forming an aggregate "shared" cache through cooperation among private caches. Locally active data are attracted to the private caches by their accessing processors to reduce remote on-chip references, while globally active data are cooperatively identified and kept in the aggregate cache to reduce off-chip accesses. Examples of cooperation include cache-to-cache transfers of clean data, replication-aware data replacement, and global replacement of inactive data. These policies can be implemented by modifying an existing cache replacement policy and cache coherence protocol, or by the new implementation of a directory-based protocol presented in this paper.*

Our evaluation using full-system simulation shows that cooperative caching achieves an off-chip miss rate similar to that of a shared cache, and a local cache hit rate similar to that of using private caches. Cooperative caching performs robustly over a range of system/cache sizes and memory latencies. For an 8-core CMP with 1MB L2 cache per core, the best cooperative caching scheme improves the performance of multithreaded commercial workloads by 5-11% compared with a shared cache and 4-38% compared with private caches. For a 4-core CMP running multiprogrammed SPEC2000 workloads, cooperative caching is on average 11% and 6% faster than shared and private cache organizations, respectively.

1 Introduction

Future chip multiprocessors (CMPs) will both require, as well as enable, innovative designs of on-chip memory hierarchies. The combination of a higher demand from more processing cores (especially if the number of cores increases at a faster rate than on-chip storage), the likely scaling of the working set sizes of future workloads, and the increasing costs of off-chip misses will place significant demands on the organization of on-chip storage. This storage will need to be organized to maximize the number of requests that can be serviced on chip [14, 20, 26]. Increasing on-chip wire delays will create additional, and perhaps conflicting, demands on how this storage should be organized: on-chip data should be kept in storage that is close to the processor(s) working on it to minimize the access latency. This collection of potentially conflicting yet very important demands call for novel organizations and management of the on-chip storage, so that both goals can be achieved simultaneously.

Proposals for the on-chip memory hierarchy of CMPs have borrowed heavily from the memory hierarchies of traditional multiprocessors. Here, each processor has private L1 data and instruction caches, and the L2 caches can either be shared or private. A shared L2 cache organization is preferable if reducing the collective number of L2 misses is important, whereas an organization with private L2 caches is preferable if reducing the L2 access latency and cache design complexity is important.

Most CMP proposals use the private L1 cache structures of traditional multiprocessors, although the L1 cache may not use inclusion [4]. The interesting question has to do with the organization of on-chip L2 cache resources. Proposed CMP designs [5, 19, 32] have emphasized minimizing the number of off-chip accesses and thus have used a banked, shared L2 cache organization. In such designs, data are spread evenly across all banks and only a fraction of references are typically satisfied by the banks that are close to the requesting processor. Hence the average L2 cache access latency is heavily influenced by the latency of accessing remote cache banks, which in turn is influenced by on-chip wire delays. While a shared cache may be a good choice when on-chip wire delays are not significant, it can rapidly become a poor choice when such delays increase.

Private L2 caches have the advantage that most L1 misses can be handled locally, so the number of remote on-chip L2 cache accesses is reduced. This allows private L2 cache based designs to be more tolerant of increasing on-chip wire latencies. However, due to inefficient use of the aggregate L2 cache capacity, such designs typically result in many more off-chip accesses than a shared L2 cache, and thus they have not been favored by CMP cache designers.

Recently, there have been several proposals for hybrid CMP cache designs that attempt to achieve the low access latency advantages of private L2 caches and the low off-chip miss rates of shared L2 caches [6, 8, 35]. These proposals typically start out with a banked shared cache and attempt to keep data that is likely to be accessed on an L1 miss (e.g., L1 cache victims) close to where it is being consumed, i.e., in the local L2 bank or other proximate L2 banks. Some proposals further use both data migration and replication to optimize important sharing patterns [6, 8], but they require significant hardware changes and do not allow adaptive control of such optimizations.

Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)
0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

We believe that on-chip cache hierarchies with private L2 caches deserve fresh thought. In addition to their latency benefits, private L2 cache designs have several other potential benefits that are likely to be of increasing importance for future CMPs. These include: (1) private cache based designs lend themselves to easier implementation of performance isolation [18, 33], priority and QoS [16, 34], which traditionally have been assumed or needed by applications and operating systems; (2) compared with a shared cache whose directory information is stored with cache tags and distributed across all banks, a private cache is more self-contained and thus serves as a natural unit for resource management (e.g., power off to save energy); (3) instead of building a highly-associative shared cache to avoid inter-thread contention, the same set-associativity is available for an aggregate cache formed by the private caches, each with much lower set-associativity, thus reducing power, complexity and latency overhead; (4) the bandwidth requirement of the cross-chip interconnection network is generally much lower with private L2 caches, and this leads to a simpler and potentially higher performing network. But partaking of these potential advantages first requires a solution to the major, and predominant, drawback of private L2 cache designs: the larger number of off-chip cache misses compared to shared L2 cache designs.

This paper presents *CMP Cooperative Caching (CC)*, a proposal for organizing the on-chip hierarchy to achieve the benefits of both private and shared cache designs. Using private L2 caches as the basic design point, cooperative caching manages the aggregate on-chip L2 cache resources so that they cooperate to overcome the inefficient use of resources and associated limitations of traditional private L2 cache designs. Cooperative caching uses a collection of mechanisms that attempt to keep data in the aggregate on-chip L2 cache resources as much as possible, thereby achieving much of the capacity benefits of equivalent-sized shared L2 cache designs. For example, one mechanism used by cooperative caching is to put a locally evicted block in another on-chip L2 cache that may have spare capacity, rather than evict it from the on-chip hierarchy entirely. We demonstrate that using private L2 caches to form a cooperatively managed aggregate L2 cache can achieve the latency benefits of private L2 caches and the capacity benefits of shared L2 caches, for a range of CMP configurations.

The rest of the paper is organized as follows. In section 2 we explain the cooperative caching idea and different cooperation policies that private caches can use to achieve the capacity advantages of a shared cache. Section 3 discusses possible implementations of cooperative caching, including a central directory-based design. Section 4 covers our performance evaluation using full-system simulation with an out-of-order processor model. Related work is discussed in Section 5 and we conclude in Section 6.

2 CMP Cooperative Caching

2.1 Background

The basic concept of CMP cooperative caching is inspired by software cooperative caching algorithms [9], which have been proposed and shown to be effective in the context of file and web caching [9,10]. Although physically separate, individual file/web caches cooperatively share each other's space to meet their dynamic capacity requirements, thus reducing the traffic to the remote servers. In a similar vein, CMP cooperative caching tries to create a globally-managed, "shared," aggregate on-chip cache via the cooperation of the different on-chip caches, so that the aggregate cache resources can better adapt to the dynamic caching demands of different applications. To simplify the discussion in this paper, we will assume a CMP memory hierarchy with private L1 instruction and data caches for each processing core and banks of L2 caches. The ideas of cooperative caching are equally applicable if there are more levels of caches on the chip.

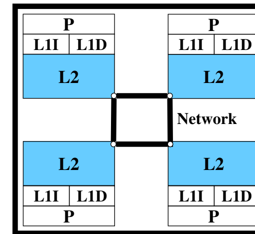

Figure 1. Cooperative Caching (The shaded area represents the aggregate shared cache formed via cooperation.)

Figure 1 depicts the cooperative caching concept for a CMP with four cores. Each core's local L2 caches are physically close to it, and privately owned such that only the processor itself can directly access them. Local L2 cache misses can possibly be served by remote on-chip caches via cache-to-cache transfers as in a traditional cache coherence protocol. Cooperative caching will further attempt to make the ensemble of private L2 caches (the shaded area in Figure 1) appear like a collective shared cache via cooperative actions. For example, cooperative caching will attempt to use remote L2 caches to hold (and thus serve) data that would generally not fit in the local L2 cache, if there is spare space available in a remote L2 cache.

Cache cooperation, for example to share the aggregate cache capacity, is a new hardware caching opportunity afforded by CMPs. As we shall see, cooperation between caches will require the exchange of information and data between different caches, under the control of a cache coherence engine. Such an exchange of information and data, over and above a basic cache coherence protocol, was not considered practical, or fruitful, in multiprocessors built with multiple chips. For example, for a variety of reasons, when a data block is evicted from the L2 cache of one processor, it would not be placed in the L2 cache of another processor. The complexity of determining which L2 cache to place the evicted block into, and transferring the block to its new L2 cache home, would be significant. Moreover, this additional complexity would provide little benefit, since the latency of getting a block from memory would typically be lower than getting it from another L2 cache. The situation for CMP on-chip caches is very different: the transfer of information and data between on-chip L2 caches can be done relatively easily and efficiently, and the cost of an

Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)
0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

off-chip miss is significant. This situation is analogous to web caches: the cost of communicating information and data among web caches, so as to reduce the total number of misses, is typically much lower than the cost of servicing the misses from the remote server.

2.2 Cooperative Caching Framework

We view cooperative caching as a unified framework for organizing the on-chip cache resources of a CMP. Two major applications for CC immediately come to mind: (1) optimizing the average latency of a memory request so as to improve performance, and (2) achieving a balance between dynamic sharing of cache resources and performance isolation, so as to achieve better performance and quality of service in a throughput-oriented environment. We focus on the first application in this paper and leave the second one for future work, though some aspects of the second will appear in the evaluation.

Cooperative caching tries to optimize the average latency of a memory request by combining the strengths of private and shared caches adaptively. This is achieved in three ways: (1) by using private caches as the baseline organization, it attracts data locally to reduce remote on-chip accesses, thus lowering the average on-chip memory latency; (2) via cooperation among private caches, it can form an aggregate cache having similar effective capacity to a shared cache, to reduce costly off-chip misses; (3) by controlling the amount of cooperation, it can provide a spectrum of choices between the two extremes of private and shared caches, to better suit the dynamic workload behavior.

2.3 Policies to Reduce Off-chip Accesses

The baseline organization of private L2 caches allows for low latency access to data that can be contained in a processing core's private L2 cache. We now consider how the multiple L2 caches can cooperate to allow better use of aggregate on-chip cache resources and thereby reduce the number of off-chip accesses. We present three policies for cooperation. The first policy facilitates cache-to-cache transfers of on-chip "clean" blocks to eliminate unnecessary off-chip accesses to data that already reside elsewhere on the chip. The second policy replaces replicated data blocks to make room for unique on-chip copies (called *singlets*), thereby making better use of the on-chip cache resources. The third policy allows singlet blocks evicted from a local L2 cache to be placed in another L2 cache, possibly taking the cache slot of an inactive data block, thereby keeping potentially more useful data on the chip.

2.3.1 Cache-to-cache Transfers of Clean Data

In a hierarchy with private L2 caches, a memory access can be avoided on an L2 cache miss if the data can be obtained from another cache. Such inter-cache data sharing (via cache-to-cache transfers) can be viewed as a simple form of cooperation, usually implemented by the cache coherence protocol to ensure correct operation. Except for a few protocols that have the notion of a "clean owner,"[1] modern multiprocessors employ variants of invalidate-based coherence protocols that only allow cache-to-cache transfers of "dirty" data (meaning that the data was written but has not been stored back to the lower level storage). If there is a miss on a block that only has clean copies in other caches, the lower-level storage (usually the main memory) has to respond with the block, even though it is unnecessary for correct operation and can be more expensive than a cache-to-cache transfer in a CMP.

There are two main reasons why coherence protocols employed by traditional multiprocessors do not support such sharing of clean copies. First, cache-to-cache transfer requires one and only one cache to respond to the miss request to ensure correctness, however maintaining a unique clean owner is not as straightforward as maintaining a dirty owner. Second, in traditional multiprocessors, off-chip communication is required whether sourcing the clean copy from another cache or from the main memory, but the latency savings of the first option is often not big enough to justify the complexity it adds to the coherence protocol. In fact, in many cases obtaining the data from memory can be faster than obtaining it from another cache.

However, CMPs have made off-chip accesses significantly more costly than on-chip cache-to-cache transfers; currently there is an order of magnitude difference in the latencies of the two operations. Furthermore, a high percentage of misses in commercial workloads can be satisfied by cache-to-cache transfers of clean data, due to frequent misses to (mostly) read-only data (especially instructions). These factors make it more appealing to let caches share clean data.[2] Our simulation results show that for commercial workloads, sharing clean data can reduce 10-50% of off-chip misses, with only a slight increase in traffic in the network connecting the on-chip caches.

2.3.2 Replication-aware Data Replacement

The baseline private L2 caches employed by cooperative caching allow replication of the same data block in multiple on-chip caches. When cache capacity exceeds the program's working set size, replication reduces cache access latency because more requests can be satisfied by replicated local copies. However, when cache size is dwarfed by the working set size, replicate blocks will compete for the limited capacity with unique copies. Cooperative caching uses replication-aware data replacement to optimize capacity by discriminating against replicated blocks in the replacement process. This policy aims to increase the number of on-chip unique blocks, thus improving the probability of finding a given block in the aggregate on-chip cache.

We define a cache block in a valid coherence state as a *singlet* if it is the only on-chip copy, otherwise it is a *replicate* because replications exist. Cooperative caching employs a simple policy to trade off between access latency and capacity: evict singlets only when no replicates are available as victims. The private cache's replacement policy is augmented to prefer evicting replicates. If all the blocks

[1] For example, the Illinois protocol [28], the EDWP protocol [2], and Token Coherence [23].

[2] For example, IBM Power4 systems [32] add two states (SL and T) to their MESI-based coherence protocol to partially support cache-to-cache transfers of clean data, but don't select follow-up owners when the initial owner replaces the block.

IEEE COMPUTER SOCIETY

in the private cache's candidate set are singlets, a victim is chosen by the default cache replacement policy (e.g., LRU).

A singlet block evicted from a cache can be further "spilled" into another cache on the chip. Using the aforementioned replacement policy, both invalidated and replicate blocks in the receiving cache are replaced first, again reducing the amount of replication. By giving priority to singlets, all private caches cooperate to evict blocks with other on-chip copies, and replace them with unique data blocks that may be used later by other caches. This increases the number of unique on-chip cache lines, and thus reduces the number of off-chip accesses.

Many options exist to choose the host cache for spilling; we randomly choose a host but give higher probabilities to close neighbors. This random algorithm needs no global coordination, and allows one cache's victims to reach all other on-chip caches without requiring rippled spilling. Keeping spilled singlets close to their evicting caches can reduce the spilling time and access latency for later reuse.

2.3.3 Global Replacement of Inactive Data

Spilling a victim into a peer cache both allows and requires global management of all private caches. The aggregate cache's effective associativity now equals the aggregate associativity of all caches. For example, 8 private L2 caches with 4-way associativity effectively offers 32-way set associativity for cooperative caching to exploit.

Similar to replication-aware data replacement, we want to cooperatively identify singlet but inactive blocks, and keep globally active data on-chip. This is especially important for multiprogrammed workloads with heterogeneous access patterns. Because these applications do not share data and have little operating system activity, almost all cached blocks are singlets after the initial warmup stage. However, one program with poor temporal locality may touch many blocks which soon become inactive (or dead), while another program with good temporal locality will have to make do with the remaining cache space, frequently evicting active data and incurring misses.

Implementing a global-LRU policy for cooperative caching would be beneficial but is difficult because all the private caches' local LRU information has to be synchronized and communicated globally. Practical global replacement policies have been proposed to approximate global age information by using reuse counters [29] or epoch-based software probabilistic algorithms [11]. In this paper we modify N-Chance Forwarding [9], a simple and fast algorithm from cooperative file caching research, to achieve global replacement.

N-Chance Forwarding was originally designed with two goals: it tries to avoid discarding singlets, and it tries to dynamically adjust each program's share of aggregate cache capacity depending on its activity. Each block has a recirculation count. When a private cache selects a singlet victim, it sets the block's recirculation count to N, and forwards it to a random peer cache to host the block. The host cache receives the data, adds it to the top of its LRU stack (for the chosen set) and evicts the least active block in its local cache. If a recirculating block is later evicted, its count is decremented and it is forwarded again unless the count becomes zero. If the block is reused, its recirculation count is reset to 0. To avoid a ripple effect where a spilled block causes a second spill and so on, receiving a spilled block is not allowed to trigger a subsequent spill.

The parameter N was set to 2 in the original proposal [9]. A larger N gives singlets more opportunities to recirculate through different private caches, and hence makes it more likely to replace a non-singlet without being reused before its eviction from the aggregate cache. CMP cooperative caching sets N to 1, as replication control is already employed. We call the modified policy *1-Fwd*.

1-Fwd dynamically balances each private cache's allocation between local data accessed by its own processor and global data stored to improve the aggregate cache usage. The active processors' local references will quickly force global data out of their caches, while inactive processors will accumulate global data in their caches for the benefit of other processors. This way, both capacity sharing and global age-based replacement are achieved.

2.4 Cooperation Throttling

Because cooperative caching uses the cache replacement/placement policy as the unified technique to control replication and achieve global cache management, it offers a single control point for cooperation throttling which allows it to adapt to a spectrum of sharing behaviors. A cooperation probability can be consulted upon cache replacement, to decide how often to use replication-aware data replacement instead of the baseline replacement policy, and a spill probability can decide whether to spill a singlet victim or not. If both probabilities are set to zero, cooperative caching defaults to private caches (albeit with support for cache-to-cache transfer of clean data). Its behavior moves towards an unconstrained shared cache as the probabilities increase.

2.5 Role of L1 caches and Inclusion

A discussion of the interactions between the L1 and L2 caches in cooperative caching is in order. Cache hierarchies with private L2 caches often employ inclusion, since this simplifies the implementation of the coherence protocol, especially in traditional multiprocessors. Maintaining inclusion implies that a block is invalidated in an L1 cache when it is evicted from or invalidated in the companion L2 cache, i.e., the L1 cache can only maintain a copy of data in the companion L2 cache. Because cooperative caching's objective is to create a global shared L2 cache from the individual private L2 caches, an arbitrary L1 cache needs to be able to cache any block from the aggregate L2 cache, and not only a block from the companion L2 cache bank. Maintaining inclusion in the traditional sense (i.e., between an L1 cache and its associated L2 cache) unnecessarily restricts this ability. Thus, for cooperative caching to be effective, inclusion is maintained between L1 caches and the aggregation of all L2 banks, while each L1 cache can be non-inclusive with respect to its associated L2 bank.

2.6 Summary

Cooperative caching uses private caches to form a globally managed aggregate cache via three optimizations:

IEEE COMPUTER SOCIETY

(1) making use of on-chip clean data; (2) replication-aware data replacement; and (3) global replacement of inactive data. Through cooperative cache replacement and spilling-based victim placement, the three optimizations are unified and can be integrated with any existing cache replacement policy and coherence protocol.

Using these optimizations, cooperative caching can reduce the number of off-chip accesses and their contribution to the average access latency. Although these optimizations also cause more L1 cache misses to be satisfied by remote on-chip caches, a private L2 cache's reference stream will also keep frequently used data in it to maintain a high local cache hit ratio. The general net effect of using private caches and cooperative optimizations is higher performance than can be achieved with either shared or private cache organization.

3 Hardware Implementation

In this section we present our hardware implementation of cooperative caching functions by exploiting a CMP's high-bandwidth, low-latency on-chip communication and flexible on-chip topology. The requirements for cooperative caching are described in Section 2 and summarized below.

- The cooperative private caches need to maintain cooperation related hints for replacement, including whether a cache block is a singlet and the reuse status of a spilled block (required by our 1-Fwd policy).
- The coherence protocol needs to support cache-to-cache transfers of clean blocks, and spilling, a form of data migration among private caches for capacity sharing.

We believe that these requirements can be met by modifying various existing implementations. The private caches can add two bits to each tag, indicating whether the block was once spilled but has not been reused (for 1-Fwd) and whether the block is a singlet (for replication-aware data replacement). The singlet bit can be maintained by private caches by either observing other caches' fetch and eviction activities via a snooping protocol, or by receiving notification from a directory when duplication is removed. Because these bits are only used for victim selection, having no correctness implication, they can be imprecise.

On the protocol side, a clean owner can be selected by a snooping protocol through arbitration, or by a directory protocol by maintaining exact presence vector information. Spilling can be implemented in either a "push" or "pull" strategy. In a pull-based implementation, a victim is buffered locally while the evicting cache notifies a randomly chosen host cache to fetch the block. The host cache then pulls the block by issuing a special prefetch request, which migrates the block using a conventional cache coherence protocol. A push-based implementation involves sending the block from the evicting cache to the host cache, which may be split into two data transfers if a directory is involved. To handle race conditions, the host cache needs to acknowledge the sender to finish the spilling process.

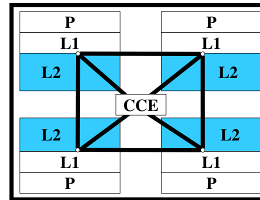

Figure 2. Private Caches with a Central Directory

Challenges remain to support cooperative caching with reduced latency and space overhead. A snooping protocol incurs bus arbitration overhead on every local L2 miss, which may degrade performance for workloads with low local L2 hit ratios. A directory protocol can reduce such overhead, using a directory lookup instead of waiting for responses from all caches, but requires an implementation that only provides state information for on-chip blocks in order to avoid the prohibitive amount of storage needed to provide directory information for all memory blocks.

3.1 Private Caches with a Central Directory

Our proposed implementation of cooperative caching uses a specialized, on-chip, central directory to answer these challenges. This implementation is based on a MOSI directory protocol to maintain cache coherence, but differs from a traditional directory-based system in several ways: (1) the directory memory for private caches is efficiently implemented by duplicating the tag structures (not the exact tag content) of all private caches; (2) the directory is centralized to serve as the serializing point for coherence and provide fast access to all caches; (3) the directory is connected to individual cores using a special point-to-point ordered network, separate from the network connecting peer caches for data transfers. It should be noted that this design can be used for other CMP systems while CMP cooperative caching can also be implemented in various other ways.

Figure 2 illustrates the major on-chip structures for a 4-core CMP. The Central Coherence Engine (CCE) embodies the directory memory and coherence engine, and its proximity to all caches can reduce the network latency. As the number of cores increases for future CMPs, the CCE will become a performance bottleneck. We limit the number of cores managed by a CCE (e.g., 4 or 8), and believe that CCE-managed small-scale clusters of cores can be used as building blocks for large-scale CMPs. However the design and evaluation of such cluster-based, hierarchical CMPs are not covered in this paper.

3.2 Central Coherence Engine (CCE)

The internal structure of the CCE is shown in Figure 3, with its directory memory organization illustrated. Similar to the remote cache shadow directory used in [25], the CCE's directory memory is organized as a duplication of all private caches' tag arrays. Figure 3 shows the directory for an 8-core CMP, each core having 2-way associative L1 I/D split caches and a 4-way associative L2 cache. The tags are indexed in exactly the same way as in an individual core. Each tag entry consists of the tag bits and state bits. In

IEEE COMPUTER SOCIETY

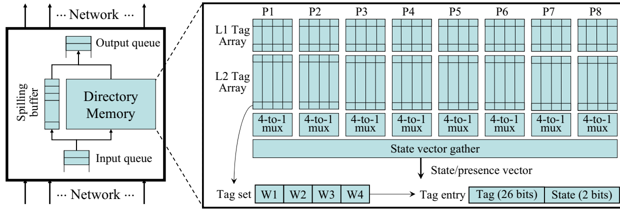

Figure 3. CCE and Directory Memory Structure (8-core CMP with 4-way associative L2 caches)

our implementation, the directory memory is multi-banked using low-order tag bits to provide high throughput.

A directory lookup will be directed to the corresponding bank, to search the related cache sets in all the duplicated tag arrays in parallel. The results from all tag arrays are gathered to form a state/presence vector as in a conventional directory implementation. The latency of a directory lookup is expected to be almost the same as a private L2 cache tag lookup, since the gather step should only marginally increase the latency.

Because cooperative caching requires the L1 caches to be non-inclusive with the local L2 caches, the CCE has to duplicate the tags for all cache levels. The tags are updated by the protocol engine to reflect what blocks are stored in each core's private caches, and what their states are. However, the CCE does not have to know whether a block resides in a processor's L1 or L2 cache because we want to be able to move the block between different cache levels without notifying the CCE. In essence, the CCE is concerned with knowing whether a processor's private (L1 and L2) caches have a block, but not the precise location of the block.

Table 1 lists the storage overhead for an 8-core CMP with a 4-way associative 1MB per-core L2 cache, assuming a system with 4 Terabytes of total physical memory and a 128-byte block size. The total capacity of extra bits in cache blocks, duplicate tag arrays and spilling buffers is 216KB, increasing the on-chip cache capacity by 2.3% (or 4.6% for a 64-byte block size). This ratio is similar to Piranha [5] and lower than CMP-NuRapid [8]. We do not model the area of the separate point-to-point network as it requires the consideration of many physical constraints, which is not the focus of this paper. However, we believe it should be comparable to that of existing CMP's on-chip networks.

Besides maintaining cache coherence, the CCE also needs to support cooperation-related on-chip data transfers and provide singlet information for the private caches. The implementation of these functions is discussed below.

3.2.1 Sharing of Clean Data

To support on-chip cache-to-cache transfers of clean data, the CCE needs to select a clean owner for a miss request. By searching the CCE directory memory, the CCE can simply choose any cache with a clean copy as the owner and forward the request to it, which will in turn forward its clean block to the requester. This implementation requires no extra coherence state or arbitration among the private caches. On the other hand, CCE directory state has to be updated when evicting a clean copy. This requirement is met by extending the baseline cache coherence protocol with a "PUTS" (or PUT-Shared) transaction, which notifies the CCE about the replacement of a clean data block. On receiving such a request, the CCE updates its directory state.

Table 1. Storage Overhead

Component	Location	Size (KB)
Cache tag duplication	Directory	192
Singlet/reuse bits	Caches	16
Spilling buffers	CCE	8
	Total	216

3.2.2 Spilling

In the current implementation, private caches and the CCE cooperate to implement push-based spilling, which consists of two data transfers. The first transfer is a normal write back initiated by the private cache, sending the block to the CCE, which temporarily buffers the data and acknowledges the sender. The second transfer ships the block from the CCE to a randomly chosen host cache. The cache that receives the spilled block will then acknowledge the directory to release the buffer.

3.2.3 Communicating Singlet Information

When the CCE receives a write back message, or a PUTS message which indicates the eviction of a clean block, it checks the presence vector to see if this action leaves only one copy of the block on-chip. If so, an advisory notification will be sent to the cache holding the last copy of the block, which can set the block's singlet bit to 1.

4 Performance Evaluation

We now present an evaluation of cooperative caching for a variety of different scenarios. We compare the effectiveness of cooperative caching to the private and shared cache schemes and also compare against victim replication [35].

4.1 Methodology

We use a Simics-based [21] full-system execution-driven simulator. The cache simulator is a modified version of Ruby from the GEMS toolset [22]. The processor module *ms2sim* is a timing-directed functional simulator that models modern out-of-order superscalar processors using Simics Microarchitecture Interface (MAI). Table 2 lists the relevant configuration parameters used in our simulations.

Table 2. Processor and Cache/Memory Parameters

Component	Parameters
Processor pipeline	4-wide, 10-stages
Window/scheduler	128 / 64 entries
Branch predictors	12K YAGS + 64-entry RAS
Block size	128 bytes
L1 I/D caches	32KB, 2-way, 2-cycle
L2 caches/banks	Sequential tag/data access, 15-cycle
On-chip network	Point-to-point, 5-cycle per-hop
Main Memory	300 cycles total

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Table 5. Multithreaded Workload Miss Rate and L1 Miss Breakdown

	Thousand misses per transaction Off-chip (Private / Shared / CC)	L1 Misses breakdown (Private / Shared / CC)		
		Local L2	Remote L2	Off-chip
OLTP	9.75 / 3.10 / 3.80	90% / 15% / 86%	7% / 84% / 13%	3% / 1% / 1%
Apache	1.60 / 0.90 / 0.94	65% / 9% / 51%	15% / 77% / 36%	20% / 14% / 13%
JBB	0.13 / 0.08 / 0.10	72% / 10% / 57%	14% / 80% / 32%	14% / 10% / 11%
Zeus	0.71 / 0.46 / 0.49	67% / 9% / 45%	15% / 78% / 41%	19% / 12% / 13%

Table 3. Workloads

Multiprogrammed (4-core)	
Name	**Benchmarks**
Mix1	apsi, art, equake, mesa
Mix2	ammp, mesa, swim, vortex
Mix3	apsi, gzip, mcf, mesa
Mix4	ammp, gzip, vortex, wupwise
Rate1	4 copies of twolf, WS larger than 1MB
Rate2	4 copies of art, WS smaller than 1MB
Multithreaded (8-core)	
Name (#trs)	**Setup**
OLTP (400)	IBM DB2 EEE v7.2, 25K warehouses, 128 users
Apache (2500)	20000 files, 3200 clients, 25ms think time
JBB (12000)	Sun HotSpot 1.4.0, 1.5 warehouses per-core
Zeus (2500)	Similar to Apache except being event-driven

Table 4. Network and Cache Configurations

4-core, 4MB total L2 capacity			
	Network	L2 assoc.	Worst-case latency
Private	2X2 mesh	4-way	50-cycle
Shared	2X2 mesh	16-way	40-cycle
8-core, 8MB total L2 capacity			
	Network	L2 assoc.	Worst-case latency
Private	3X3 mesh	4-way	70-cycle
Shared	4X2 mesh	32-way	60-cycle

For benchmarks, we use a mixture of multithreaded commercial workloads and multiprogrammed SPEC2000 workloads. The commercial multithreaded benchmarks include OLTP (TPC-C), Apache (static web content serving using the open source Apache server), JBB (a Java server benchmark) and Zeus (another static web benchmark running the commercial Zeus server) [1]. Multiprogrammed workloads are combinations of heterogeneous and homogeneous SPEC CPU2000 benchmarks. We use the same set of heterogeneous workloads as [8] for their representative behavior, and include two homogeneous workloads with different working set sizes to explore extreme cases. Table 3 provides more information on the workload selection and configuration. The commercial workloads are simulated with an 8-core CMP, while the multiprogrammed workloads use a 4-core CMP, as we believe the scale of CMP systems may be different for servers vs. desktops.

Our default configuration associates each core with a 1MB 4-way associative unified L2 cache. Inclusion is not maintained between L1/L2 caches for cooperative caching (thus the baseline private caches) for the reasons described in section 2.5. For the shared cache scheme, private L1 caches are inclusive with the shared L2 cache to simplify the protocol design. Throughout our evaluation, we compare private L2 caches with a shared L2 cache having the same aggregate associativity and total capacity. We classify L2 hits for a shared cache into local and remote L2 hits, meaning hits into a processor's local and remote L2 banks, respectively. Unless noted otherwise, all caches use LRU for replacement.

Mesh networks are used for intra-chip data transfers, modeling non-uniform hit latencies for the shared cache. We model on-chip network and cache latencies similar to those in [35], but model a 12-FO4 processor cycle as compared to a 24-FO4 cycle delay to reflect a high-frequency design. Cooperative caching and private caches communicate with the CCE using point-to-point links having one-hop latency, adding 10 cycles of extra latency for local L2 misses (including one extra network hop and directory access latencies). Network and CCE contention are both modeled. Table 4 provides more details on network and cache configurations.

4.2 Multithreaded Workload Results

Instead of reporting IPC and misses per instruction, we measure the performance of commercial workloads using a work-related throughput metric [1] and run multiple simulations with random perturbation to compensate for workload variability. The number of transactions simulated for each benchmark is reported in Table 3. Table 5 shows the off-chip miss rates and L1 miss breakdowns and helps to explain the performance advantage of cooperative caching. For each benchmark, we report the off-chip miss rate in terms of thousand misses per transaction (column 2), and break down L1 misses into local and remote L2 hits as well as off-chip accesses (columns 3-5). An ideal caching scheme should have an off-chip miss rate as low as the shared cache, and local L2 hit ratio as high as the private scheme. Cooperative caching has much lower off-chip miss rates (within 4-25% of a shared cache) than the baseline private caches, and 5-6 times higher local L2 hit ratios than the shared cache. These characteristics suggest cooperative caching will likely perform better than both private and shared cache schemes unless off-chip miss rates are very low (favoring private caches) or memory latencies are extremely long (favoring a shared cache).

Figure 4 compares the performance of private, shared and cooperative caching schemes, as transaction throughput normalized to the shared cache. Four cooperative caching schemes are included for each benchmark: from left to right they use system-wide cooperation/spill probabilities of 0%, 30%, 70% and 100% respectively. Currently we set the spill and cooperation probabilities equal to each other.

IEEE COMPUTER SOCIETY

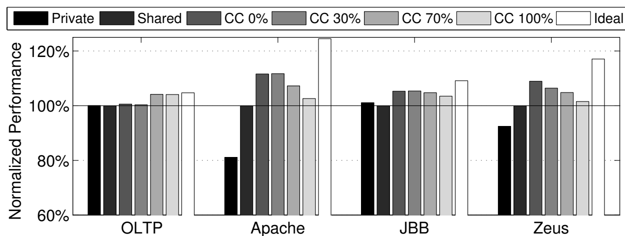

Figure 4. Multithreaded Workload Performance

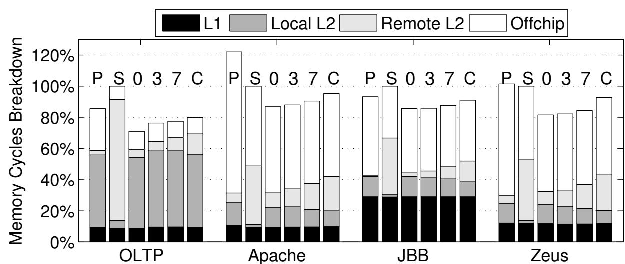

Figure 5. Multithreaded Workload Average Memory Access Latency (from left to right in each group: Private (P), Shared (S), CC 0% (0), CC 30% (3), CC 70% (7) and CC 100% (C))

As discussed in Section 2.4, this probability is used by L2 caches to decide how often to apply replication-aware data replacement and spilling of singlet victims. The default cooperative caching scheme uses a 100% cooperation probability, to optimize capacity by having no cooperation throttling. We choose only four different probabilities as representative points along the sharing spectrum, although cooperative caching can support finer-grained throttling by simply varying the cooperation probability.

For our commercial workloads, the default cooperative caching scheme ("CC 100%") always performs better than the private and shared caches. The best performing cooperation probability varies with the different benchmarks, which boosts the throughput to be 5-11% better than a shared cache and 4-38% better than with private caches. An ideal shared cache scheme is included to model a shared cache with the same uniform access latency as a local bank (15-cycle). Cooperative caching achieves over half of the performance benefit of an ideal shared cache for all benchmarks.

The average memory access latencies (normalized to the shared cache) for different schemes are shown in Figure 5. In each case we break down the access latency into cycles spent in L1 hits, local and remote L2 hits and off-chip accesses. We calculate the average latency by assuming no overlap between memory accesses. Comparing Figures 4 and 5, we see that lower access latency does not necessarily result in better performance. For example, in the case of OLTP, private caches have a relatively lower access latency but effectively the same performance as a shared cache. This is because off-chip accesses have a much smaller contribution to the average latency for shared caches than they do for private caches, whereas the contribution of on-chip accesses is much larger. An out-of-order processor can tolerate some of the on-chip

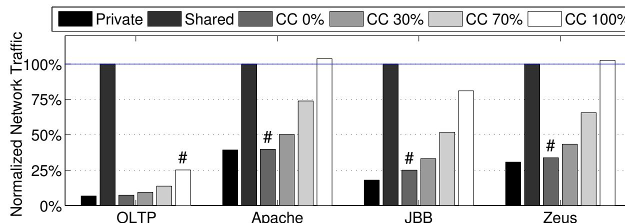

Figure 6. Multithreaded Workload Bandwidth ("#" indicates the best performing CC scheme)

latencies, even to remote L2 cache banks, but can do little for off-chip latencies. For Apache and Zeus, whose off-chip accesses consist of over 50% of the total memory cycles, on-chip caches play an important role in reducing the number of off-chip accesses and thereby improving performance.

Figure 6 presents the relative amount of network traffic (normalized to the bandwidth consumption generated with a shared cache) to accomplish the same amount of work (e.g., an OLTP transaction) for the different caching schemes. Cooperative caching consumes bandwidth to "recirculate" data, but filters traffic by satisfying L1 misses from the local L2 caches. Although unthrottled cooperative caching can sometimes generate more traffic than a shared cache, the best performing cooperative caching schemes' bandwidth requirements are lower by more than 50%.

4.3 Multiprogrammed Workload Results

In this section, we analyze cooperative caching's performance for multiprogrammed workloads. We compare performance using the aggregate IPCs from 1 billion cycles of simulation. No cooperation throttling is used because a single system-wide cooperation probability is not sufficient to accomodate the heterogeneity across benchmarks, and we plan to study adaptation mechanisms for heterogeneous workloads in the future.

Multiprogrammed SPEC2000 workloads differ from commercial multithreaded workloads in several ways: (1) no replication control is needed for private caches as little sharing exists among threads; (2) consequently most L1 cache misses in the private cache scheme are satisfied by the local L2 cache, which often leads to reduced average on-chip cache latency and better performance than a shared cache; (3) the aggregate on-chip cache resources still need to be managed globally to allow dynamic capacity sharing among programs with different working set sizes. We expect cooperative caching to retain the advantages of private caches and reduce the number of off-chip misses via cooperation.

Table 6 lists the off-chip miss rates and L1 miss breakdowns for private, shared and cooperative caching schemes. As with multithreaded workloads, cooperative caching can effectively reduce the amount of both off-chip (shown by the low miss rates in column 2) and cross-chip references (demonstrated by the high local L2 hit ratios in column 3). The off-chip miss rates are only 0-33% higher than a shared cache, and the local L2 hit ratios are close to

IEEE COMPUTER SOCIETY

Table 6. Multiprogrammed Workload Miss Rate and L1 Miss Breakdown

	Misses per thousand instructions Off-chip (Private / Shared / CC)	L1 Misses breakdown (Private / Shared / CC)		
		Local L2	Remote L2	Off-chip
Mix1	3.1 / 2.0 / 2.4	78% / 19% / 67%	3% / 73% / 22%	19% / 9% / 11%
Mix2	3.0 / 1.6 / 1.8	64% / 35% / 75%	4% / 55% / 14%	32% / 9% / 11%
Mix3	1.2 / 0.7 / 0.8	91% / 20% / 87%	1% / 77% / 9%	7% / 3% / 4%
Mix4	0.6 / 0.3 / 0.3	95% / 12% / 90%	0% / 86% / 8%	4% / 2% / 2%
Rate1	0.8 / 0.6 / 0.8	90% / 20% / 80%	3% / 76% / 13%	7% / 4% / 6%
Rate2	53 / 51 / 41	31% / 7% / 24%	11% / 47% / 34%	58% / 46% / 42%

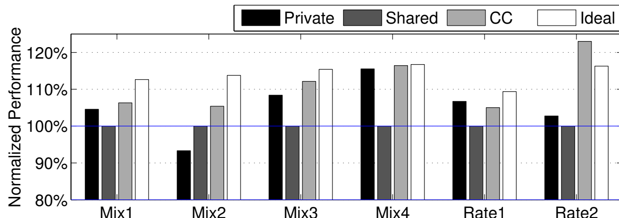

Figure 7. Multiprogrammed Workload Performance

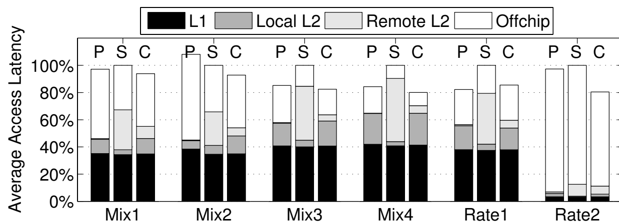

Figure 8. Multiprogrammed Workload Average Memory Access Latency (from left to right in each group: Private, Shared, and cooperative caching)

those using private caches.[3]

Notice the high off-chip miss rates of Rate2 (over 40 misses per thousand instructions) are caused by running four copies of `art`, whose working set exceeds 1MB (which is one processor's fair share of the total L2 capacity). Surprisingly, a shared cache has an off-chip miss rate similar to private caches due to inter-thread contention, which indicates the importance of achieving performance isolation. Cooperative caching has 20% fewer misses because it allows more stable dynamic capacity sharing through replacement-based global cache management (as compared to a shared cache's request-driven management). Further investigation of cooperative caching's performance isolation features will be the topic of another paper.

The aggregate IPCs for the different schemes, normalized to the shared cache scheme, are shown in Figure 7. Cooperative caching outperforms the private scheme by an average of 6% and the shared caches by 10%. For Rate2, cooperative caching performs better than the ideal shared scheme because it has lower miss rate than a shared cache. Figure 8 shows the average memory access latency, assuming no overlap between accesses. It illustrates that private caches run faster than

[3]For Mix2, cooperative caching's local L2 hit ratio is higher than the private scheme because one of the benchmarks (`ammp`) experiences much fewer off-chip misses thus progresses faster and reaches a different computation phase during the simulation.

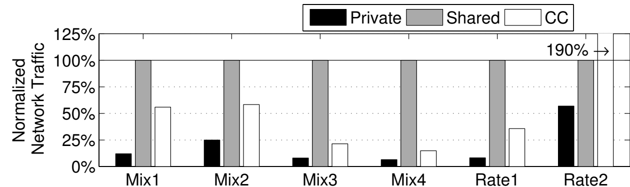

Figure 9. Multiprogrammed Workload Bandwidth

a shared cache for Mix1, Mix3 and Mix4 by reducing cross-chip references, while a shared cache improves Mix2's performance by having many fewer off-chip misses. Cooperative caching combines their strengths and outperforms both for all heterogeneous workloads. For homogeneous workloads, Rate1 consists of four copies of `twolf`, whose working set fits in the 1MB L2 cache, so private caches are the best choice while cooperative caching is slightly behind. Cooperative caching reduces the off-chip miss rate for Rate2, and consequently improves its performance by over 20%.

Figure 9 shows the amount of network traffic per committed instruction, normalized to the shared scheme. Except for Rate2, cooperative caching generates 25-60% of the network traffic of a shared cache. Cooperative caching generates many spills, and thus more cross-chip traffic, for Rate2 because `art`'s large working set causes frequent local L2 evictions. Although `art`'s absolute bandwidth requirement is low, it still indicates the importance of cooperation throttling for multiprogrammed workloads, especially for benchmarks with large working sets and less temporal locality.

4.4 Sensitivity Studies

We first evaluate the performance robustness of cooperative caching with a sensitivity study of commercial workload performance using in-order, blocking processors. We choose to use in-order processors mainly to reduce simulation time, but they also represent a relevant design choice [19]. The main idea here is to assess the benefit of cooperative caching across a spectrum of memory hierarchy parameters.

Figure 10 compares the relative performance (transaction throughput normalized to the shared cache scheme) of private, shared and cooperative caching for different system sizes (4-core vs. 8-core), per-core cache capacities (512KB, 1MB, and 2MB) and memory latencies (300 cycles vs. 600 cycles). It is clear that the shared cache performs better than private caches when per-core cache capacity is small, while the performance gap closes

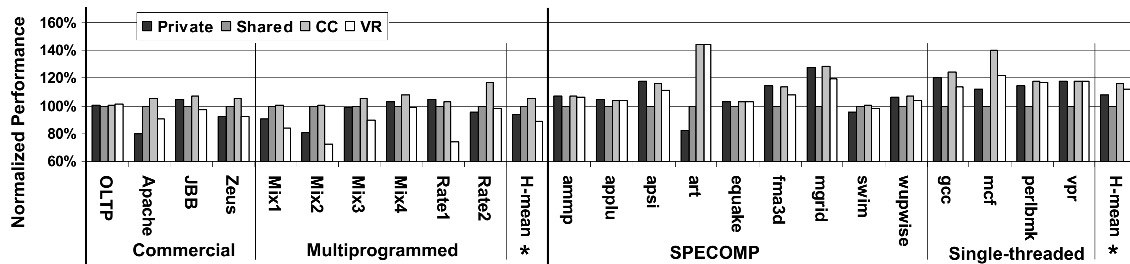

Figure 11. Comparison with Victim Replication

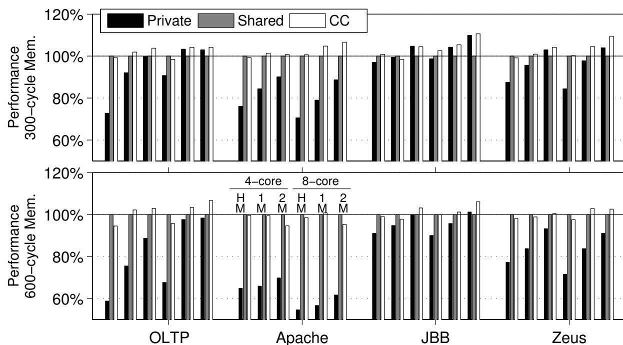

Figure 10. Performance Sensitivity (300 and 600 cycles memory latencies; from left to right for each group: 4-core and 8-core CMPs with 512KB, 1MB and 2MB per-core caches)

Table 7. Speedups with Varied CCE Latencies

Extra cycles	0	+5	+10	+15
Multithreaded	7.5%	4.7%	3.2%	0.1%
Multiprogrammed	11%	8.0%	7.0%	5.9%

when per-core capacity increases. However cooperative caching's performance is more dependent on the aggregate on-chip cache capacity. As the aggregate cache size increases, cooperative caching is increasingly better than a shared cache, while its advantage over the private cache scheme gradually decreases. Overall, cooperative caching achieves the best performance for most configurations with a 300-cycle memory latency. When memory latency doubles, the latency of off-chip access dominates, and cooperative caching becomes similar to a shared cache, having -2.2% to 2.5% average speedups.

We then study cooperative caching's performance sensitivity to the CCE latency. Table 7 shows the speedups of cooperative caching over the baseline shared cache design when the CCE latency is further increased. Speedups over private caches are not included because both cooperative caching and private caches are implemented using the CCE, and increasing CCE latency has a similar effect on both designs. We can observe that cooperative caching continues to perform better than a shared cache with 10 extra cycles of CCE latency.

4.5 Comparison With Victim Replication

We choose to compare cooperative caching with victim replication [35] but not other recent CMP caching schemes [6, 7, 17] for several reasons: (1) both victim replication and cooperative caching use cache replacement as the underlying technique to control replication; (2) both schemes are based on a traditional cache organization, while other proposals require significant changes in the cache organization; (3) they both use a directory-based protocol implementation, rather than requiring different styles of coherence protocols (e.g., snooping protocol or token coherence).

The comparison is conducted using in-order processors with the same L1/L2 cache sizes and on-chip latency parameters as [35] to specifically match its evaluation. We also tried using the set of parameters as in Section 4.1, however, victim replication is worse than both shared and private schemes for 3 out of 4 commercial benchmarks. As in [35], we use random replacement for victim replication, because it's not straightforward to set the LRU information for the replica.

To create as realistic a match to the previous paper as possible, we also include results for 9 SPECOMP benchmarks [3] and 4 single-threaded SPEC2000 benchmarks with the MinneSPEC reduced input set [27], all running on 8-core CMPs. The single-threaded benchmarks are in common with [35], and the SPECOMP benchmarks have characteristics similar to the multithreaded workloads it used. We report the performance measurement (either transaction throughput or IPC, normalized to the shared scheme), instead of the average memory access latency used by [35].

Figure 11 compares the performance of private, shared, cooperative caching and victim replication schemes for all four classes of workloads: commercial multithreaded workloads with large working sets and substantial inter-thread sharing, SPEC2000 multiprogrammed workloads, SPECOMP multithreaded workloads with little sharing and single-threaded workloads. Victim replication outperforms both private and shared schemes for SPECOMP and single-threaded workloads, on average by 2% and 12%. However, victim replication performs poorly for commercial and multiprogrammed workloads, being on average 6% slower than private caches and 11% slower than a shared cache.

Cooperative caching consistently performs better than victim replication (except for OLTP). Cooperative caching provides the best performance for 3 out of 4 commercial workloads, 5 out of 6 multiprogrammed workloads and all single-threaded workloads; it is less than 1.4% slower than the best schemes for all SPECOMP benchmarks. Across all of these benchmarks, cooperative caching is on average 8% better than private caches, 11% better than a shared cache, and 9.2% better than victim replication.

IEEE COMPUTER SOCIETY

5 Related Work

CMP cooperative caching is inspired by software cooperative caching schemes [9–11]. The same key issues and observations of file/web caching can be applied to improve on-chip CMP cache organization, but different mechanisms are needed to support cooperation at the level of processor caches. In balancing space allocation between local vs. global data, cooperative caching is similar to variable allocation, global replacement based virtual memory management. In the hardware domain, cooperative caching is related to COMA architectures [12, 30] in that they both try to reduce the impact of long-latency remote accesses by attracting data to their consumers' local storage. They also face similar issues such as directory implementation, replacement policy and replication control, although different tradeoffs and solutions are needed for DSM vs. CMP architectures. For CMP caching designs, two lines of research are most relevant to our proposal: (1) proposals to optimize both the latency and the capacity of CMP caches; and (2) coherence protocol optimizations to reduce complexity and improve performance.

Optimizing Latency and Capacity

Besides the many CMP caching studies [18, 20, 26, 31], several hybrid caching schemes [6, 8, 13, 15, 34, 35] are closely related to our proposal of optimizing both CMP cache capacity and access latency. Beckmann and Wood [6] study the performance of CMP-NUCA schemes and the benefit of using LC transmission lines to reduce wire delay. Cooperative caching is orthogonal to such new technologies and can potentially use fast wires to speed up on-chip accesses. Cooperative caching is similar to victim replication [35] in that replication is controlled through the cache replacement policy. Victim replication is simpler by identifying replications via static address mapping. Cooperative caching differs by using private caches for both dynamic sharing and performance isolation.

CMP-NuRapid [8], like CMP-NUCA, also optimizes for special sharing patterns using distributed algorithms (i.e., controlled replication for read-only sharing, *in-situ* communication for read-write sharing, and capacity stealing for non-sharing). In contrast, cooperative caching aims to achieve these optimizations through a unified technique: cooperative cache placement/replacement, which can be implemented in either a centralized or distributed manner. CMP-NuRapid implements a distributed directory/routing service by maintaining forward and reverse pointers between the private tag arrays and the shared data arrays. This implementation requires extra tag entries that may limit its scalability, and increases the complexity of the coherence protocol (e.g., the protocol has to avoid creating dangling pointers). Cooperative caching tries to avoid such issues by using a simple, central directory engine with less space requirements.

Based on a non-uniform cache architecture (NUCA) [17], Huh et al. [15] design a CMP cache to support a spectrum of sharing degrees, denoting the number of processors sharing a pool of their local L2 banks. The average access latency can be optimized by partitioning the aggregate on-chip cache into disjoint pools, to fit the running application's capacity requirement and sharing patterns. Static mappings with a sharing degree of 2 or 4 are found to work the best. Dynamic mapping can improve performance at the cost of complexity and power consumption. Cooperative caching is similar in trying to support a spectrum of dynamic sharing points, but achieves it through cooperation among private caches with dynamic throttling.

Similarly, synergistic caching [13] groups neighboring cores and their private L1 caches into clusters to allow fast access of shared data among neighbours. It offers three duplication modes (i.e., beg, borrow and steal) to balance between latency and capacity and shows that no single duplication mode performs the best across all benchmarks.

Lastly, Yeh and Reinman [34] propose a distributed NUCA cache consisted of ring-connected private caches to improve both throughput and fairness for multiprogrammed workloads. Cooperative caching can benefit from their approach to achieve quality of service.

Coherence Protocol Optimizations

Multiprocessor researchers have studied various coherence protocol optimizations for decades. To save space, only techniques that are closely related to cooperative caching are discussed. Although the concept of CMP cooperative caching is orthogonal to the implementation of the coherence protocol, techniques to improve both protocol simplicity and performance are related to the proposed mechanisms. Token coherence [23] and its application to multiple-CMP systems [24] simplifies the protocol design by decoupling the protocol's correctness substrate and performance policy. The CCE utilizes a fast central directory, similar to the central coherence unit proposed by Takahashi et al. [31] and a directory memory structure resembling [25]'s remote cache shadow directory.

6 Conclusion

We have presented cooperative caching as a unified approach to manage a CMP's aggregate on-chip cache resources. Cooperative caching answers the challenges of CMP cache designs in two ways: it combines the strengths of both private and shared caches to reduce the number of expensive cross-chip and off-chip accesses, and it supports a spectrum of capacity sharing points to suit the requirements of specific workloads. Our simulation shows that using cooperative caching achieves the best performance for different CMP configurations and workloads. Cooperative caching can reduce the runtime of simulated workloads by 4-38%, and performs at worst 2.2% slower than the best of private and shared caches in extreme cases. Cooperative caching also outperforms the victim replication scheme by 9% on average over a mixture of multithreaded, single-threaded and multiprogrammed workloads, while the performance advantage increases for workloads with large working sets. Using cooperative private caches also allows other applications such as power optimization and simpler implementation of performance isolation, which are left as future work.

0-7695-2608-X/06 $20.00 © 2006 IEEE

Acknowledgements

This research is supported in part by NSF grants EIA-0071924 and CCR-0311572, and donations from Intel Corporation.

References

[1] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] J. K. Archibald. A cache coherence approach for large multiprocessor systems. In *the 2nd ICS*, pages 337–345, 1988.

[3] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPECOMP: A new benchmark suite for measuring parallel computer performance. In *the International Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[4] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *the 15th ISCA*, pages 73–80, 1988.

[5] L. A. 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.

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

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *the 36th MICRO*, pages 55–66, Dec 2003.

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication and capacity allocation in CMPs. In *the 32th ISCA*, pages 357–368, June 2005.

[9] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *the 1st OSDI*, pages 267–280, Nov 1994.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE Transactions on Networking*, 8(3):281–293, 2000.

[11] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *the 15th SOSP*, pages 201–212, Dec 1995.

[12] E. Hagersten, A. Landin, and S. Haridi. DDM: A cache-only memory architecture. *IEEE Computer*, 25(9):44–54, 1992.

[13] S. Harris. *Synergistic Caching in Single-Chip Multiprocessors*. PhD thesis, Stanford University, 2005.

[14] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future CMPs. In *the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, Sep 2001.

[15] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *the 19th ICS*, pages 31–40, June 2005.

[16] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *the 18th ICS*, pages 257–266, June 2004.

[17] 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.

[18] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *the 13th International Conference on Parallel Architecture and Compilation Techniques*, 2004.

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

[20] 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.

[21] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[22] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 2005.

[23] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *the 30th ISCA*, pages 182–193, June 2003.

[24] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving multiple-CMP systems using token coherence. In *HPCA-11*, pages 328–339, Feb 2005.

[25] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. High-throughput coherence control and hardware messaging in Everest. *IBM Journal of Research and Development*, 45(2), 2001.

[26] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *the 23rd ISCA*, pages 67–77, May 1996.

[27] A. K. Osowski and D. J. Lilja. MinneSPEC: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, June 2002.

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

[29] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way cache: Demand based associativity via global replacement. In *the 32nd ISCA*, pages 544–555, June 2005.

[30] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *HPCA 1*, pages 276–285, Jan, 1995.

[31] M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki. A shared-bus control mechanism and a cache coherence protocol for a high-performance on-chip multiprocessor. In *HPCA 2*, pages 314–322, Feb 1996.

[32] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. IBM Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[33] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII*, pages 181–192, Oct, 1998.

[34] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *CASES'05*, pages 237–248, Sep 2005.

[35] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled CMPs. In *the 32th ISCA*, pages 336–345, June 2005.

IEEE COMPUTER SOCIETY