COARSE-GRAIN COHERENCE TRACKING

by

Jason F. Cantin

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

2006

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

A dissertation entitled

Course-Grain Coherence Tracking

submitted to the Graduate School of the
University of Wisconsin-Madison
in partial fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Jason Frederick Cantin

Date of Final Oral Examination:

May 2, 2006

Month & Year Degree to be awarded: **December** **May** **August** 2006

****

Approval Signatures of Dissertation Committee

Signature, Dean of Graduate School

Martin Cadwallader /EH

Committee's Page. This page is not to be hand-written except for the signatures

To my wife Candy

COARSE-GRAIN COHERENCE TRACKING

Jason F. Cantin

Under the supervision of

Associate Professor Mikko H. Lipasti and Professor James E. Smith

At the University of Wisconsin-Madison

To maintain coherence in conventional shared-memory multiprocessor systems, processors first check other processors' caches before obtaining data from memory. This coherence checking consumes considerable interconnect bandwidth in broadcast-based systems, and, as a byproduct, increases access latency for nonshared data. Furthermore, it consumes substantial amounts of power, both in the system interconnect and cache tag arrays. Simulation results for a set of commercial, scientific, and multiprogrammed workloads running on a four-processor system show an average of 71% (and up to 94%) of broadcasts are unnecessary, and on average 89% of snoop-induced cache tag lookups miss in the L2 cache.

This dissertation proposes Coarse-Grain Coherence Tracking (CGCT), a new technique that supplements a conventional coherence mechanism and optimizes the performance of conventional coherence enforcement. CGCT monitors the coherence status of large regions of

memory and uses the status information to avoid unnecessary broadcasts and filter unnecessary snoop-induced cache tag lookups. Simulation results show CGCT can eliminate 47-64% of the broadcasts, filter 71-87% of the snoop-induced cache tag lookups, and reduce average execution time 7.3-10.9%. Moreover, CGCT does not affect system compatibility, does not violate cache coherence, and does not violate memory consistency.

In addition to optimizing coherence enforcement, CGCT can enable new optimizations that further improve performance and power-efficiency. In this dissertation I will show that CGCT can enable processors to prefetch data in a safe, efficient, and timely manner and without disturbing other processors. I will also show that CGCT can be used to implement power-efficient DRAM speculation in the memory controllers, detecting regions shared by other processors, and only fetching lines from DRAM if they are not likely to be sourced from another processor's cache.

Acknowledgements

Most of all I would like to thank my wife, Candy, for her continued support, encouragement, patience, proofreading, and invaluable graphing and spreadsheet management skills. Before meeting her I was in a difficult period; I had been unable to make any significant progress on my research for two years, and a class was the only reason to get out of bed in the morning. She fed me homemade food, took me dancing, listened to my frustrations, and brought me a printer when I had a deadline and nothing seemed to work. After meeting her I was able to put together a master's thesis. Two weeks after she agreed to marry me I completed my preliminary examination. We were married 14 months later, four days after my final defense.

This work would not have been possible if not for my parents, David and Brenda Cantin. They have always put my education first. My father introduced me to electronics at a young age, having previously worked to repair radios for helicopters at a military base. My mother tolerated my tinkering; though at times she worried I would burn the house down. But, that was only because once I accidentally ignited a rocket engine in the basement and filled the house with smoke (just once).

This thesis benefited from the numerous questions, comments, and suggestions of my committee members, Professors Mark Hill, Michael Schulte, and Parameswaran Ramanathan. I am especially grateful to my two advisors, Professor James E. Smith and Mikko Lipasti, for their help and guidance during the course of this research. In addition to guiding my research, they have also served as positive role models in my professional development.

I have benefited immeasurably from my experience interning at IBM. Working with Steven Kunkel, Aaron Sawdey, William Starke, Steven Fields, and others at IBM was invigorating, and led to the filing of several patents. Stephen Stevens, an exemplary manager, worked hard on my behalf and was an extraordinary help. This experience influenced me to pursue a different line of research, optimizing coherence enforcement, which resulted in this dissertation.

The University of Wisconsin has a large computer architecture program with many students, and the opportunity for interaction and collaboration was part of my reason for coming here. I enjoyed working and discussing ideas with the many students in our lab, including Nidhi Aggarwal, Wooseok Chang, Ashutosh Dhodapkar, Timothy Heil, Shiliang Hu, Ho-Seop Kim, Marty Licht, Kyle Nesbit, and S. Subramanya Sastry. I also had the opportunity to interact with bright students from other groups, including Brian Fields, Ravi Rajwar, Trey Cain, and Kevin Lepak, and many others.

Sincere thanks to the faculty and staff at the University of Cincinnati, where I was an undergraduate student in Electrical Engineering. My years there were among the most memorable. Early on, Phil A. Wilsey, through his maniacally paced Computer Organization and Architecture class, changed my life by showing how to combine logic gates in simple ways to produce powerful computers. From that point on, I would design computers. Later, I teamed up with Fred Beyette to find ways to combine logic with integrated optics, leading to research projects still underway today.

My experience at Digital Equipment Corporation was also key to my choosing to study computer architecture. There was an excitement, energy, and a belief in what we were doing that I have not witnessed anywhere else. It started when Paul Gronowski interviewed me for a co-op

position, and offered me a rare position as a circuit designer. I was paired with Gary Moyer, from whom I learned just about everything there is to know about latches and clocking. While there, I was fortunate enough to learn from such brilliant people as Andrew Lentvorski, Steve Atkins, John Kowaleski, Dan Bailey, Mike Smith, Matt Reilly, and Joel Emer.

This research was financially supported by NSF Grant CCR-083126, CCR-0133437, and CCF-0429854; and graduate research fellowships from the University of Wisconsin Foundation, NSF, and IBM. We greatly appreciate their generous contributions.

Table of Contents

Table of Figures

List of Tables

1. Introduction and Motivation

Cache-coherent multiprocessor systems are widely used computing platforms, with applications ranging from commercial transaction processing and database services to large-scale scientific computing. They have become a critical component of internet-based services in general. As system architectures have advanced to incorporate larger numbers of faster processors, the memory system has become critical to overall system performance and scalability. Improving bandwidth, reducing latency, and reducing power consumption in the memory system have become key design issues.

To maintain coherence and exploit fast cache-to-cache transfers, multiprocessors commonly broadcast memory requests to all the other processors in the system [1, 2, 3, 4]. While broadcasting is a quick and simple way to find cached copies of data, locate the appropriate memory controllers, and order memory requests, it consumes considerable interconnect bandwidth and, as a byproduct, increases latency for non-shared data.

1.1 Coarse-Grain Coherence Tracking

To reduce the bottleneck caused by broadcasting, high performance multiprocessor systems decouple the coherence mechanism from the data transfer mechanism, allowing data to be moved directly from a memory controller to a processor over a separate network [1, 2, 3] or separate virtual channels [4]. This approach to dividing data transfer from coherence enforcement has significant performance potential because the broadcast bottleneck can be alleviated. Many memory requests simply do not need to be broadcast to the entire system, either because the data

is not currently shared, the request writes modified data back to memory, the request is an instruction fetch and the instructions are not currently being modified, or the request is for non-cacheable I/O data.

In this dissertation, I leverage the decoupling of the coherence and data transfer mechanisms by developing *Coarse-Grain Coherence Tracking (CGCT)*, a new technique that allows a processor to increase substantially the number of requests that can be sent directly to memory without a broadcast and without violating coherence. CGCT can be implemented in an otherwise conventional multiprocessor system. A conventional cache coherence protocol (e.g., write-invalidate MOESI) is employed to maintain coherence over the processors' caches. However, unlike a conventional system, each processor maintains a second structure for monitoring coherence status at a granularity larger than a single cache line (Figure 1.1). This structure is called the *region coherence array* (RCA) and maintains coarse-grain coherence state over large, aligned memory *regions*, where a region encompasses a power-of-two number of conventional cache lines.

On snoop requests, each processor's RCA is snooped along with the cache line state, and the region's coarse-grain state is piggybacked onto the conventional snoop response. The requesting processor stores this information in its RCA to avoid broadcasting subsequent requests for lines in the same region. As long as no other processors are caching data in that region, requests can go directly to memory and do not require a broadcast.

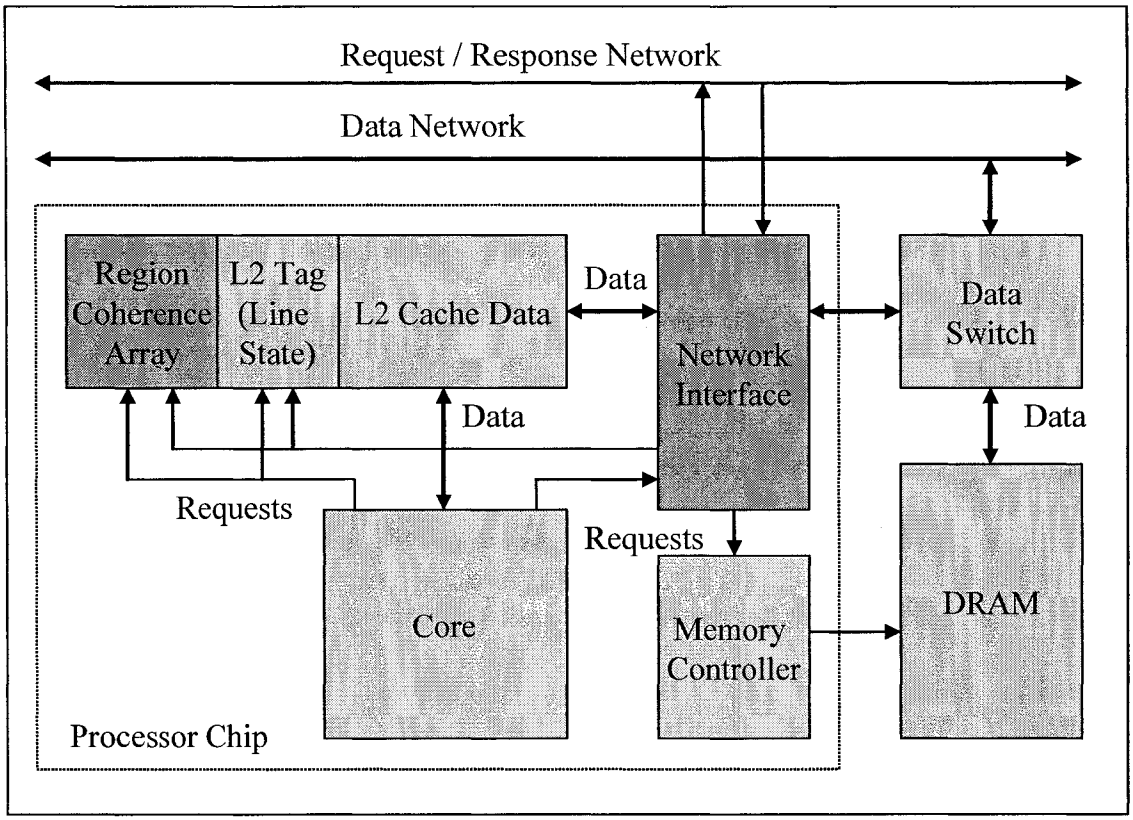

Figure 1.1: Processor modified to implement Coarse-Grain Coherence Tracking. A Region Coherence Array is added, and the network interface may need modification to send requests directly to the memory controller.

As an example, consider a shared-memory multiprocessor system with two-levels of cache and an RCA in each processor. One of the processors, *processor A*, performs a load operation to address *X*. The load misses in the L1 cache, and a read request for *X* is sent to the L2 cache. At the same time, the RCA is checked for the corresponding region *Rx*. The L2 cache coherence state and the region coherence state are read in parallel to determine the status of the line. There is a miss in the L2 cache and the region state is *invalid*, so a data read request is broadcast. All the other processors snoop the request; check their cache for address *X* and their RCA for region *Rx*, and send back a snoop response to *processor A* with the external status of the line and the

region. If no other processor is caching data from *Rx*, an entry for *Rx* is allocated in *processor A*'s RCA with an exclusive state. Until another processor makes a request for a cache line in *Rx*, *processor A* can access any memory location in the region without a broadcast.

Figures 1.2 and 1.3 illustrate the potential of CGCT to reduce broadcast traffic. Figure 1.2 shows the percentage of unnecessary broadcast requests for a set of commercial, scientific, and multiprogrammed workloads on a simulated four-processor PowerPC system (refer to Chapter 3 for information on the simulated system and workloads). On average, 71% of the requests can be handled without a broadcast if the processor has oracle knowledge of the coherence state of data in other caches in the system. The largest contribution is from reads and writes (including prefetches) for data that is not shared at the time of the request. The next most significant contributor is write-backs, which generally do not need to be seen by other processors. These are followed by instruction fetches, for which the data is usually clean-shared. The smallest contributor, although still significant, is Data Cache Block (DCB) operations that invalidate, flush, or zero-out cached copies in the system. Most of these are Data Cache Block Zero (DCBZ) operations used by the AIX operating system to initialize physical pages. Figure 1.3 shows the percentage of broadcasts that remain unnecessary when coherence status is tracked at a coarse-granularity for regions ranging from 128-bytes to 4KB. Most broadcasts are unnecessary, and there is significant potential to detect these unnecessary broadcasts by tracking coherence permissions at a coarse granularity.

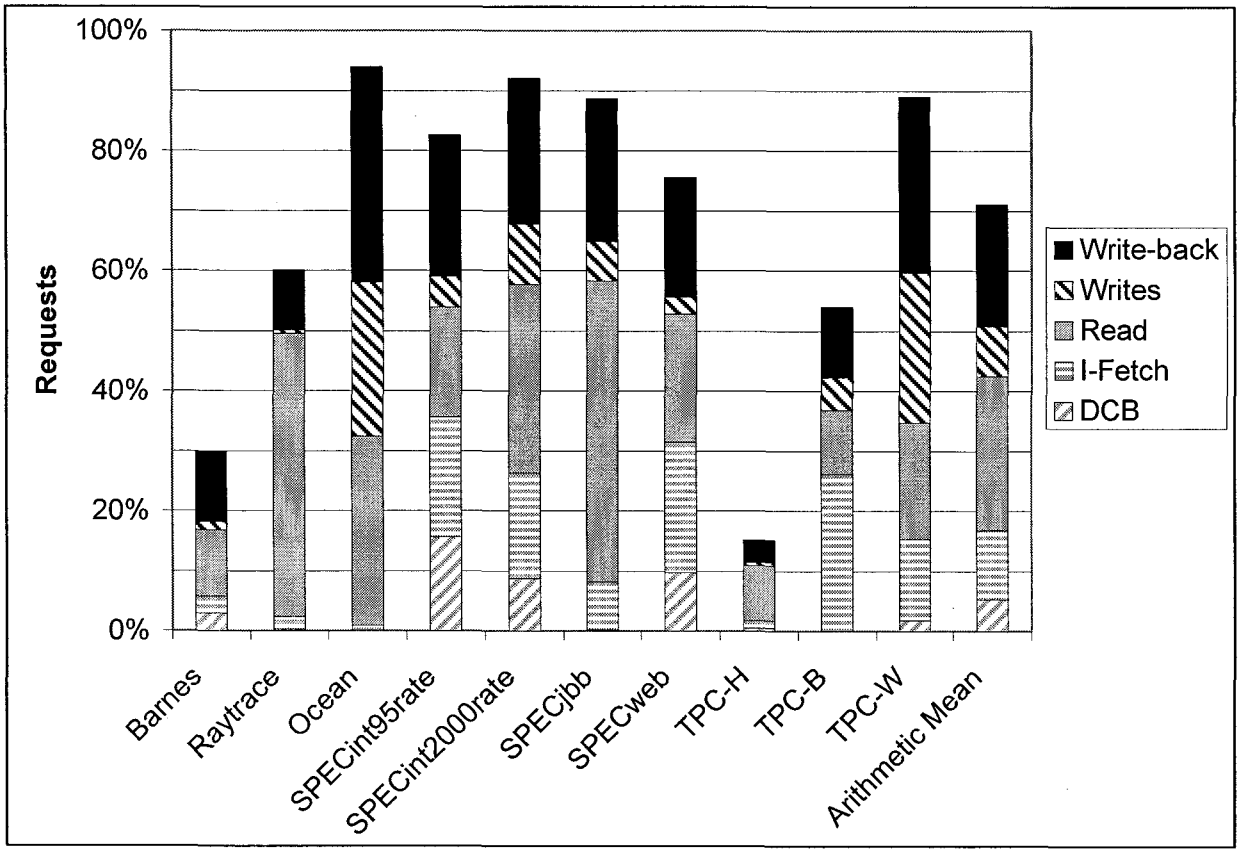

Figure 1.2: Unnecessary broadcasts in a four-processor system.

From 15% to 94% of requests can theoretically be handled without a broadcast. The arithmetic mean across the different workload categories is 71%.

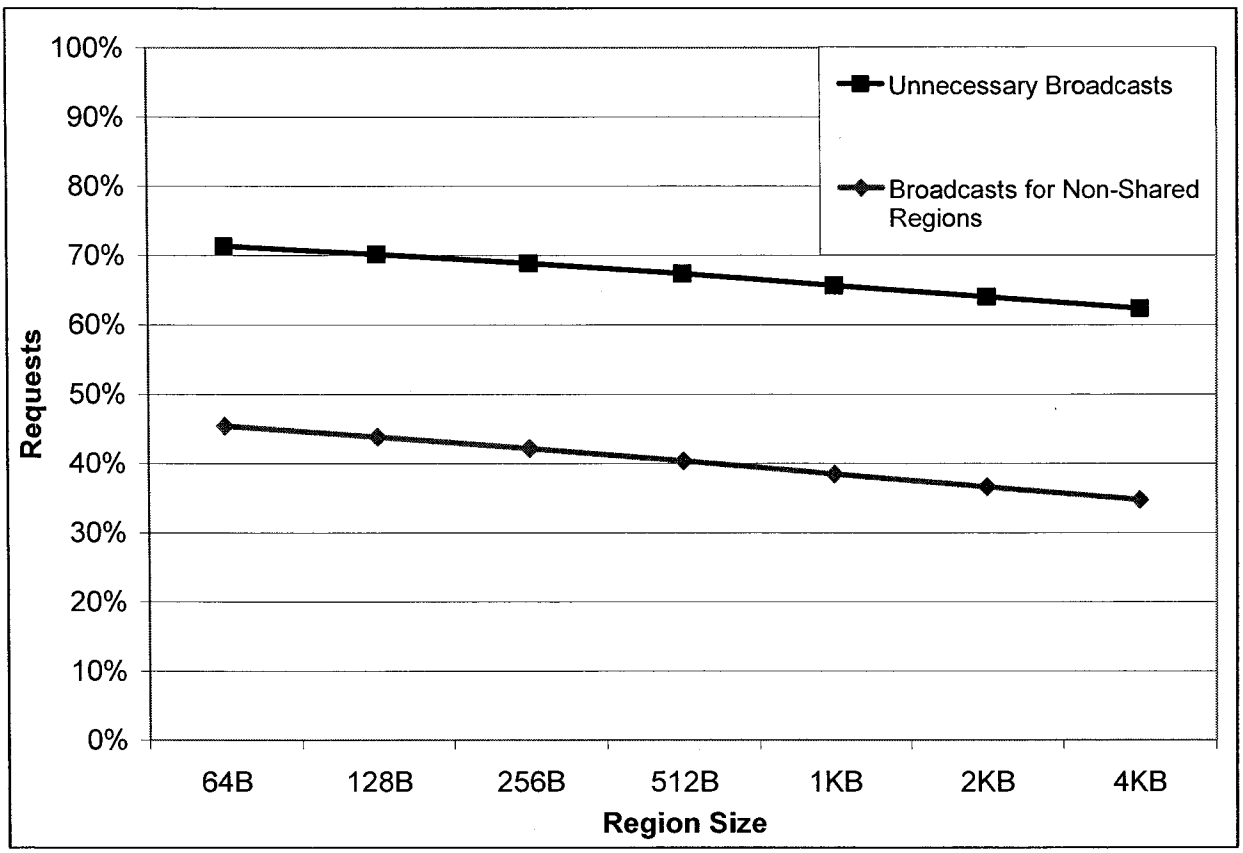

Figure 1.3: Unnecessary broadcasts in a four-processor system tracking coherence status at a coarse-granularity.

This graph shows the percentage of requests for which a broadcast is unnecessary (for varying region sizes), and the percentage for which a broadcast is unnecessary because the data is not shared by other processors. Here, a broadcast is deemed unnecessary if not only the requested line, but an aligned region of memory around that line can be accessed without a broadcast. The cache line size is 64 bytes.

If a significant number of the unnecessary broadcasts can be eliminated in practice, there will be large reductions in traffic over the broadcast interconnect mechanism. This will reduce overall bandwidth requirements, queuing delays, and cache tag lookups. Memory latency will be reduced because data requests will be sent directly to memory, without first going to an

arbitration point and broadcasting to all processors. Some requests that do not require a data transfer, such as requests to upgrade a shared copy to a modifiable state and DCB operations, can be completed immediately without an external request.

Figure 1.4 illustrates the potential of CGCT to filter snoop-induced cache tag lookups. It shows the percentage of external requests that do not require a cache tag lookup for the same workloads and system described above. On average, 87% of external requests do not need to check the cache, either because the requested line is not present or the request is an instruction fetch and the instructions have not been modified. Of these, 70% are from broadcasts that do not need to be performed and may be eliminated with *a priori* knowledge of the status of lines in other processor's caches. For region sizes ranging from 128-bytes to 4Kbytes (2 to 64 lines), the figure shows the percentage of external requests that could be filtered with a perfect implementation of CGCT (i.e., if no lines from the region are cached, or the request is an instruction fetch and no lines have been modified, a cache tag lookup is not performed). Also shown for varying region sizes is the percentage of snoop-induced cache tag lookups that result from a broadcast that could be avoided with a perfect implementation of CGCT (i.e., the unnecessary broadcasts from Figure 1.3) . Most snoop-induced cache tag lookups are unnecessary, and most of these result from unnecessary broadcasts. However, even if all the unnecessary broadcasts are removed, there is still potential to filter 8-20% more snoop-induced cache tag lookups.

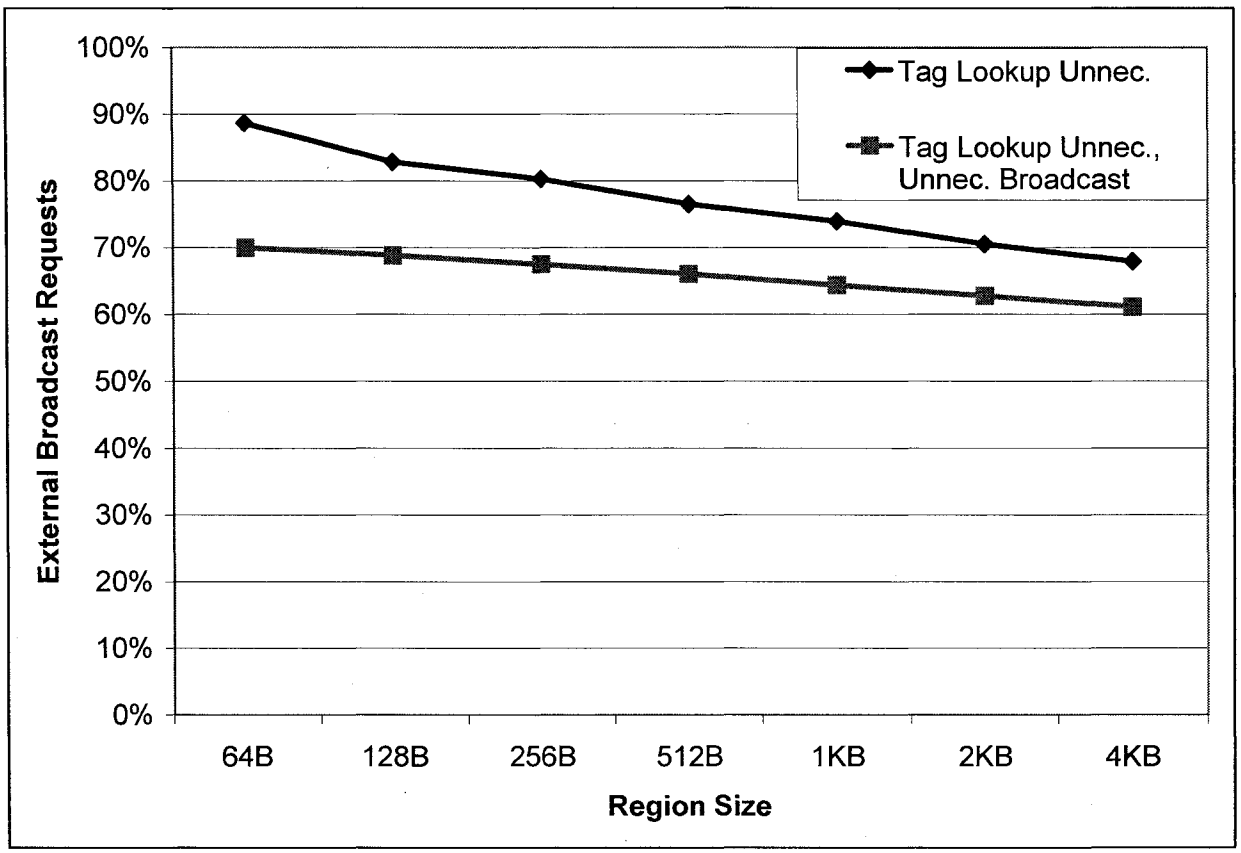

Figure 1.4: Unnecessary snoop-induced cache tag lookups in a four-processor system.

A cache tag lookup is deemed unnecessary if the line is invalid, the request is an instruction fetch and the line is not modified, or the request is a write-back. This graph shows the percentage of external requests that do not need to check the cache for the line, and for varying region sizes (X-axis) the percentage of external requests that could access any line in an aligned region of data surrounding the requested line without checking the cache.

In this dissertation I will show that Coarse-Grain Coherence Tracking, eliminates most of the unnecessary broadcasts and provides the benefits just described. It does this by exploiting spatial locality beyond the cache line size and temporal locality beyond the cache.

1.2 Optimizations Enabled by CGCT

In addition to optimizing coherence enforcement, CGCT enables new optimizations that rely on *a priori* knowledge of the coherence status of lines. In this dissertation I propose and evaluate two such optimizations: Stealth Prefetching (SP) and Power-Efficient DRAM Speculation (PEDS). Other optimizations are possible, but are left for future work.

Stealth Prefetching is a new form of Region Prefetching [5, 6, 7] that is targeted at shared memory multiprocessor systems. It uses CGCT to detect regions of memory that are not shared by other processors and prefetches the lines in those regions. After a threshold number of L2 misses to a region, the rest of the lines in the region are prefetched efficiently from DRAM and transferred to a small buffer close to the processor. The lines are kept there until accessed by the processor, invalidated by other processors' requests, or evicted. What makes Stealth Prefetching "stealthy" is that it does not broadcast prefetch requests, does not interfere with other processors sharing data, and does not prevent other processors from obtaining exclusive copies of lines.

Power-Efficient DRAM Speculation (PEDS) is a new optimization targeted at systems that begin the DRAM access part-way through the snoop, when the memory controller receives the broadcast request. It takes advantage of the CGCT mechanism to identify requests that are likely to be satisfied by data from other processor's caches and uses this information to avoid fetching lines from DRAM unnecessarily to save power.

1.3 Contributions

This section outlines the key contributions of this dissertation.

- **Proposes a new technique for optimizing coherence enforcement.** I propose Coarse-Grain Coherence Tracking, a new technique that supplements a conventional coherence protocol, and optimizes coherence enforcement. CGCT decouples the acquisition of coherence permissions from the request, transfer, and caching of data; tracking coherence status for large regions of memory to optimize subsequent requests.
- **Develops and evaluates an implementation of Coarse-Grain Coherence Tracking.** I present Region Coherence Arrays, and characterize their performance with execution-driven simulation of commercial, scientific, and multiprogrammed workloads.
- **Evaluates and compares competing CGCT implementations.** I evaluate RegionScout Filters, a concurrently proposed implementation of CGCT, and compare them qualitatively and quantitatively to Region Coherence Arrays.
- **Proposes and evaluates new optimizations enabled by CGCT.** I propose a set of new optimizations, including Stealth Prefetching, and Power-Efficient DRAM Speculation. Stealth Prefetching uses CGCT information to identify non-shared regions of data that can be prefetched safely, aggressively, and efficiently. Power-Efficient DRAM speculation uses CGCT information to identify regions of data that are shared, and for which data is likely to be sourced from another cache, to avoid fetching data from DRAM unnecessarily to save power.

1.4 Dissertation Outline

This dissertation is divided into three main parts. The first part introduces, motivates, and describes Coarse-Grain Coherence Tracking, beginning with this chapter (Chapter 1). This is

followed by a survey of related work (Chapter 2), and a description of the simulation methodology and workloads (Chapter 3). The proposed implementation of CGCT, Region Coherence Arrays, is presented and evaluated next (Chapter 4).

The second part of this dissertation describes and evaluates a different implementation of CGCT concurrently proposed by others, namely, RegionScout Filters (Chapter 5). I compare RegionScout Filters qualitatively and quantitatively to RCAs, using execution-driven simulation results for a range of structure sizes and region sizes.

Finally, the third part of this dissertation develops and evaluates two new optimizations enabled by CGCT. First, Stealth Prefetching is described and evaluated (Chapter 6). Next, the dissertation describes and evaluates Power-Efficient DRAM Speculation (Chapter 7). The dissertation ends with a discussion of avenues for future work (Chapter 8) and conclusions (Chapter 9).

2. Related Work

This chapter surveys work related to that presented in this dissertation. I start with a discussion of early work on finding the optimal cache line size to trade off locality, overhead, and bandwidth, as well as using caches with adjustable line sizes (Section 2.1). I then discuss sectoring and sublining caches, a set of techniques that decouple the granularity at which coherence is maintained from the granularity at which data is cached (Section 2.2). Next, and most related to CGCT, are hardware and software methods for avoiding unnecessary broadcasts and snoop-induced cache tag lookups, including RegionScout and directory protocols (Section 2.3). Section 2.4 discusses prefetching work related to the Stealth Prefetching technique proposed in this dissertation. Section 2.5 surveys work related to Power-Efficient DRAM speculation, also proposed in this dissertation. This is followed by a recently proposed technique that uses a structure similar to a Region Coherence Array to accelerate store misses, and is a potential application of CGCT (Section 2.6). Finally, Section 2.7 discusses a technique using coarse-grain information to optimize caching policies, another potential application of CGCT.

2.1 Cache Line Size Studies and Optimizations

There have been a number of studies that analyze the effect of cache line size on system performance, both for single-processor [8, 9, 10, 11, 12] and multiprocessor systems [13, 14]. For systems with a single processor, the tradeoff is between spatial locality, tag storage overhead, internal fragmentation, and data bandwidth or transfer latency. A large cache line exploits spatial locality and better amortizes tag storage, however it can also increase internal

fragmentation and reduce the effective capacity of the cache. Furthermore, a large line takes more time to transfer and can waste data bandwidth by fetching unneeded data from memory. In multiprocessor systems there is data sharing to consider: too large a cache line can group together objects that are not shared but used by different processors (false sharing), causing unnecessary invalidations for the line. CGCT decouples the granularity at which coherence is maintained from the granularity at which data is cached, providing a new solution to the long-standing problem of how to exploit spatial locality without cache fragmentation, long transfer times, or false sharing.

Dubnicki and LeBlanc proposed adjustable cache line sizes [15]. This allows the hardware to dynamically increase/decrease the size of individual lines to trade off spatial locality and false-sharing based on application needs. The line size is adjusted by splitting a large, built-in cache line into smaller, adjacent lines when there is false-sharing, and merging adjacent lines when there is spatial locality. However, like sublining, the built-in line size is limited by cache fragmentation. Veidenbaum, Tang, Gupta, Nicolau, and Ji later proposed an adjustable cache line size scheme that uses a small hardware cache line (e.g., 8B), and fetches multiple lines to make a larger "virtual" cache line if spatial locality is present [16]. This scheme does not suffer from cache fragmentation, but adds latency to fetch multiple lines, and can require multiple lines be written back to memory to make room for a virtual line. Coarse-Grain Coherence Tracking does not increase fetch latency; the cache organization is not changed.

2.2 Sectored/Sublined Caches: Decoupling Coherence from Caching

Sector caches have been proposed to decouple the cache line size from the transfer size and/or the granularity over which coherence is maintained [12, 17, 18, 19, 20, 21, 22, 23, 24, 25]. Sector caches have large entries (sectors) containing multiple contiguous cache lines sharing one tag. Sector caches are often referred to as subblock caches or subline caches in the literature. Using large sectors reduces the number of cache locations, reduces tag storage overhead [12, 18, 24], and can "minimize the extent of the associative search" [17]. A small line size is used as the unit of data transfer to minimize transfer latency and efficiently utilize data bandwidth [12, 24], or it is used as the granularity over which coherence is maintained to avoid increased false-sharing [20, 21, 22], or both. Some designs transfer additional lines to exploit spatial locality [23], or prefetch additional lines later [12]. Similar to Coarse-Grain Coherence Tracking, lines belonging to sectors can be obtained from memory without a broadcast if coherence state is maintained for the sector; however this is limited to exploiting spatial locality sacrificed by having a small transfer size. Also, the partitioning of a cache into sectors increases internal fragmentation, increasing the cache miss rate significantly for some applications [18, 19, 25]. There have been proposals to fix this problem, including Decoupled Sectored Caches [19], CAT caches [25], and the Pool of Subsectors Cache Design [18]. All of which achieve lower miss rates by enabling sectors to share space for data, breaking the one-to-one mapping of data to cache tag entries in a traditional sectored cache. Decoupled Sectored Caches allow multiple cache tag entries to correspond to an entry in the data array, and tag bits are added to the data array such that a cache hit occurs if there is a match in both the tag array and the data array [19]. CAT caches store a pointer with the data that points to an entry in the cache tag array, such that

address matching is performed on the tag pointed to by the matching line in the data array. A tag represents a sector, and as lines are allocated in the cache the tag array is searched for the corresponding sector, and if it is found the pointer for the newly allocated line in the data array is set to that entry. The Pool of Subsectors Cache Design has more sectors in a set than there is space for in the data array. There is room in the data array for only a subset of the lines from all the sectors in the set, and the sectors share space for data [18]. Each sector address tag keeps pointers into the data array for each line. Although not all of the lines from a sector are used, space in the data array is not wasted, and sectors are only replaced when there is no longer room in the tag array (leading to fewer cache misses). In contrast to all these techniques, Coarse-Grain Coherence Tracking is implemented with a logically separate structure, does not place restrictions on the placement of data in the cache, and can track memory beyond the capacity of the cache to exploit more spatial and temporal locality.

Similar to Coarse-Grain Coherence Tracking, lines belonging to sectors in an exclusive state can be obtained from memory without a broadcast. However, Coarse-Grain Coherence Tracking is implemented with a logically separate structure and does not place restrictions on the placement of data in the cache. Coarse-Grain Coherence Tracking can maintain information for large regions of data, beyond that in the cache, without increasing false-sharing.

2.3 Optimizing Coherence Enforcement

Directory-based cache coherence protocols improve the scalability and efficiency of shared-memory multiprocessor systems [26, 27, 28]. Systems with directory-based cache coherence protocols contain a distributed directory, a hardware table with entries for each line of memory

for keeping track of which processors are caching the data. Each processor node contains a portion of the directory and the memory over which it maintains coherence. Memory requests are first sent to the processor node containing the directory for the requested line (the *home node*). The directory is accessed to obtain the list of processors sharing the line, and the request is then forwarded to the processors on that list. These processors check their caches for the requested data, and send their responses to the requesting processor. Directory-based systems do not broadcast requests; they forward requests to only the processors that have the requested data. Hence, they have very low request traffic and scale to very large numbers of processors. However, the three network hops required for a cache-to-cache transfer penalize requests to shared data. Directory-based systems essentially trade latency for scalability.

Some processor architectures, such as PowerPC [29] provide bits that the operating system can use to mark virtual memory pages as *coherence not required* (i.e., the "WIMG" bits). Taking advantage of these bits, the hardware does not need to maintain coherence or broadcast requests for data in these pages. However, in practice it is difficult to use these bits because they require OS support, complicate process migration, and are limited to virtual-page-sized regions of memory [30].

Ekman, Dahlgren, and Stenström proposed the *Page Sharing Table* (PST), a snoop-energy reduction technique for chip-multiprocessors with virtual caches [31]. This technique uses vectors that identify sharing at the page level. Every node keeps precise information about the pages it is caching. This information is used to form a page-level sharing vector in response to coherence requests. Subsequent requests are snooped only by those nodes that have lines within the same page, reducing energy consumption. Additional bus lines are required for broadcasting

and collecting the sharing vectors. Occasionally, flushing of the cache contents is necessary to maintain correctness.

Moshovos et al. proposed Jetty, a snoop-filtering mechanism for reducing cache tag lookups [32]. This technique is aimed at saving power by predicting whether an external snoop request is likely to hit in the local cache, avoiding power-consuming cache tag lookups if they are unnecessary. Like Coarse-Grain Coherence, Jetty can reduce the overhead of maintaining coherence; however Jetty does not avoid sending requests and does not reduce snoop request latency.

Moshovos concurrently proposed a technique based on Jetty that avoids sending requests as well as tag lookups [33, 34, 5]. It uses a Jetty-type hash table to conservatively predict what regions are cached (Cached Region Hash), and a separate structure (the Not Shared Region Table, or NSRT) to keep track of which regions do not have lines cached by other processors. The Jetty filter for each processor is used to provide an additional bit in the snoop response indicating whether lines in the region are cached, and this bit is stored in the requesting processor's NSRT. The NSRT is consulted on cache misses to determine if a snoop required. This technique is similar to CGCT, but uses imprecise information to reduce storage overhead and complexity. However, this technique is focused only on data requests for non-shared data, and does not filter snoop requests as effectively.

Zebchuk and Moshovos recently proposed RegionTracker, a new technique that uses coarse-grain tracking of data in the low-level on-chip caches to reduce cache tag lookup latency and power consumption [35]. Like Jetty and RegionScout, RegionTracker uses a Cached Region Hash to efficiently track regions from which the processor is caching lines. A new structure, the

Cached Block Vector (CBV), is added to track the status and location of lines in regions recently touched by the processor. When a region is touched for the first time (i.e., the CRH entry indexed by the region address of a processor request has a zero-count), an entry for the region is allocated in the CBV. The CBV contains information for each line in the region, such as whether it resides in the cache, in which way it is located if the cache is set-associative, and coherence information. This information is updated by cache allocations, evictions, and coherence state changes such that the data in the CBV accurately portrays a subset of the information in the cache tag array. Processor requests first check the CBV for the region to determine its status and location in the data array before checking the large, slow, and power-hungry cache tag array. If the region is present in the CBV, the request can obtain data from the cache data array (from the way pointed to by the CBV), or if the line is not cached begin an external request right away (without having to first check the cache tag array). This reduces latency and power consumption for processor requests while conserving cache tag lookup bandwidth, at the cost of a small increase in latency if the requested region is not in the CBV. Due to spatial locality, a significant portion of the processor requests hit in the CBV. Furthermore, RegionTracker can potentially supplement a RegionScout implementation to further reduce cache tag lookup power consumption.

2.4 Prefetching Region Data

Lin, Reinhardt, and Burger proposed Scheduled Region Prefetching (SRP) [6]. SRP aggressively prefetches data at the granularity of regions to exploit spatial locality beyond the cache line. To avoid hurting performance with too many memory requests, prefetches are performed only when

the Rambus DRAM channels are idle. To mitigate cache pollution, prefetched lines are inserted into the cache with low replacement priority. They later extended this work with density vectors [7] to mitigate the copious data traffic created by SRP. Density vectors consist of a bit vector for each region with bits set for each line accessed during an epoch; an epoch ends when a line is requested a second time. Only lines accessed previously are prefetched again to avoid wasted bandwidth. Later, software hints were added to further improve prefetch accuracy and avoid superfluous prefetches [36]. Stealth Prefetching uses similar techniques, such as prefetching only lines touched since the last prefetch, using the RCA to track which lines in a region were touched and/or are present in the cache. However, an important distinction is that Stealth Prefetching is designed to work in multiprocessor systems, only prefetches non-shared data, and does so without increasing broadcast traffic.

Zhang and Torrellas proposed *Memory Binding and Group Prefetching,* a technique that uses software hints to identify groups of data that are accessed together (e.g., fields in a record) and uses simple hardware to prefetch the data in the groups together [37]. This work was targeted specifically at irregular applications that do not exhibit large amounts of spatial locality (for which large cache lines and sequential prefetching do not work well) but can improve performance for both regular and irregular applications.

2.5 Power-Efficient DRAM Speculation

Fan, et al. investigated memory controller policies for manipulating DRAM power states in cache-based systems [38]. This research is focused on utilizing the low power modes of modern DRAM modules to power down modules not in use. Analytical modeling was used to study the

gap between clusters memory requests and the threshold time after which the module should change state. Their results indicated that the best solution was to power down DRAM modules as soon as they become idle, and not try to predict how long they would remain idle. Power-Efficient DRAM Speculation can extend this work by increasing the effective idle time of memory modules, that is the time between useful data reads, and allowing DRAM modules to remain in low power modes for longer periods of time. In contrast, PEDS actually reduces the number of DRAM reads, reducing active power and potentially allowing DIMMs to stay in low-power modes for longer periods.

Delaluz, et al. proposed Scheduler-Based DRAM Energy Management, in which the operating system transitions DRAM modules to low-power operating modes to reduce power [39]. The operating system scheduler keeps track of accesses to memory modules made my processes, and attempts to power down memory modules where possible without hurting performance. This technique benefits from the global view that the scheduler has, as opposed to compiler-based approaches, and requires little hardware support. However, the authors note that this technique can be used in concert with hardware techniques to optimize further.

Hur and Lin proposed adaptive memory schedulers that use the history of recently scheduled DRAM operations to decide which available DRAM operations to schedule next [40, 41]. Operations are prioritized to minimize latency and balance the mix of reads and writes to that of the application. This is done with a set of history-based FSM's that the scheduler adaptively selects depending on workload behavior. This work is very useful, but the proposed schedulers do not take into account the fact that some DRAM reads are performed unnecessarily. Power-Efficient DRAM Speculation adds a new dimension to this work, allowing schedulers to

not only choose between reads and writes based on hardware hazards and the program mix, but also based on whether a given read is likely to fetch useful data.

2.6 Improving Store Memory-Level Parallelism

Chou, Spracklen, and Abraham proposed the Store Miss Accelerator (SMAC) to reduce the performance impact of stores that miss in the cache [42]. The SMAC is an associative array that contains information about lines recently cached by the processor. Each entry represents a 2KB region of memory, and has a bit for each 64B cache line in the region that is set when a modified copy of the line is evicted from the cache. The bit remains set unless another processor requests the line or the entry is evicted from the SMAC. On a store miss, if the corresponding region is present in the SMAC and the bit for the line is set, an exclusive copy will be obtained from memory. This information is exploited by writing the store data to the cache early, before the rest of the line is retrieved from memory, and committing the store to free space in the processor queues. The updated bytes in the cache are merged with the rest of the cache line when it arrives from memory. This technique can reduce pressure on processor queues, and reduce stalls from these structures filling up. However, in addition to storage space for the SMAC, this technique requires that a valid bit be added to the cache for each individual byte, increasing cache storage requirements by more than 10%. Nonetheless, this optimization is an application of Coarse-Grain Coherence Tracking and could be implemented with a Region Coherence Array.

2.7 Optimizing Caching Policies

The concept of summarizing information about cached data at a region granularity and using the information to optimize subsequent data accesses was first proposed by Johnson, Hwu, Merten, and Connors [43, 44, 45, 46]. They defined a *macroblock* as a group of adjacent cache-line-size chunks of memory, and proposed adding a tagged hash table (the *Memory Address Table, or MAT*) to each level of the cache hierarchy, to detect and better exploit temporal and spatial locality [43]. Each entry contains saturating counters to record when cached data is reused (temporal locality), and when different bytes within a cache line are used (spatial locality). Based on these counts, levels of the cache are bypassed to avoid replacing useful data with data that has low temporal locality, and only the needed bytes are fetched from memory if little spatial locality is present. Bypassed data is placed in a small associative buffer, like a victim cache [47], allowing reuse of data that has little temporal or spatial locality. This work was extended in [44], where a *Spatial Locality Detection Table (SLDT)* was proposed. The SLDT is a small associative structure that tracks spatial locality across adjacent cache lines, which is later recorded in the MAT for long term tracking. This information is used to adjust the memory fetch size from a single cache line to multiple adjacent cache lines in a macroblock when significant spatial locality is present. This research was extended again with a theoretical analysis of the upper bounds and results for Windows applications in [46]. A similar technique can be implemented using CGCT, adding bits to the storage for each region to detect spatial and temporal locality.

Martin, et al. subsequently proposed Destination-Set Prediction using macroblocks to aggregate information for spatially-related data [48]. Destination-Set Prediction is a technique for predicting the destination-set of a memory request, the subset of processors in the system that

must receive it to maintain coherence. By predicting the destination set early, a memory request can be sent directly and exclusively to that set, without broadcasting to all the processors in the system and without first consulting a directory (i.e., multicast snooping [49]). An accurate predictor can enable a system with request traffic that approaches that of a directory protocol and average memory latency that approaches that of a broadcast protocol. By aggregating information for spatially-close lines, the proposed predictor could exploit spatial locality while using less storage.

3. Experimental Methods

This chapter describes the simulation infrastructure, baseline system parameters, and workloads used to evaluate Coarse-Grain Coherence Tracking in this dissertation. Section 3.1 describes the simulation infrastructure; this is followed by a list of baseline system parameters used for simulations in Section 3.2 (simulation parameters for hardware added to implement Region Coherence Arrays, RegionScout, Stealth Prefetching, and Power-Efficient DRAM Speculation are given in their respective chapters). Finally, Section 3.3 discusses the workloads simulated and their datasets.

3.1 Simulation Infrastructure

In this dissertation, detailed timing simulations are performed with *PHARMsim* [50], an execution-driven multiprocessor simulator built on top of SimOS-PPC [51]. *PHARMsim* models out-of-order processors with two-level hierarchy with MOESI. The simulator implements the PowerPC ISA [29] and runs both user-level and system code from applications running on IBM's AIX 4.3.

3.2 Baseline System Parameters

For the baseline system, I modeled a four-processor broadcast-based shared-memory multiprocessor with a Fireplane-like interconnect and 1.5GHz processors with resources similar to the UltraSparc-IV [52]. Unlike the UltraSparc-IV, the processors feature out-of-order instruction issue, on-chip 2MB L2 caches (1MB per processor), and support sequential consistency. A

detailed list of parameters for the baseline system is in Table 3.1. All simulation results presented in this dissertation use these baseline parameters, except Chapter 5. In Chapter 5 the size of the L2 caches was halved to correlate with earlier work on RegionScout Filters done by Moshovos [5, 33].

Table 3.1: Simulation parameters

System	
Processors Cores Per Processor Chip	2
Processor Chips Per Data Switch	2
DMA Buffer Size	512-Byte
Processor	
Processor Clock	1.5GHz
Processor Pipeline	15 stages
Fetch Queue Size	16 instructions
BTB	4K sets, 4-way
Branch Predictor	16K-entry Gshare
Return Address Stack	8 entries
Decode/Issue/Commit Width	4/4/4
Issue Window Size	32 entries
ROB	64 entries
Load/Store Queue Size	32 entries
Int-ALU/Int-MULT	2/1
FP-ALU/FP-MULT	1/1
Memory Ports	1
Caches	
L1 I-Cache Size/Associativity/Block-Size/Latency	32KB 4-way, 64B lines, 1-cycle
L1 D-Cache Size/Associativity/Block-Size/Latency	64KB 4-way, 64B lines, 1-cycle (Writeback)
L2 Cache Size/Associativity/Block-Size/Latency	1MB 2-way, 64B lines, 12-cycle (Writeback)
Prefetching	Power4-style, 8 streams, 5 line runahead MIPS R10000-style exclusive-prefetching
Cache Coherence Protocols	Write-Invalidate MOESI (L2), MSI (L1)
Memory Consistency Model	Sequential Consistency
Interconnect	
System Clock	150Mhz
Snoop Latency	106ns (16 cycles)
Critical Word Transfer Latency (Same Data Switch)	20ns (3 cycles)
Critical Word Transfer Latency (Same Board)	47ns (7 cycles)
Critical Word Transfer Latency (Remote)	80ns (12 cycles)
Memory	
Memory Controllers	2 (1 Per Chip)
DRAM Latency	106ns (16 cycles)
DRAM Latency (Overlapped with Snoop)	47ns (7 cycles)

Note that the baseline system implements two conventional forms of prefetching, namely stream prefetching [2] and exclusive prefetching [53]. These prefetchers are used in all simulations, including those performed to evaluate Stealth Prefetching.

Figure 3.1 illustrates the timing of the critical word for different scenarios of an external memory request. For direct memory requests employed by systems implementing CGCT, I assume that a request can begin one CPU cycle after the L2 miss for memory collocated with the CPU (memory controller is on-chip), after two system cycles for memory connected to the same data switch, after four system cycles for memory on the same board, and after six system cycles for the memory on other boards. The Fireplane system overlaps the DRAM access with the snoop; so direct requests see the full DRAM latency (9 system cycles).

The request latency is shortest for requests to the on-chip memory controller; otherwise the reduction in overhead versus snooping is offset by the latency of sending requests to the remote memory controller. This makes the results conservative because the version of AIX used for evaluation makes no effort to place data in physical memory close to the processors that use it and no effort to schedule processes on processors close to the data they need.

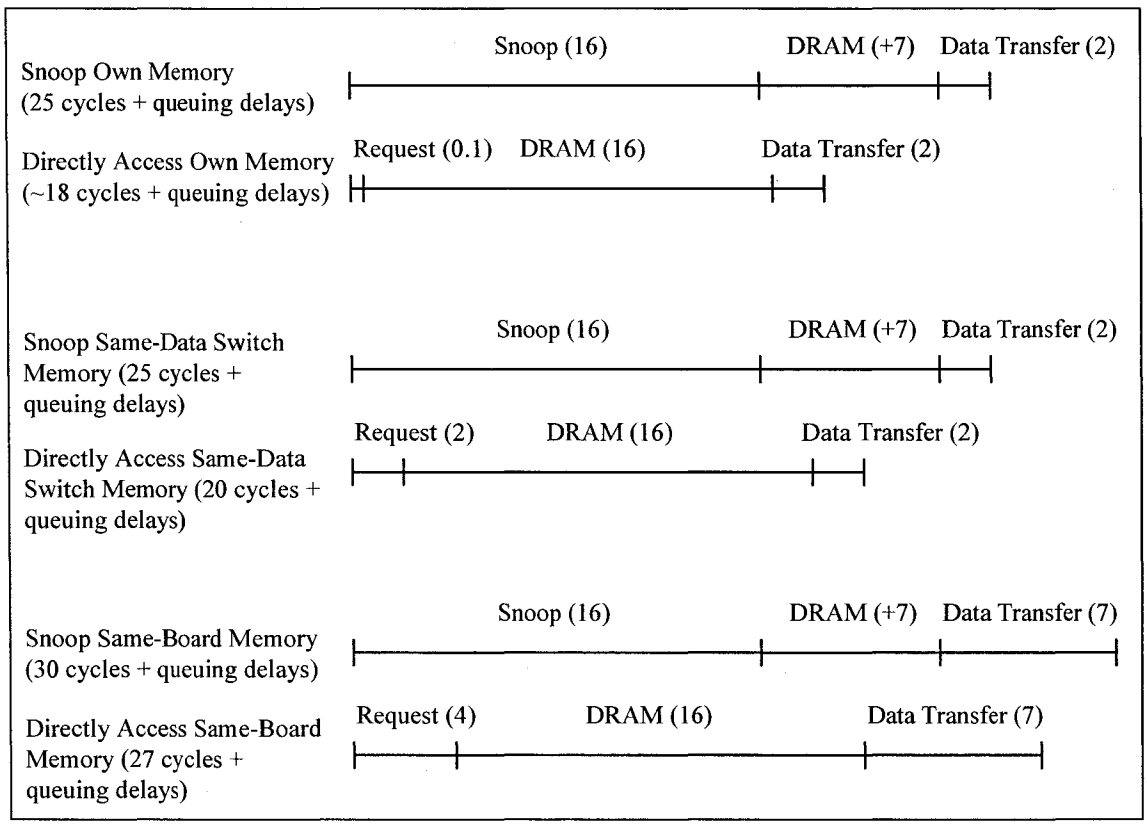

Figure 3.1: Memory request latency.

Requests for local memory benefit the most from CGCT due to the relatively large reduction in latency. Requests for memory farther away have a lower relative benefit due to the increasing request/data transfer times.

3.3 Workloads

For workloads I use a combination of commercial, scientific, and multiprogrammed benchmarks. Simulations are started from memory and disk checkpoints taken on an IBM RS/6000 server running AIX 4.3 and include system code. Cache checkpoints are used to warm the simulated system's L2 caches before starting simulations. The benchmarks and their datasets are in Table 3.2.

Table 3.2: Benchmarks for timing simulations.

Category	Benchmark	Comments
Scientific	Barnes	SPLASH-2 Barnes-Hut N-body Simulation, 8K Particles
	Ocean	SPLASH-2 Ocean Simulation, 514 x 514 Grid
	Raytrace	SPLASH-2 Raytracing application, Car
Multiprogramming	SPECint95Rate	Standard Performance Evaluation Corporation's 1995 CPU Integer Benchmarks
	SPECint2000Rate	Standard Performance Evaluation Corporation's 2000 CPU Integer Benchmarks, Combination of reduced-input runs
Commercial	SPECjbb2000	Standard Performance Evaulation Corporation's Java Business Benchmark, IBM jdk 1.1.8 with JIT, 20 Warehouses, 2400 Requests
	SPECweb99	Standard Performance Evaulation Corporation's World Wide Web Server, Zeus Web Server 3.3.7, 300 HTTP Requests
	TPC-H	Transaction Processing Council's Decision Support Benchmark, IBM DB2 version 6.1, Query 12 on a 512MB Database
	TPC-W	Transaction Processing Council's Web e-Commerce Benchmark, DB Tier, Browsing Mix, 25 Web Transactions
	TPC-B	Transaction Processing Council's Original OLTP Benchmark, IBM DB2 version 6.1, 20 clients, 1000 transactions

Results from individual benchmarks are averaged together giving equal weight to each workload category. First, the arithmetic mean is computed for results from scientific benchmarks. Next, the multiprogrammed workloads are averaged together, followed by the commercial workloads. The resultant arithmetic means are then combined together to yield an overall arithmetic mean that weights each category equally.

Due to workload variability, in all experiments several runs were performed for each benchmark with small random delays added to memory requests to perturb the system [54]. The results of these runs are averaged together, and the 95% confidence intervals are shown where appropriate.

4. Coarse-Grain Coherence Tracking

This chapter presents CGCT, a new technique that avoids broadcasts and filters unnecessary cache tag lookups in a broadcast-based shared-memory multiprocessor system (Section 4.1). Section 4.2 presents Region Coherence Arrays, an effective implementation of CGCT. This is followed by a discussion of the protocol that a Region Coherence Array uses to track the local and global status of regions (Section 4.3). This is followed by a delineation of the system modifications required to incorporate a Region Coherence Array (Section 4.4). The following section presents results characterizing the effectiveness of Region Coherence Arrays (Section 4.5). Section 4.6 analyzes the remaining potential for Region Coherence Arrays to avoid broadcasts. Section 4.7 summarizes the findings of the chapter.

4.1 Coarse-Grain Coherence Tracking

CGCT is a new technique that allows a processor to determine in advance that a memory request does not require a broadcast [5, 33, 34, 55]. When a broadcast snoop is performed, a system with CGCT collects coherence information for not only the line, but a large region of memory around the requested line (where a region is an aligned area of physical memory that encompasses a power-of-two number of cache lines). This information is stored and used to determine whether subsequent requests must be broadcast to coherently access memory. Data requests that do not require a broadcast are sent directly to memory, conserving broadcast bandwidth, conserving cache tag lookup bandwidth, and for some systems, reducing memory latency. Non-data requests

such as upgrades and invalidations that do not require a broadcast are not sent externally at all, reducing their latency considerably.

CGCT can be implemented as a layered extension to an otherwise conventional multiprocessor system. A conventional cache coherence protocol is employed to maintain coherence over the processors' caches. However, unlike a conventional multiprocesser system each processor contains additional hardware for monitoring the coherence status of large regions. This hardware keeps track of regions from which the processor is caching lines, and when snooped by external requests, it provides a *region snoop response*. This response is piggybacked onto the conventional snoop response sent back to the requesting processor and it is used by the requesting processor to determine if broadcasts are necessary for subsequent requests.

CGCT can extend a broadcast-based multiprocessor system to achieve much of the benefit of a directory-based multiprocessor system [26, 27, 28], including low interconnect and cache tag lookup traffic and low-latency access to non-shared data. However, with an underlying broadcast protocol, intervention latency is kept low and hardware overhead is small compared to implementing a directory. CGCT accomplishes this by exploiting spatial locality beyond the cache line and temporal locality beyond the capacity of the cache, without increasing false-sharing or internal fragmentation.

4.2 Region Coherence Arrays

This dissertation proposes Region Coherence Arrays (RCAs), an effective implementation of CGCT. An RCA is a tagged array that tracks the coherence status of regions cached by the processor. Each entry contains an address tag for the region, a region coherence state, and a

count of the lines cached by the processor (or a bit mask representing which lines in the region are cached by the processor). The *region coherence state* indicates whether the processor or other processors are sharing or modifying lines in the region and is maintained by a *region protocol* (Section 4.3).

On cache misses the RCA state is checked to determine if memory requests need a broadcast to maintain coherence. On external snoop requests the RCA is checked to provide a snoop response for the region, and to determine if the external snoop request must access the cache.

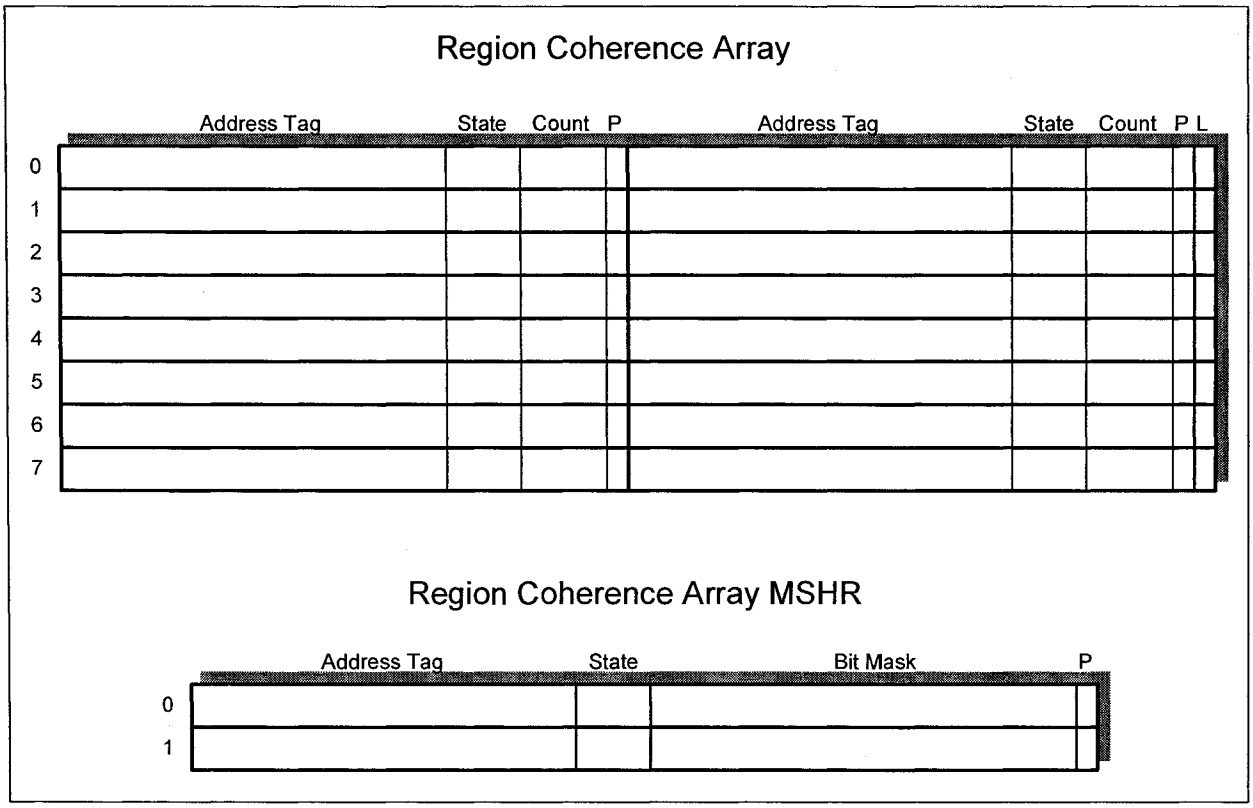

Figure 4.1: Structure of a Region Coherence Array and Region Coherence Array MSHR.

Shown is a 2-way set-associative RCA with 8 sets. Each set stores information for two regions, including address tags, region coherence states, line counts, parity bits ("P"), and a bit (L) for implementing a Least-Recently-Used (LRU) replacement policy. Parity is maintained over the address tags, state, and line counts, but not over the LRU bit. Also shown is a set of 2 RCA MSHRs, each with an address tag, region coherence state, a bit mask for the lines that have been evicted from the cache, and a parity bit.

Figure 4.2 depicts an example of how a Region Coherence Array operates. In part (a), node *A* requests line *x* in region *X*, and checks its RCA (step 1). No matching entry is found, so one is allocated and the request is broadcast (step 2). All remote nodes check their RCA and respond that they do not cache any line in region *X*. Node *A* receives the response, and updates the region

state for X to "non-shared" (step 3). In part (b), node A is about to request line y in region X and first checks its RCA (step 1). An entry is found in a "non-shared" state, so the request is sent directly to memory (step 2). In part (c), node B requests line z in region X. It checks its RCA (step 1). It does not find a matching entry, so it broadcasts its request (step 2). Upon receiving the request, node A downgrades its region state to "shared", and checks the cache for the line (step 3).

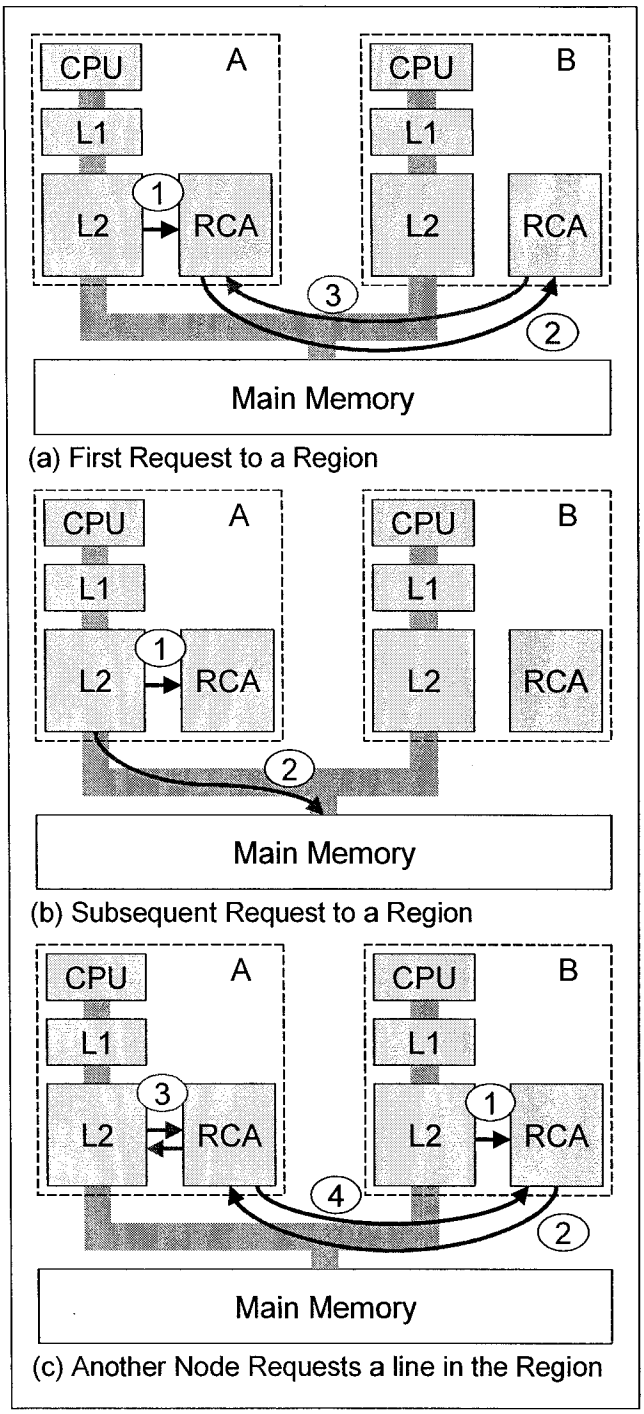

Figure 4.2: Example operation of a Region Coherence Array.

First request to a region results in a broadcast (a). Subsequent requests can go straight to memory without broadcasting (b). Another processor broadcasts a request for a line in the region (c).

4.2.1 RCA MSHRs

For processors to respond correctly to external requests, the RCA must maintain inclusion over the on-chip caches. That is, if a line is cached there must be a valid entry in the RCA for the corresponding region so that the RCA does not respond incorrectly to external requests. If the line is cached, the region is shared. Similarly, every memory request for which the requesting processor's region state is invalid must be broadcast to the system to acquire permissions to the region and to inform other processors that may also be accessing lines in the region. To maintain inclusion, lines must sometimes be evicted from a processor's cache before a region can be evicted from the RCA. However, this is made infrequent by first evicting regions from which the processor is not caching lines (using the count or bit mask in each RCA entry to detect such regions).

When a region is evicted from the RCA, its state must be buffered until all of its lines have been removed from the cache. To ensure coherence is maintained, state for the region must be maintained in the RCA as long as lines remain in the cache. However, evicting a large region can take time, and when a region is evicted it is because space is needed for another region that has been requested by the processor. To avoid stalling the processor, a small set of buffers is needed to hold state information for regions in the process of being evicted. These buffers are called *RCA MSHRs*, after the Miss Information/Status Handling Registers used in lockup-free caches [56]. When a region is evicted from the RCA, its state is moved to one of these buffers to free the RCA entry for the new region.

RCA MSHRs consist of an address tag, region coherence state, and a bit mask for each of the lines in the region to keep track of which have been evicted. As the caches perform each

eviction, they send a message to the RCA, and the corresponding bit is set in the RCA MSHR. The region may be discarded once all the lines have been accounted for and the processor no longer caches any lines from the region. If a bit-mask or line count is used in the RCA to keep track of which lines are cached, that data is also moved to the RCA MSHR. In this case, eviction requests only need to be sent for the lines known to be cached; otherwise an eviction request is sent for each line in the region.

The RCA MSHRs are accessed with the RCA on an external request, and provide a snoop response for the region while it is in the process of being evicted. In the rare case that a processor requests a line in a region that is currently being evicted from its RCA, that request is stalled until the region has been completely evicted and removed from the RCA MSHR (our evaluations found little benefit in adding the capability to reinstate the region). Finally, if a request is made for a region not in the RCA causing a region eviction to make room for it, and if all the RCA MSHRs are busy evicting other regions, the request must stall until an RCA MSHR is available.

4.3 Region Protocol

RCAs use a *region protocol* to update and maintain the region state. The region protocol observes the same request stream as the underlying conventional cache coherence protocol, and updates the region state in response to requests from the processor and other processors in the system. It ensures that the region state encodes the maximum permissions held for any line in the region by the processor (and that of other processors) so that broadcasts can be avoided and snoop-induced cache tag lookups can be filtered without violating coherence.

4.3.1 Region Protocol States

The proposed region protocol consists of seven base states (Table 4.1). First, the *Invalid* state indicates that no lines are cached by the processor and the state of lines in other processors' caches is unknown. For the other base states, the first letter of the name indicates whether there are clean or dirty copies of lines from the region cached by the processor. The second letter indicates whether other processors may have clean or dirty copies of lines from the region in their caches. The states CI and DI are the *exclusive states*; no other processors are caching lines from the region and requests by the processor for lines from the region do not need a broadcast. The CC and DC states are *externally-clean*; only reads to obtain clean copies (such as instruction fetches) can be performed without a broadcast. Finally, CD and DD are the *externally-dirty* states; broadcasts must be performed to ensure that copies of lines in other processors' caches are found.

Table 4.1: Region protocol states.

	Processor	Other Processors	Broadcast Needed?
Invalid (I)	No Cached Copies	Unknown	Yes
Clean-Invalid (CI)	Unmodified Copies Only	No Cached Copies	No
Clean-Clean (CC)	Unmodified Copies Only	Unmodified Copies Only	For Modifiable Copy
Clean-Dirty (CD)	Unmodified Copies Only	May Have Modified Copies	Yes
Dirty-Invalid (DI)	May Have Modified Copies	No Cached Copies	No
Dirty-Clean (DC)	May Have Modified Copies	Unmodified Copies Only	For Modifiable Copy
Dirty-Dirty (DD)	May Have Modified Copies	May Have Modified Copies	Yes

In addition to the base states listed above, one pending (transient) state is needed to implement the region protocol. When a region is requested for the first time, an entry in the RCA must be allocated before broadcasting the request to ensure that space is available in the RCA.

(There are a finite number of RCA MSHRs, and a region cannot be evicted until one becomes available). When the broadcast snoop is completed there must be resources available to hold the region state, else the data cannot be used without violating inclusion. Once an entry is allocated it is first assigned this pending state, indicating that the region is pending a broadcast snoop. The status of the region in other processors' caches is unknown. The region state is changed to one of the base states once the first broadcast snoop for a line from the region has completed.

The state transition diagrams depicted in Figures 4.3-4.5 illustrate the region protocol. For clarity, the exclusive states are solid gray, and the externally-clean states are shaded. Each base state transition is labeled with the request type and if applicable the global status of the requested region.

From the Invalid state, the next state depends both on the request and the snoop responses (left side of Figure 4.3). Instruction fetches and Reads of shared lines will change the region state from Invalid to CI, CC, or CD, depending on the region snoop response. Read-For-Ownership (RFO) operations and Reads that bring data into the cache in an exclusive state cause the region state to transition to DI, DC, or DD. If the region is already present in a clean state, then loading a modifiable copy updates the region state to the corresponding dirty state. A special case is for CI, which silently changes to DI when a modifiable copy of a line is loaded (dashed line).

The transitions in Figure 4.4 are upgrades based on the region snoop response. They not only update the status of the region to reflect the state of lines in the cache, but also upgrade the region to an externally-clean or exclusive state. For example, a snoop is required when in the CC state for RFO operations. If the snoop response to a prior request indicates that no processors are sharing the region anymore, the state can be upgraded to DI.

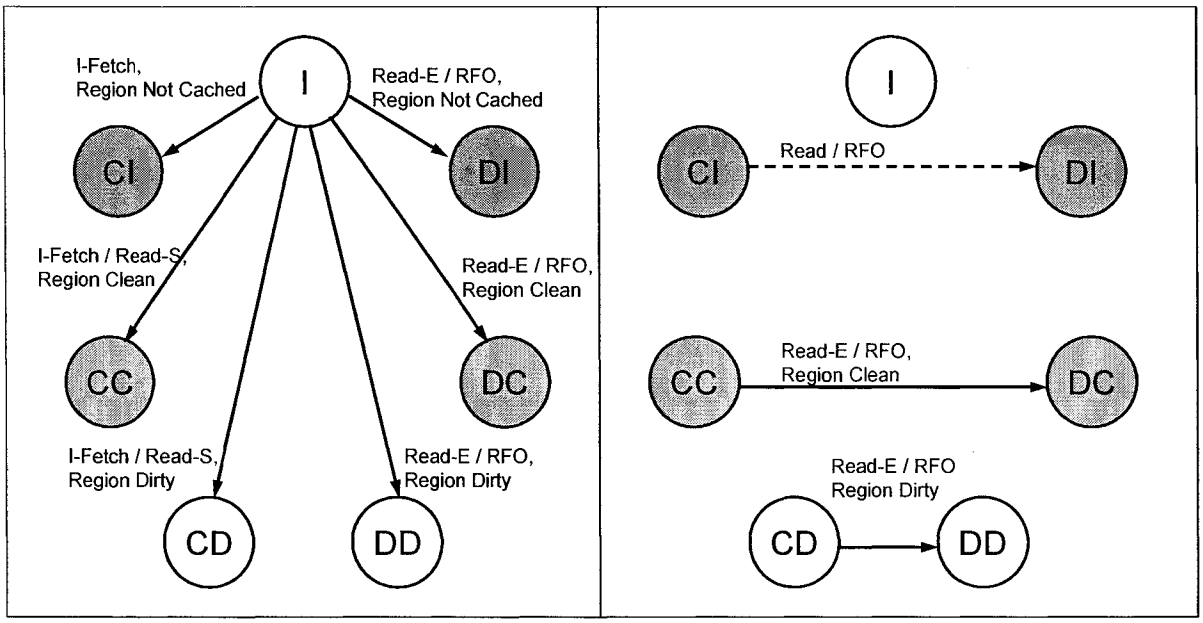

Figure 4.3: State transition diagrams for requests made by the processor

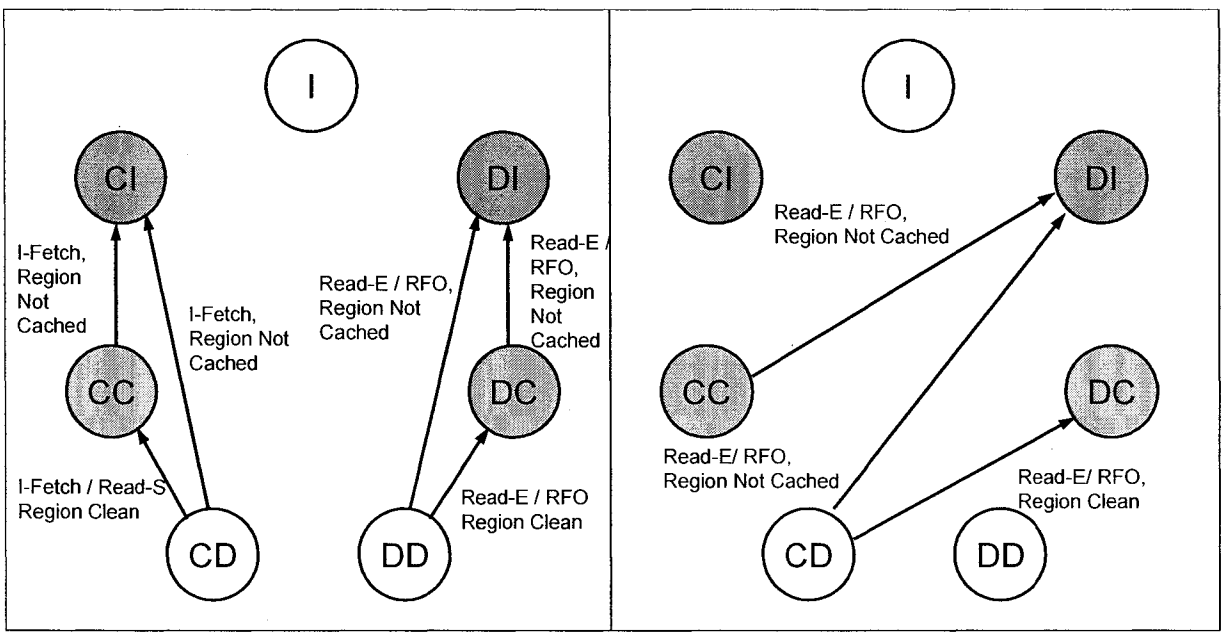

Figure 4.4: State transition diagrams for processor requests that upgrade the region state.

The top part of Figure 4.5 shows how external requests to lines in the region downgrade the region state to reflect that other processors are now sharing or modifying lines in the region. Whether the region state is downgraded to externally-dirty or externally-clean depends on whether the external request obtains a modifiable copy of the line.

The bottom part of Figure 4.5 shows the state transitions for evictions. Also shown is a form of self-invalidation (for the region state, not the cache line state as in prior proposals for dynamic self-invalidation [57]). When broadcasts cannot be avoided, it is often because the region is in an externally-dirty state. Frequently there are no lines cached by the remote processors (possibly due to migratory data). Invalidating regions that have no lines cached in response to external requests improves performance significantly for the protocol. To accomplish this, a line count is used to keep track of the number of lines from the region that are cached by the processor; the count is incremented on allocations and decremented on invalidations and replacements. If an external request hits in a region and the line count is zero, the region is invalidated so that later requests may obtain exclusive access to the region.

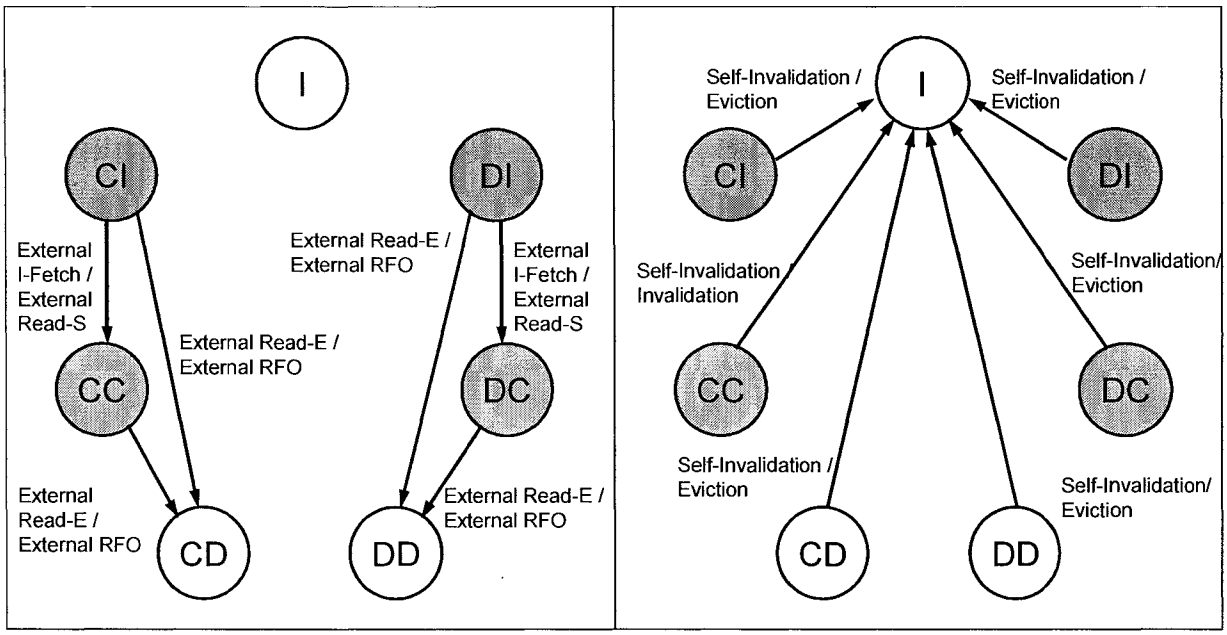

Figure 4.5: State transition diagrams for external requests

In the region protocol used in this dissertation, loads are not prevented from obtaining exclusive copies of lines. In the state diagrams above, memory read-requests originating from loads are broadcast unless the region state is CI or DI. An alternative approach can avoid broadcasts by accessing the data in externally-clean states directly from memory and loading them into the cache in a shared state, however this will lead to subsequent upgrades for many of these lines. Future work will investigate adaptive optimizations to address this issue.

4.4 System Modifications to Implement Region Coherence Arrays

There are four modifications that must be made to a broadcast-based multiprocessor system to implement CGCT with Region Coherence Arrays. First, there must be a means for processors to communicate with memory controllers without broadcasting requests. Second, additional bits are needed in the snoop response messages to convey the region snoop response. Third, chip area is

needed for the RCA, RCA MSHRs, and associated logic. Finally, the RCA needs the ability to evict cache lines for inclusion.

4.4.1 Direct Access to Memory Controllers

Systems such as those from Sun Microsystems [1] and IBM [2, 3] have the memory controllers integrated onto the processor chip; however these controllers are accessed only via external requests. Nonetheless, a direct connection from the processor to the on-chip memory controller should be straightforward to implement. For requests to memory governed by other processors' memory controllers, it may be necessary to add virtual channels to the data network so that request packets can be sent directly to memory controllers on other processor chips. Though adding request packets to the data network will consume data network bandwidth, unlike broadcast snoops, these requests can travel from point to point over an unordered network and consist of only one packet (cache lines can take four or more packets to transfer over a modern data network). Also, it is easier to add bandwidth to an unordered data network than a global broadcast network.

For systems like the AMD Opteron servers [4], no interconnect modifications are needed. The requests and data transfers use the same physical network, and requests are sent to the memory controller for ordering before being broadcast to other processors in the system. To implement Coarse-Grain Coherence Tracking in these systems, a request can be sent to the memory controller as before, but the global broadcast can be skipped. The memory data would be sent back to the requestor, as it would if no other processors responded with the data.

If a separate data network is unavailable and it is infeasible to add one, there is still potential to benefit from CGCT. In bus-based systems, requests can be sent over the bus with a bit indicating that they are "memory only". These requests will be ignored by other processors to save cache tag lookup power, and the memory controller can fetch the data from DRAM without waiting for other processors to assert the shared/owned signals on the bus.

4.4.2 Additional Bits in the Snoop Response

For the region protocol described above, two additional bits are needed in the snoop response. One bit indicates whether the region is in a clean state in other processors' caches (Region Clean), and the second indicates whether the region is in other processors' caches in a dirty state (Region Dirty). These bits summarize the region as a whole, and not individual lines. They are a logical sum of the region status in other processors' caches, excluding the requestor. This should not be a large overhead; snoop response packets already contain many bits for the address, line snoop response, ECC bits, and other information for routing / request matching.

4.4.3 Storage Space for the Region Coherence Array

Assuming 48-bit physical addresses and a 1MB, 2-way set-associative cache with 64-byte lines, each entry needs 29 bits for the address tag, three bits for the coherence state, and eight bytes to implement ECC (error correcting code) for the data. An RCA entry needs a region address tag, 3 state bits, a line count, a parity bit, and least-recently-used information. For an 8K-set, 2-way set-associative structure, storage is approximately 68 bits per set, irrespective of region size. Based on this design point, the area overhead of an RCA with the same associativity as the cache and

different numbers of sets is shown in Table 4.2. The storage overhead is given both as a ratio of the cache tag array space and as a ratio of the total cache space. For a given number of sets the overhead of all the region sizes is the same.

Table 4.2: Storage overhead for varying RCA sizes and region sizes.

2-way set-assoc. RCA, 48-bit addresses	Bits / Set	Total Kilobytes	Tag Overhead	Cache Overhead
2K-Entries	74	9.3	5.0%	0.8%
4K-Entries	72	18.0	9.7%	1.5%
8K-Entries	70	35.0	48.6%	2.8%
16K-Entries	68	68.0	88.3%	5.5%

For the same number of RCA entries as cache entries the cache space overhead is 5.5%. If the number of entries is halved or quartered, the cache area overhead is reduced to 2.8% and 1.5%, respectively. The relative overhead is less for systems with larger, 128-byte cache lines like the current IBM Power systems [2, 3].

In addition to the storage space for the RCA, space is needed for the RCA MSHRs. Each such MSHR has an address tag, region coherence state, and minimally a bit-mask for the lines that have been evicted. For the design point above, there are up to 4 bytes for the tag, three bits for the state, and up to eight bytes for the mask. Including LRU information and ECC, approximately 14 bytes are needed for each entry. In early evaluations, eight RCA MSHRs were found to be enough to ensure that processor stalls were extremely rare in a system with an RCA with the same number of entries as the cache. For eight RCA MSHRs, the storage space is 112 bytes, 0.1% more storage than an RCA without RCA MSHRs.

4.4.4 Inclusion over the Caches

For a system using Region Coherence Arrays to provide the correct response to an external request, inclusion must be maintained over the caches. If a region is not present in the RCA, then there must not be any lines from the region cached by the processor, else the RCA might falsely respond that it is not caching lines from the region. Therefore, when a region is evicted from the RCA, the lines in that region must be evicted from the processor's caches.

To maintain inclusion, the RCA needs the capability to send requests to the caches to evict lines. These evictions may be performed in the background with low priority. However, they must be ordered with respect to external snoop requests to avoid race conditions. In addition, once a line has been evicted for inclusion, a message must be sent back to the RCA to confirm that the eviction has been completed so that the RCA MSHR can be freed.

For inclusive cache hierarchies, the RCA can send requests to evict cache lines to the lowest level of cache above which inclusion is maintained, and any lower levels. If not, each level must be checked for cached copies of lines in the region (these are also the levels that must be snooped by external requests). Inclusion does not need to be maintained for caches beyond the broadcast coherence mechanism, such as caches for DRAM, provided these caches can be accessed without broadcasting requests.

4.4.5 Request Ordering

In traditional broadcast-based shared-memory multiprocessor systems the ordering point is the broadcast network or bus. The order in which requests are broadcast is the order in which processors observe the requests, and a processor knows that its request is ordered when it

obseves its own request on the broadcast network. In reality, the ordering of memory requests is more subtle. While external requests are ordered with respect to each other by the broadcast interconnect, external requests are not ordered with respect to processor requests until they access and update the coherence state of lines in the caches.

To implement Coarse-Grain Coherence Tracking correctly, requests sent directly to memory must be ordered with respect to other requests for correctness. These requests originate from processor requests that miss in the cache, and hence are ordered with respect to other processor requests. However, they do not use the broadcast interconnect, and must be ordered with respect to external requests before being sent to memory. Previous external requests must have updated the cache state, and subsequent requests broadcast by other processors must observe that the processor has an outstanding direct request for the line.

To properly order requests in a system that implements CGCT with a Region Coherence Array, the ordering point is the Region Coherence Array and not the cache tags. Requests must allocate a region in the Region Coherence Array and/or update the region state before sending a request directly to memory. Similarly, external requests must be sent to the Region Coherence Array in the order in which they appear on the broadcast network. Processor requests must arbitrate for the Region Coherence Array with external requests so that the region state is updated atomically, by one request at a time.

Once processor requests have accessed/updated the region state, they are ordered and may be sent directly to memory. In the baseline system, a message is then sent back to the caches to inform the cache coherence protocol that the request has been ordered, and the cache must respond to invalidations and requests for data from other processors. External requests that reach

the Region Coherence Array after a processor request has been sent directly to memory will observe that the processor is caching lines from the region, and be sent to the caches. The external request will reach the caches after the message indicating that the request is ordered, and find the line in a valid, transient state waiting for data from memory. External requests that reach the Region Coherence Array before a processor request will (a) not find the region cached, or (b) will find the region in the Region Coherence Array and update its state to externally-clean or externally-dirty, depending on the request type. In either case, the processor request has not been ordered and will not affect how the external request is handled by the caches.

4.5 Simulation Results

In this section simulation results for Region Coherence Arrays are presented. First, the effectiveness of Region Coherence Arrays for avoiding broadcasts is studied. This is followed by data on the effectiveness of Region Coherence Arrays for filtering unnecessary snoop-induced cache tag lookups. After that, execution time and broadcast traffic improvements are measured. Finally, we characterize the remaining potential for Region Coherence Arrays.

4.5.1 Effectiveness at Avoiding Broadcasts

Figure 4.6 shows results for a Region Coherence Array with the same number of sets and associativity as the cache, and varying region sizes. In this figure, both the number of requests for which a broadcast is unnecessary (from Figure 1.2) and the number of requests that are sent directly to memory or avoided altogether are shown. Figure 4.7 shows this same data, broken down into broadcasts avoided by spatial locality and temporal locality. Here, *spatial locality* is

defined as subsequent accesses to other lines in a region while the region is present in the Region Coherence Array, and *temporal locality* is defined as subsequent accesses to the same line in a region while the region is resident in the Region Coherence Array. These figures are each broken into three parts for clarity, with scientific workloads shown first, followed by multiprogrammed, and commercial workloads.

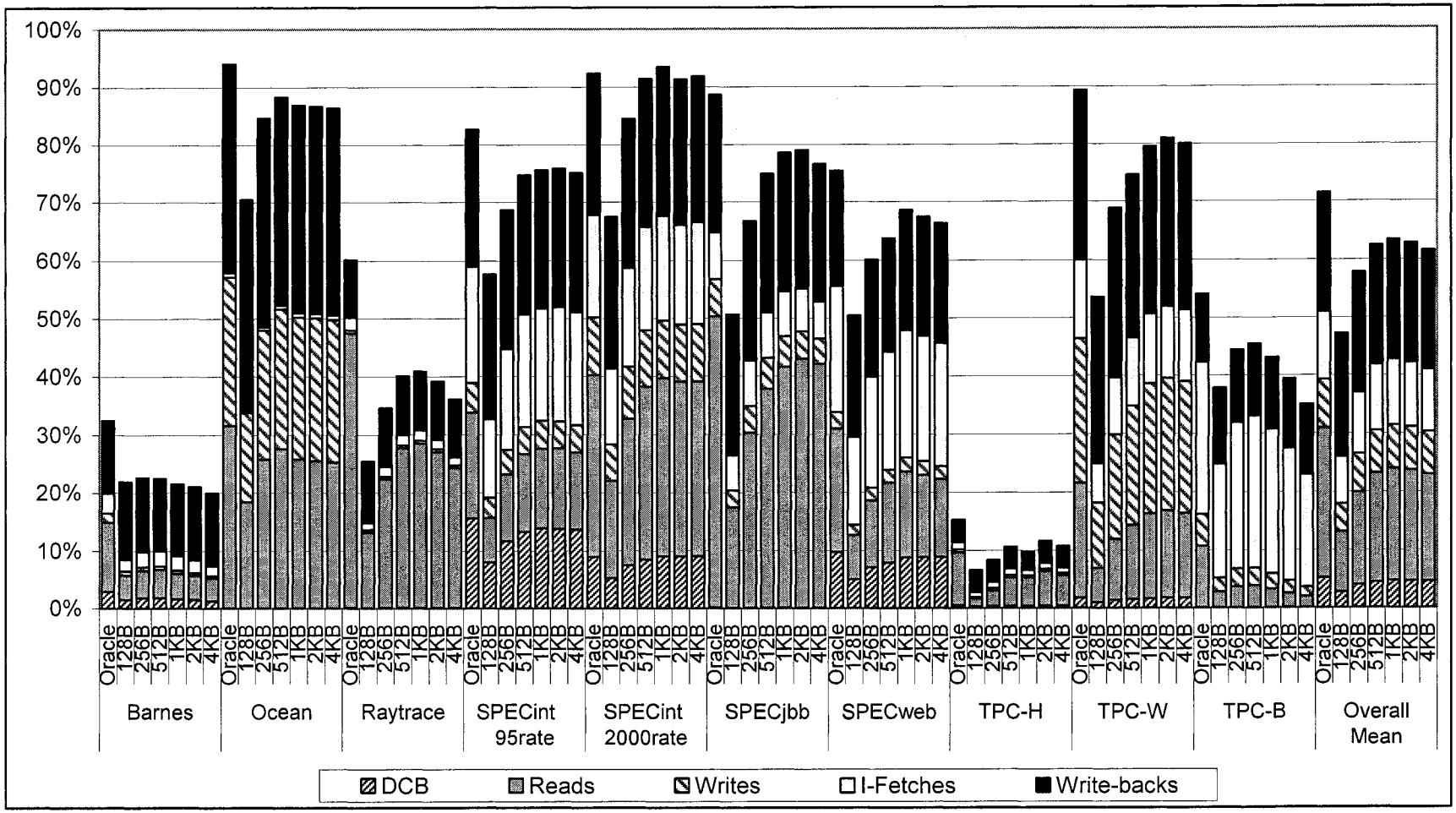

Figure 4.6: Broadcasts avoided by Region Coherence Arrays

Effectiveness of Region Coherence Arrays for avoiding unnecessary broadcasts and eliminating unnecessary external requests in a four-processor system. The leftmost bar for each benchmark shows the requests for which broadcasts are unnecessary (from Figure 1.2), and the adjacent bars show the percentage avoided for each region size.

Except for Barnes and TPC-H, all the applications experience a large reduction in the number of broadcasts. Barnes experiences a 20-23% reduction, while TPC-H experiences only a 7-12% reduction. However, even for these cases, Coarse-Grain Coherence Tracking is capturing a significant fraction of the total opportunity. TPC-H, for example, benefits a great deal from CGCT during the parallel phase of the query. But later, when merging information from the different processes there are a lot of cache-to-cache transfers, leaving a best-case reduction of only 15% of the broadcasts.

Write-backs are included in Figure 4.7; they are on top of the stacks to clearly separate the contribution of the other requests. Write-backs do not need to be broadcast, strictly speaking, but they are typically broadcast in conventional systems to find the appropriate memory controller and simplify request ordering. Because of the multitude of memory configurations resulting from different system configurations, DRAM sizes, DRAM slot occupancies, and interleaving factors, it is difficult for all the processors to track the mapping of physical addresses to memory controllers [30]. And, in a conventional broadcast-based system, there is little benefit in adding address-decoding hardware, network resources for direct requests, and protocol complexity just to accelerate write-backs. In contrast, a system that implements Coarse-Grain Coherence Tracking already has the means to send requests directly to memory controllers, and one can easily incorporate an index for the memory controller into the RCA entry. Consequently, there is a significant improvement in the number of broadcasts that can be avoided, but this will only affect performance if the system is network bandwidth constrained (not the case in the simulations here).

Examining Figure 4.7, it can be seen that more broadcasts are avoided from exploiting temporal locality than spatial locality. This is surprising; the initial hypothesis was that the major contributor would be spatial locality; subsequent accesses to other lines in the same region. In contrast, the larger contribution is from subsequent accesses to the same line in a region.

There are two reasons for this: First, the Region Coherence Array in these simulations has the same number of entries as the cache, and tracks considerably more data. The Region Coherence Array can hence exploit temporal locality beyond the cache, optimizing subsequent requests to lines that have been evicted from the cache. Second, there are cases in which subsequent broadcasts are performed for data in the cache, such as requests to upgrade a cached copy to a modifiable state and OS requests to flush data to memory.

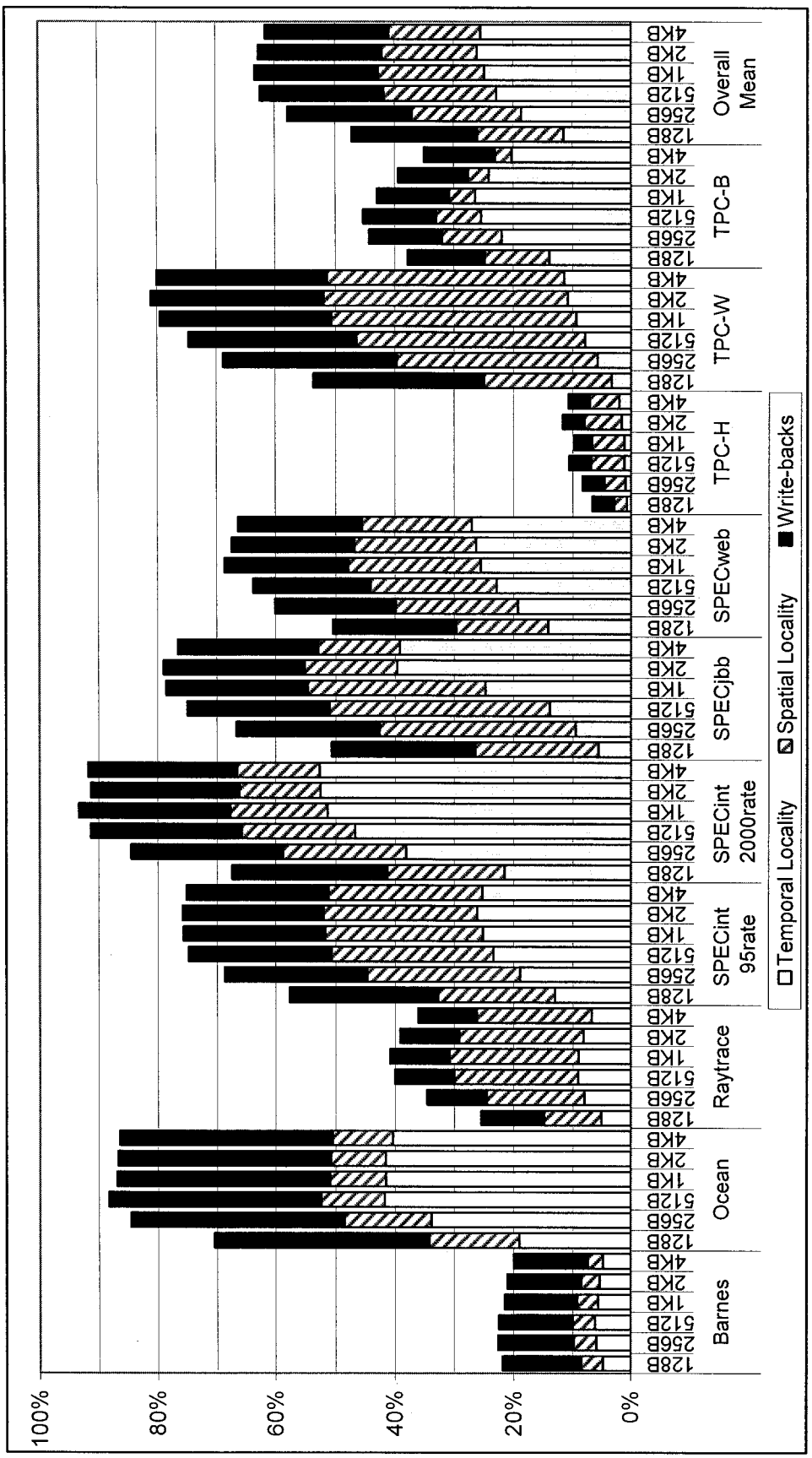

Figure 4.7: Broadcasts avoided broken down by temporal/spatial locality.

4.5.2 Effectiveness at Filtering Snoop-induced Cache Tag Lookups

Figure 4.8 shows both the percentage of snoop-induced cache tag lookups that are unnecessary (from Figure 1.4) and the number of snoop-induced cache tag lookups that can be avoided with Coarse-Grain Coherence Tracking using Region Coherence Arrays. By reducing snoop-induced cache tag lookups, power consumption and contention in the cache tag arrays can be reduced.

In Figure 4.8, the percentage of avoided snoop-induced cache tag lookups decreases with increasing region size and closely matches the potential shown in Figure 1.4. For large regions, the majority of avoided snoop-induced cache tag lookups are the result of broadcasts that were avoided by the Region Coherence Arrays in the system. However, snoop-induced cache tag lookups can be reduced an additional 10-40% by filtering external requests through the Region Coherence Array. The reduction in snoop-induced cache tag lookups appears to be independent of the reduction in broadcasts; the Region Coherence Arrays compensate for broadcasts not avoided by filtering the resultant snoop-induced cache tag lookups.

While filtering external requests through the Region Coherence Array can add latency to broadcast requests (delaying the cache tag lookup and hence the combined snoop response), this is only for broadcast requests that are satisfied by data from other processors' caches. The DRAM latency in the baseline is longer than the broadcast snoop latency, and hence latency is not added to requests that obtain data from memory.

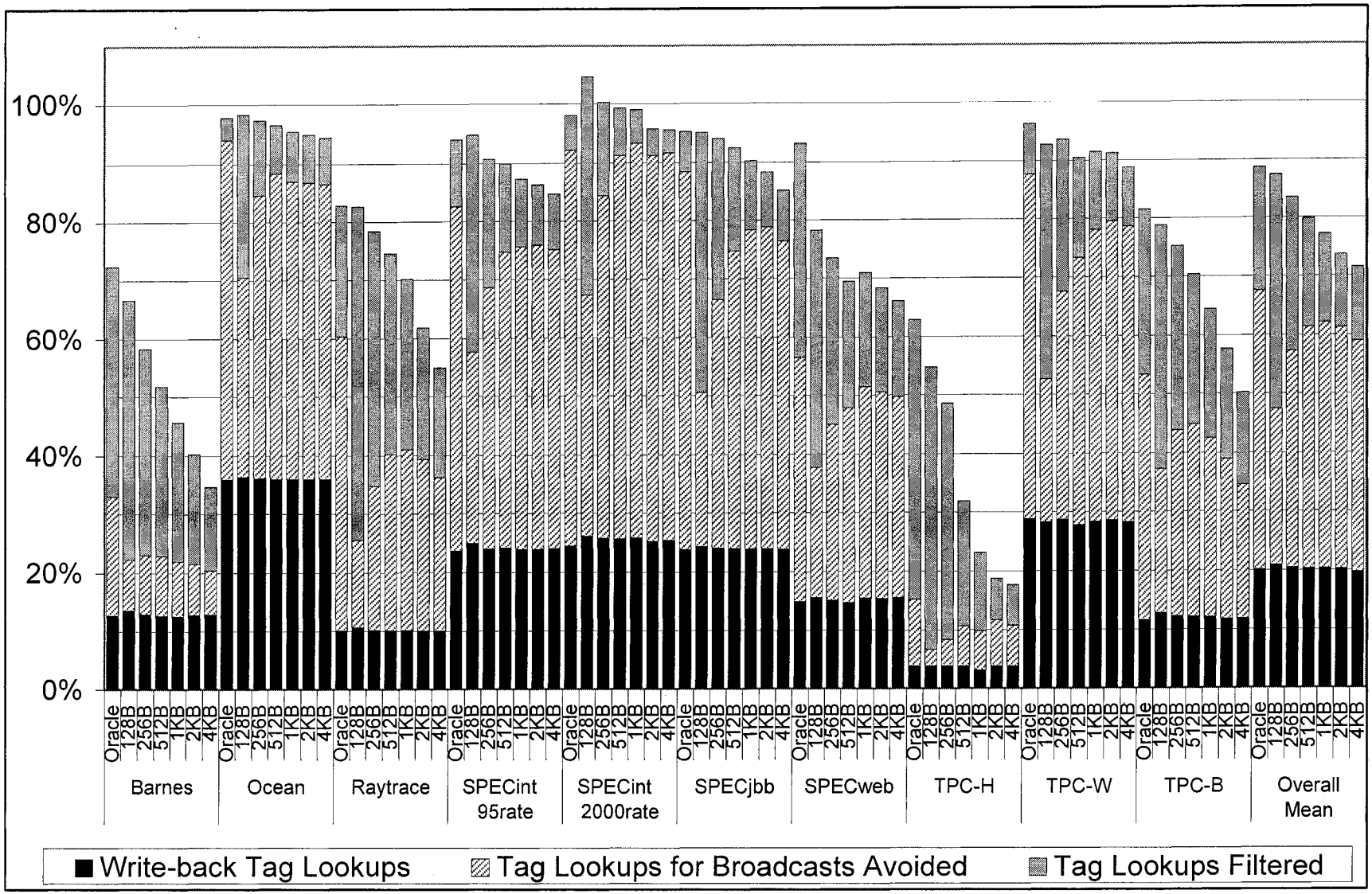

Figure 4.8: Effectiveness of CGCT for filtering snoop-induced cache-tag lookups in a four-processor system.

4.5.3 Performance Improvement

Figure 4.9 shows the reduction in execution time for Coarse-Grain Coherence Tracking with a Region Coherence Array. For each benchmark and region size, the execution time of the system with CGCT is divided by that of the baseline system. The conversion of broadcasts to direct requests reduces the average memory latency significantly, leading to execution-time reductions of 10% for region sizes of 256B and larger. The largest reduction is 28% for Ocean with 512B and larger regions. Along the same lines, Figure 4.10 shows the average speedup for Coarse-Grain Coherence Tracking with a Region Coherence Array. The speedup is computed by dividing the execution time of the baseline system by that of the system with CGCT, and subtracting 100%.

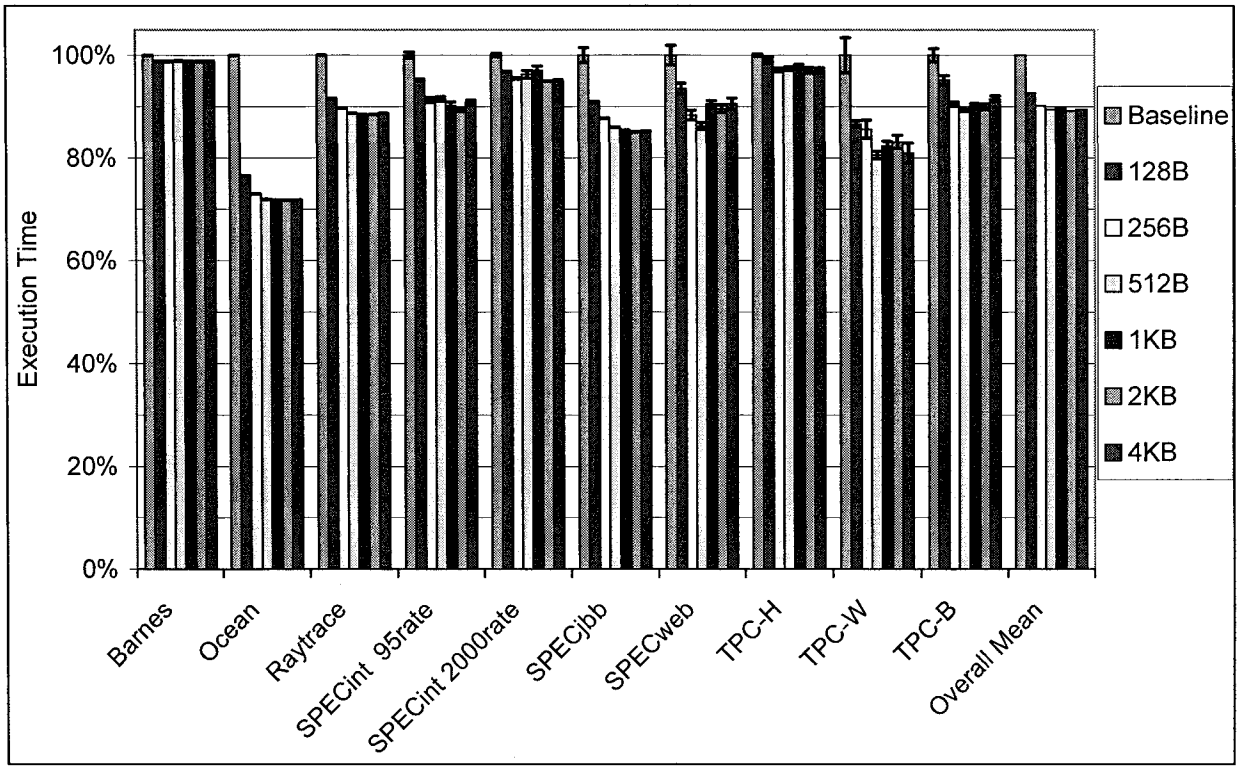

Figure 4.9: Impact of CGCT on execution time for different region sizes.

4.5.4 Scalability Improvement

By reducing the number of broadcasts, scalability is improved. In Figure 4.11, the number of broadcasts performed during the entire run of each application is divided by the number of processor cycles for both the baseline and the design with 512B regions, and is shown as the average number of broadcasts per 1,000 cycles. Figure 4.12 shows the same ratio for the peak traffic, where the peak is the largest number of broadcasts observed for any 10,000,000-cycle interval. Both the average and peak bandwidth requirements of the system have been reduced to less than half that of the baseline. Coincidentally, the workloads used here that have the highest bandwidth requirements are also those that benefit most significantly from CGCT. Also note the rate of broadcasts is lower for each benchmark despite the execution time also being shorter.

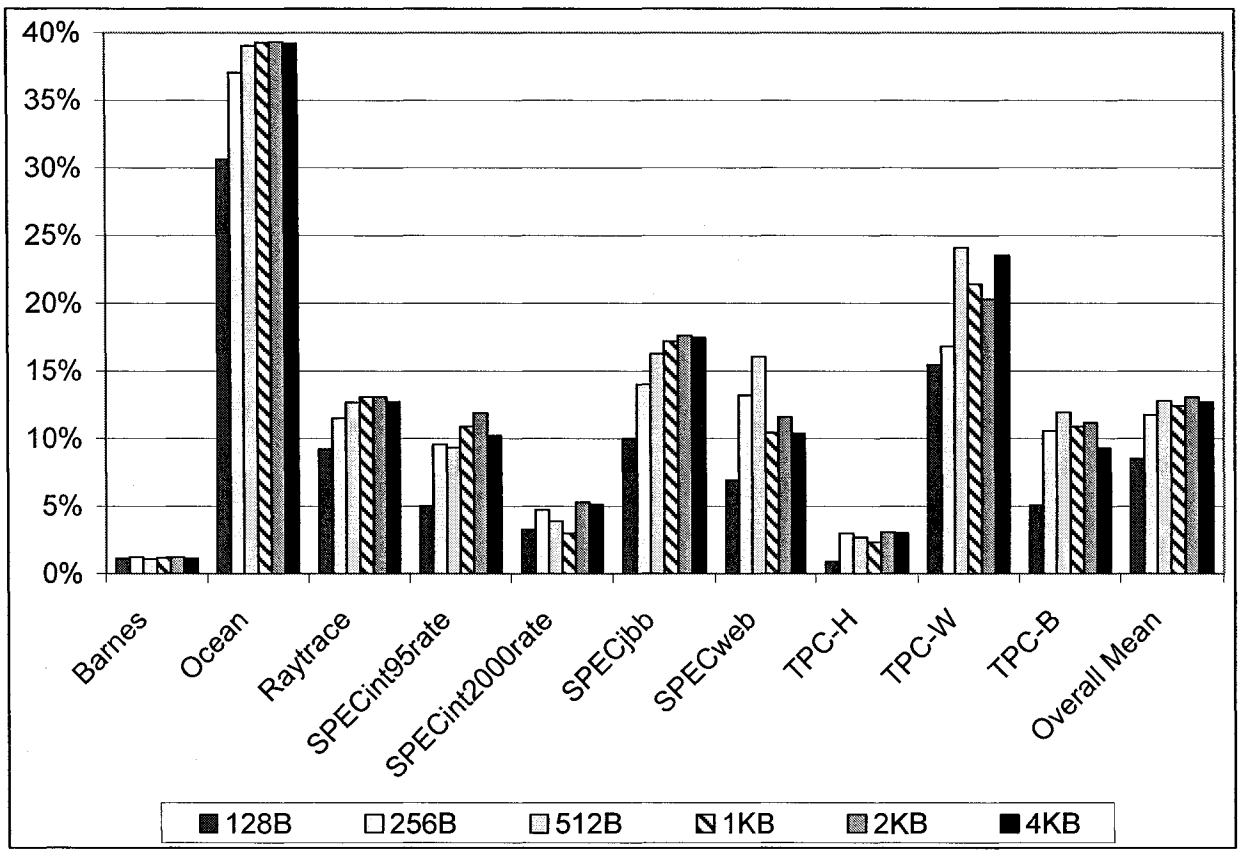

Figure 4.10: System speedup for different region sizes.

4.5.5 Performance Impact of Maintaining Inclusion

In order for Region Coherence Arrays to maintain inclusion over the caches, occasionally lines must be evicted from the cache when a region is evicted from the Region Coherence Array. To minimize the impact of these evictions, the RCA is set-associative and uses a least-recently-used (LRU) replacement policy to select regions for eviction. Unlike traditional LRU policies, this replacement policy favors regions for which no lines are cached (using the line counts to detect such regions). Figure 4.13 shows the regions evicted from the RCA broken down by the number of lines from the region that were cached (and must be evicted to maintain inclusion).

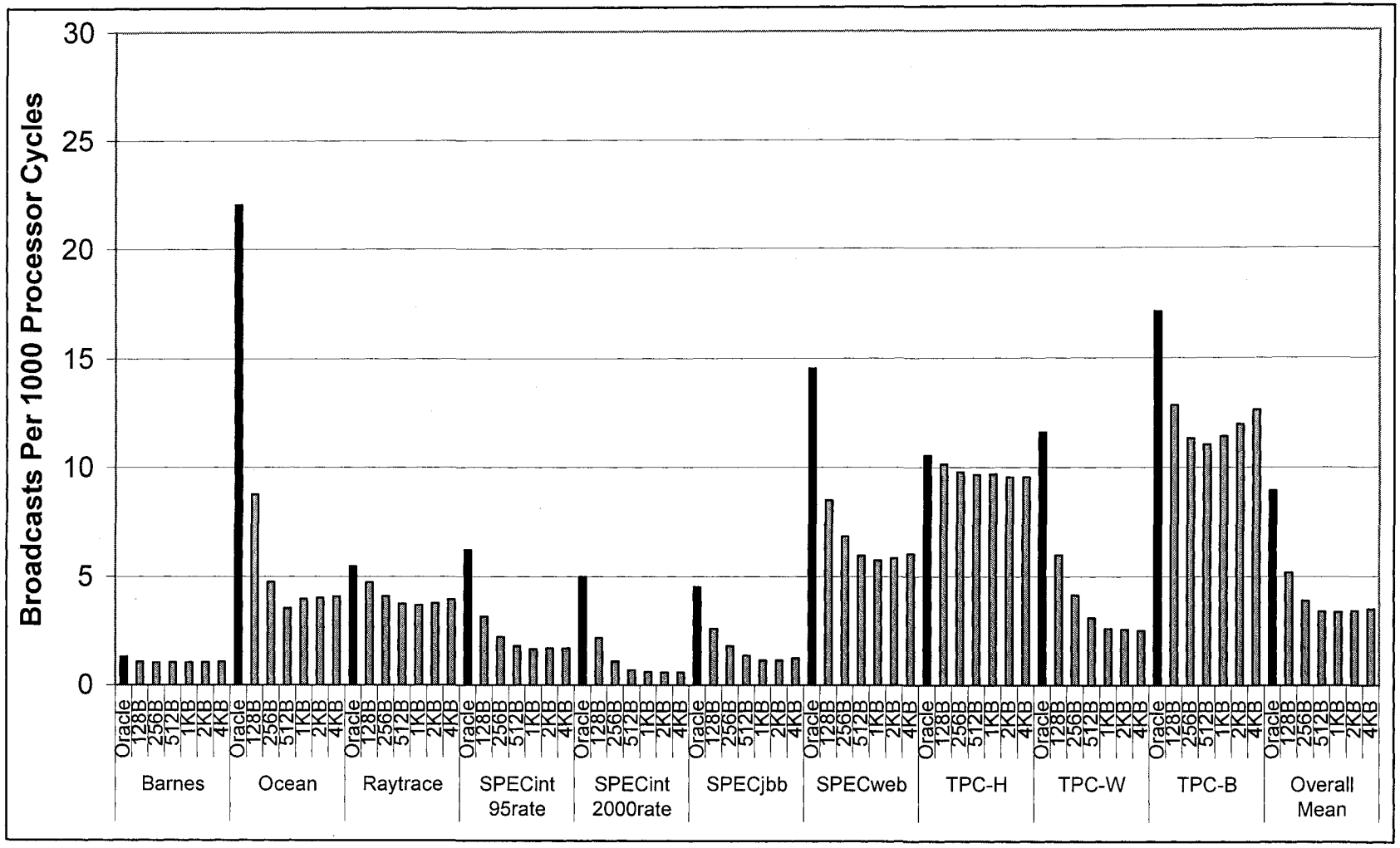

Figure 4.11: Impact on average broadcast traffic.

The average traffic for the set has gone down from 9 broadcasts per 1,000 cycles for the baseline to as low as 3.4 with Region Coherence Arrays and 1KB regions.

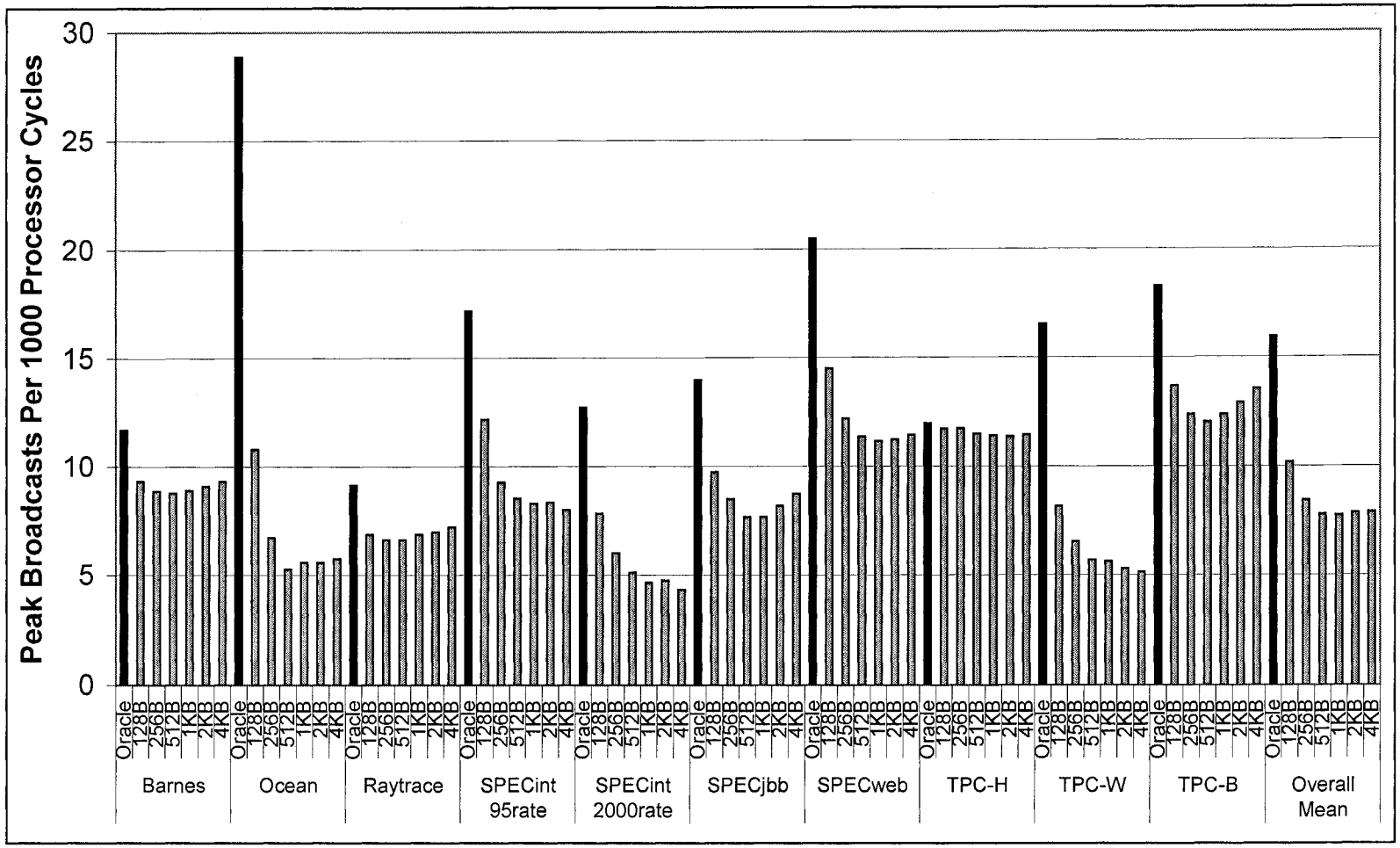

Figure 4.12: Impact on peak broadcast traffic.

The peak traffic for the set has gone down from nearly 16 broadcasts per 1,000 cycles for the baseline to 8 with Region Coherence Arrays and 512B-4KB regions. Here, the peak is the traffic for the 10,000,000-cycle interval with the most broadcasts.

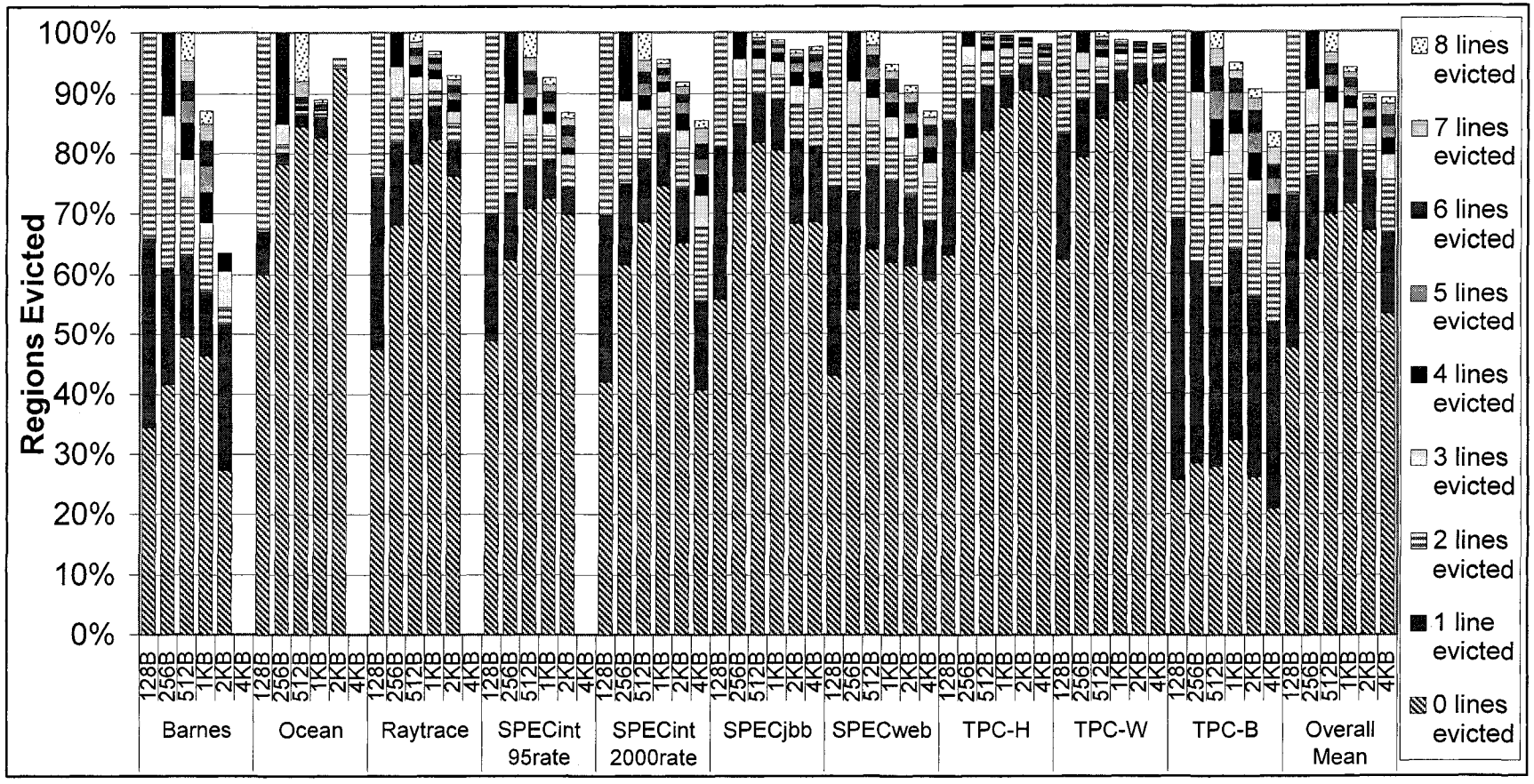

Figure 4.13: Lines evicted to maintain inclusion

Breakdown of regions evicted from the RCA by number of lines evicted from the cache to maintain inclusion. On average from 48-72% of regions evicted have zero lines cached, due to a replacement policy that favors such regions for eviction. Regions with one or fewer lines cached make up 72-80% of the regions evicted. Even for large 4KB regions with 64 lines, close to 90% of the regions evicted have 8 or fewer lines cached.

Figure 4.14 shows the L2 cache miss rate impact of these evictions. For the baseline system and a system with a Region Coherence Array with the same number of sets and associativity as the L2 cache, Figure 4.14 shows the relative L2 cache miss rate for each benchmark and region size. The average increase in L2 cache miss ratio resulting from these evictions ranges from 1% to 5.5%, decreasing with increasing region size. The larger the reach of the RCA, the less it restricts the data that may be simultaneously cached.

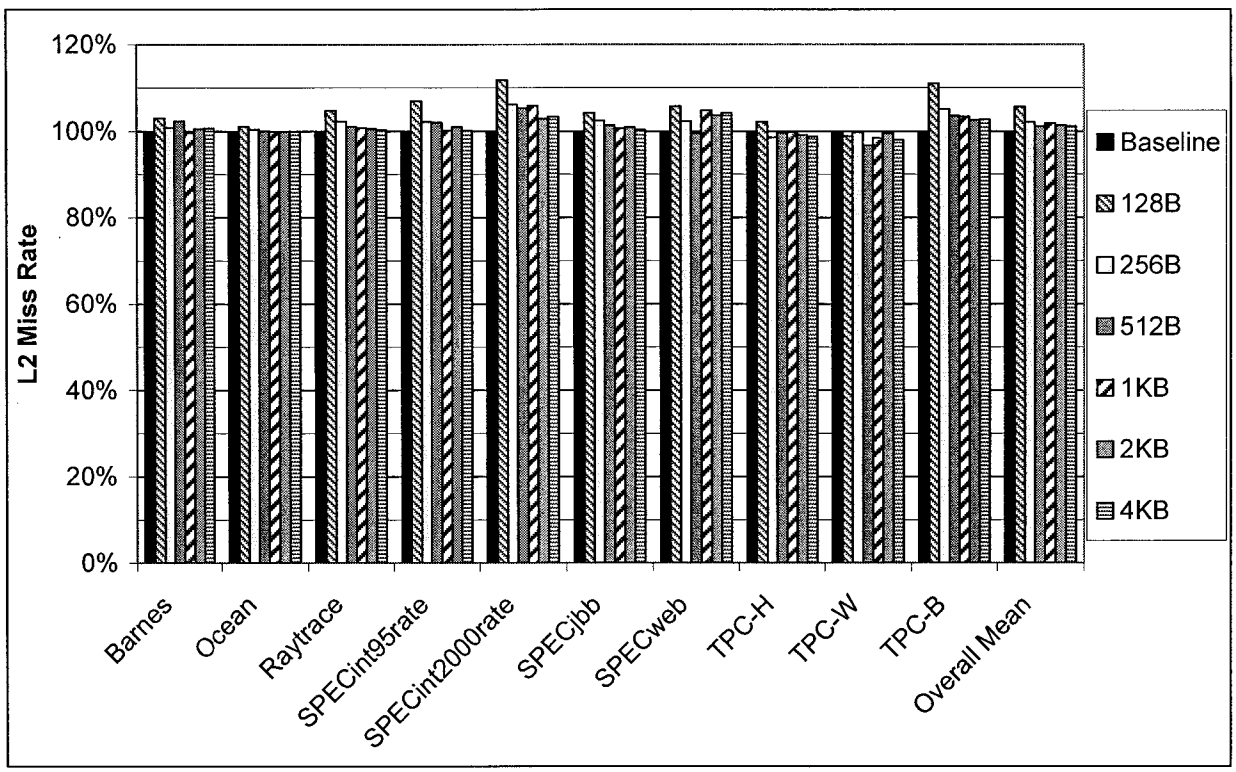

Figure 4.14: L2 cache miss ratios with and without Region Coherence Arrays. The average increase in cache miss ratio resulting from evictions to maintain inclusion ranges from 1% to 5.5%, decreasing with increasing region size. Some applications such as SPECint2000rate and TPC-B have greater than 10% miss rate increases for 128B regions (the RCA must map significantly more data than the cache to not constrain what data may be simultaneously cached).

4.6 Remaining Potential

This section briefly quantifies the remaining potential for avoiding broadcasts in systems with Region Coherence Arrays. Figure 4.15 shows the remaining potential for avoiding broadcasts. To compute the unnecessary broadcasts remaining, we took the remaining broadcasts in each execution and subtracted the unnecessary broadcasts that are used for coherence enforcement. Then for each region size we subtracted those broadcasts that could not be avoided with CGCT, i.e., the broadcast was unnecessary but other lines in the region were shared/owned. Last, we subtracted the unnecessary broadcasts that could not be avoided as a result of the region state being conservative, specifically cases in which the region state in other processors' RCAs were dirty when modifiable/modified lines were no longer cached. What remains are the broadcasts that could be avoided with perfect knowledge of the state of regions in other processors RCAs.

The largest component of the remaining potential for avoiding broadcasts is misses in the RCA, followed by broadcasts occurring when the region state is not yet known (pending). Both of these cases might be optimized by effective prefetching of the region state for a 1-25% increase in broadcasts avoided. On the other hand, very few broadcasts are the result of the region state being externally-clean or externally-dirty when in fact it is not. This is likely the result of the upgrade transitions in the region protocol (Figure 4.4) that upgrade the region state whenever possible based on the combined snoop response to broadcasts performed by the processor.

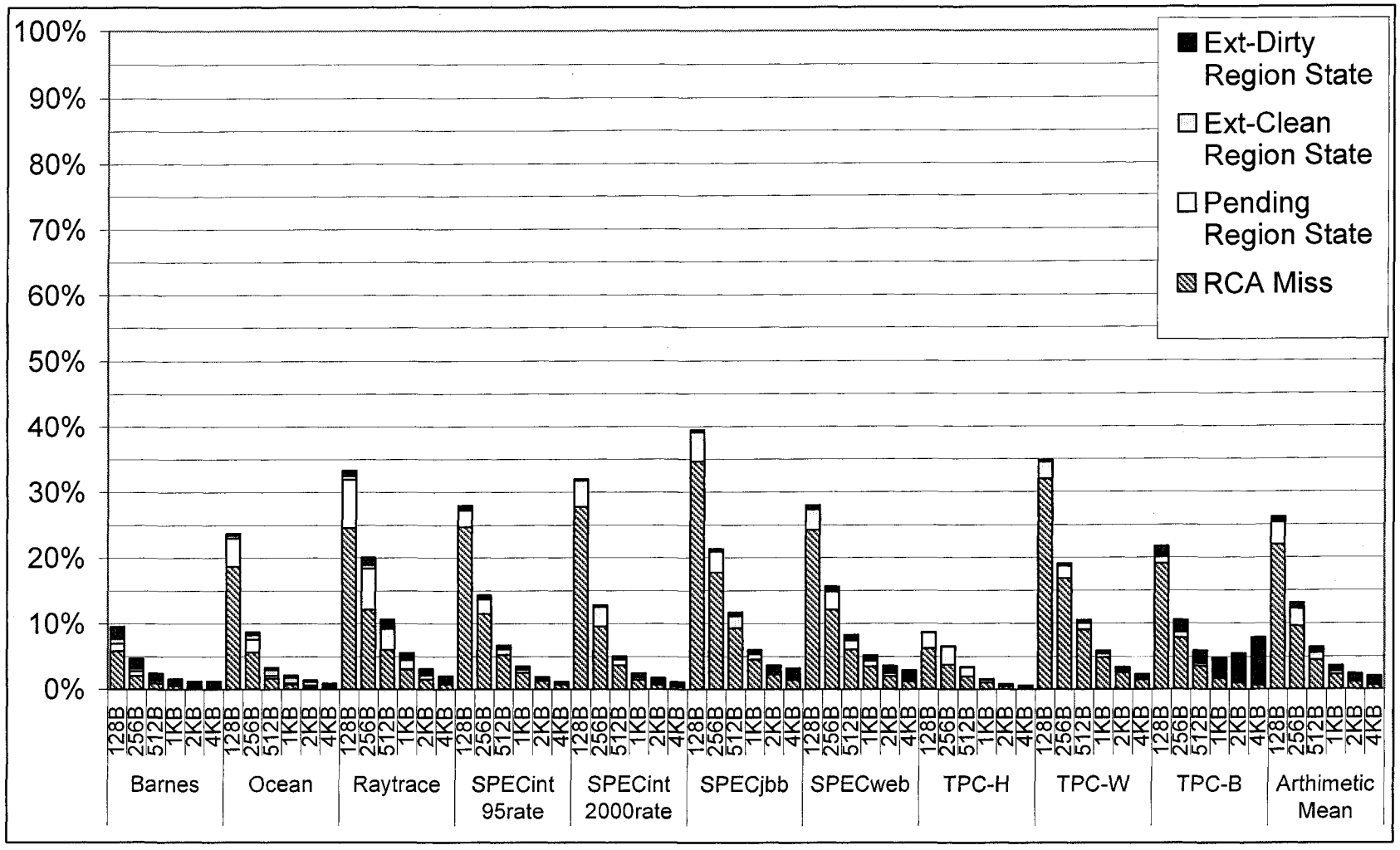

Figure 4.15: Remaining potential for avoiding broadcasts

The largest components of the remaining potential are RCA misses and broadcasts performed while in a pending region state. There is potential for prefetching the region state, especially for small region sizes.

4.7 Summary

This chapter proposed and investigated Region Coherence Arrays, an effective and low cost implementation of Coarse-Grain Coherence Tracking. The implementation of Region Coherence Arrays was discussed in detail, with a region protocol, hardware overheads, and baseline system modifications. Results for a set of commercial, scientific, and multiprogrammed workloads show that Region Coherence Arrays can avoid most of the unnecessary broadcasts and filter most of the unnecessary snoop-induced cache tag lookups in a broadcast-based multiprocessor system. Large reductions in the average and peak broadcast traffic were observed, leading to improved scalability for the system. Finally, we measured the impact of maintaining inclusion over the caches and characterized the remaining potential for Region Coherence Arrays.

5. RegionScout Filters vs. Region Coherence Arrays

This chapter describes RegionScout Filters, an alternative implementation of CGCT proposed concurrently by Andreas Moshovos [5, 33], and compares them to Region Coherence Arrays. RegionScout Filters target the common case of data requests to regions for which none of the lines are cached by other processors. They employ non-tagged hash tables to efficiently track regions from which the processor is caching lines and use a small tagged array to buffer the external status of regions recently accessed by the processor. They are space and power efficient and simple to implement, but they are less effective at avoiding broadcasts.

Section 5.1 describes the structure and operation of a RegionScout Filter. Section 5.2 qualitatively compares and contrasts RegionScout Filters and Region Coherence Arrays. This is followed by a quantitative comparison with simulation results for the same baseline system and workloads in Section 5.3. Section 5.4 discusses the two implementations and comments on ideas from both that might be combined beneficially. Finally, Section 5.5 summarizes and concludes the chapter.

5.1 RegionScout Filters

A RegionScout Filter consists of two structures located with each processor: a Cached-Region Hash (CRH) for tracking regions from which the processor is caching lines and a Nonshared-Region Table (NSRT) for buffering the external coherence status of regions recently accessed by the processor. The CRH is a non-tagged hash table indexed by region address, each entry containing a count for the lines cached from regions mapping to that entry. Optionally, the CRH

may have a bit for each entry that is set when the count is nonzero; for power efficiency this bit can be checked instead of reading the whole count and comparing it to zero. The NSRT is a small tagged array, each entry containing an address tag for the region and a valid bit.

When a processor's cache allocates or evicts a line, the CRH is indexed by the address of the surrounding region, and the count is incremented or decremented, respectively. The CRH is non-tagged, so if lines are cached from multiple regions mapping to the same CRH entry, the count is the sum of the lines cached from all regions mapping to that entry. Hence, if the CRH count indexed by a region address is nonzero, there *may* be lines from the region cached by the processor, and, conversely, there are no cached lines from *any* matching region if the count is zero. When a processor broadcasts a memory request, the other processors' CRHs are snooped to determine if they may be caching lines from the region. The region snoop response generated by each CRH consists of a single bit indicating whether the count is zero or nonzero, and the combined snoop response is the logical sum of these bits. If the region snoop response indicates that no other processors are caching lines from the requested region, an entry for the region is allocated in the NSRT of the requesting processor. Entries in the NSRT are allocated only when a region is not shared by other processors. The broadcast also invalidates any matching entries in other processors' NSRTs for correctness

After detecting an L2 cache miss, a processor checks its NSRT for the region. If there is a valid entry for the region, a broadcast is not needed and the request is sent directly to memory. If not, a broadcast must be performed to ensure coherence is enforced.

Figure 5.1 shows an example of a RegionScout Filter in operation. In part (a), processor A requests line x in region X (step 1). After checking its NSRT and finding no matching entry for

region X, processor A broadcasts the request. All remote processors probe their CRHs and report that they do not cache any line in region X (step 2). Processor A records region X as nonshared and increments the corresponding CRH entry (step 3). In part (b), processor A is about to request line y in region X and first checks its NSRT (step 1). Because a valid entry is found, it sends the request directly to memory (step 2). In part (c), processor B requests line z in region X. It first checks its NSRT (step 1). The region is not recorded in its NSRT, so processor B broadcasts its request (step 2). Processor A invalidates its NSRT entry because now region X is shared (step 3).

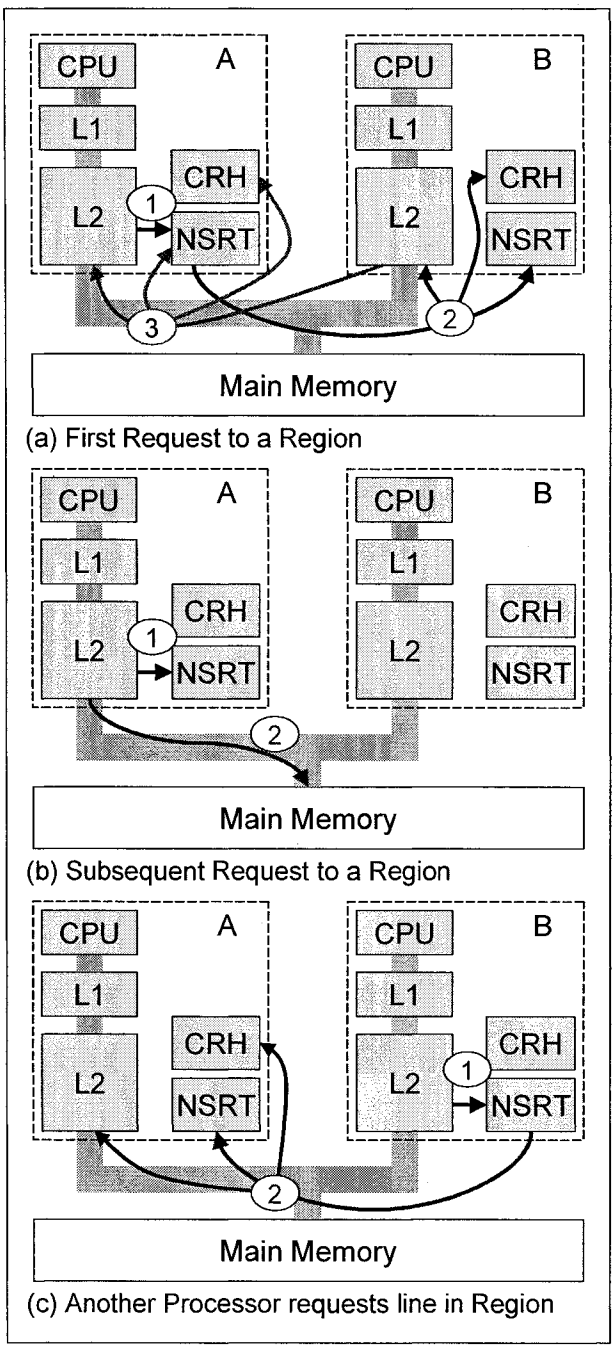

Figure 5.1: RegionScout filter example.

First request for a line in a region (a); subsequent request for a line in the same region (b); another processor requests a line in the region (c). RegionScout Filter discovers a nonshared region (a), avoids subsequent broadcasts (b), and later determines that the region has become shared (c).

5.2 RegionScout Filters vs. Region Coherence Arrays

RegionScout Filters are more space-efficient and power-efficient than Region Coherence Arrays with comparable numbers of entries. In addition, RegionScout Filters are less complex and have a lower impact on the design of a multiprocessor memory system. However, Region Coherence Arrays have four performance advantages over RegionScout Filters.

5.2.1 Power-Efficiency

RegionScout Filters should be more power-efficient than comparably sized Region Coherence Arrays. However, a quantitative comparison of the power consumption of the baseline and systems with Region Coherence Arrays and RegionScout Filters is beyond the scope of this dissertation.

RegionScout Filters are designed for low-power consumption. The RegionScout Filter is only accessed by processor requests after a cache miss is detected. The processor requests only access the small NSRT. External snoop requests access only the CRH, which is a power-efficient non-tagged hash table. There is no associative search, and the two structures can be sized independently to balance power consumption with performance.

Region Coherence Arrays are accessed in parallel with the lower-level cache so that the region state is available on a cache miss. They are also accessed by every external request, essentially handling the same request stream of the cache tag arrays in the baseline system. This is offset somewhat by the decreased number of broadcasts and snoop-induced cache tag lookups.

5.2.2 Space-Efficiency

Assuming 48-bit physical addresses, a 512-Kbyte, 2-way set-associative cache with 64-byte lines requires a 30-bit tag. Assume also that the system allows 16 outstanding requests per processor. A RegionScout CRH indexed by the low address bits needs a maximum count per entry equal to the number of lines per region multiplied by the cache associativity, added to the maximum number of outstanding requests. For 4KB regions, this is an 8-bit counter. With a parity bit, a total of 9 bits per entry is necessary. A RegionScout NSRT entry contains an address tag, a valid bit, a parity bit, and least-recently-used information. For a RegionScout filter with 4KB regions, 8K-entry CRH entries, and a 64-entry, 4-way set-associative NSRT, the storage overhead is about 9.3 Kbytes. The storage overhead drops to 5.1KB for small 128B regions. The cache and cache tag storage overheads of RegionScout CRHs and NSRTs for varying region sizes and structure sizes are shown in Tables 5.1 and 5.2, respectively.

Table 5.1: Storage for RegionScout CRH with varying entries, region sizes.

	Count	Count Nonzero	Parity	Bits / Set	Total Kilobytes	Tag Space Overhead	Cache Space Overhead
2K-Entry CRH, 128B Regions	5	1	1	7	1.8	4.5%	0.3%
2K-Entry CRH, 256B Regions	5	1	1	7	1.8	4.5%	0.3%
2K-Entry CRH, 512B Regions	6	1	1	8	2.0	5.2%	0.3%
2K-Entry CRH, 1KB Regions	6	1	1	8	2.0	5.2%	0.3%
2K-Entry CRH, 2KB Regions	7	1	1	9	2.3	5.8%	0.4%
2K-Entry CRH, 4KB Regions	8	1	1	10	2.5	6.5%	0.4%
4K-Entry CRH, 128B Regions	5	1	1	7	3.5	9.1%	0.6%
4K-Entry CRH, 256B Regions	5	1	1	7	3.5	9.1%	0.6%
4K-Entry CRH, 512B Regions	6	1	1	8	4.0	10.4%	0.7%
4K-Entry CRH, 1KB Regions	6	1	1	8	4.0	10.4%	0.7%
4K-Entry CRH, 2KB Regions	7	1	1	9	4.5	11.7%	0.7%
4K-Entry CRH, 4KB Regions	8	1	1	10	5.0	13.0%	0.8%
8K-Entry CRH, 128B Regions	5	1	1	7	7.0	18.2%	1.1%
8K-Entry CRH, 256B Regions	5	1	1	7	7.0	18.2%	1.1%
8K-Entry CRH, 512B Regions	6	1	1	8	8.0	20.8%	1.3%
8K-Entry CRH, 1KB Regions	6	1	1	8	8.0	20.8%	1.3%
8K-Entry CRH, 2KB Regions	7	1	1	9	9.0	23.4%	1.5%
8K-Entry CRH, 4KB Regions	8	1	1	10	10.0	26.0%	1.6%
16K-Entry CRH, 128-Byte Regions	5	1	1	7	14.0	36.4%	2.3%
16K-Entry CRH, 256-Byte Regions	5	1	1	7	14.0	36.4%	2.3%
16K-Entry CRH, 512-Byte Regions	6	1	1	8	16.0	41.6%	2.6%
16K-Entry CRH, 1024-Byte Regions	6	1	1	8	16.0	41.6%	2.6%
16K-Entry CRH, 2048-Byte Regions	7	1	1	9	18.0	46.8%	2.9%
16K-Entry CRH, 4096-Byte Regions	8	1	1	10	20.0	51.9%	3.3%
32K-Entry CRH, 128-Byte Regions	5	1	1	7	28.0	72.7%	4.6%
32K-Entry CRH, 256-Byte Regions	5	1	1	7	28.0	72.7%	4.6%
32K-Entry CRH, 512-Byte Regions	6	1	1	8	32.0	83.1%	5.2%
32K-Entry CRH, 1024-Byte Regions	6	1	1	8	32.0	83.1%	5.2%
32K-Entry CRH, 2048-Byte Regions	7	1	1	9	36.0	93.5%	5.9%
32K-Entry CRH, 4096-Byte Regions	8	1	1	10	40.0	103.9%	6.5%

Table 5.2: Storage for RegionScout NSRT with 64 entries, varying region sizes.

	Tag (4)	Valid (4)	Parity (4)	LRU	Bits / Set	Total Kilobytes	Tag Space Overhead	Cache Space Overhead
64-Entry, 4-way assoc, 128B Regions	37	1	1	3	159	0.3	0.81%	0.1%
64-Entry, 4-way assoc, 256B Regions	36	1	1	3	155	0.3	0.79%	0.0%
64-Entry, 4-way assoc, 512B Regions	35	1	1	3	151	0.3	0.77%	0.0%
64-Entry, 4-way assoc, 1KB Regions	34	1	1	3	147	0.3	0.75%	0.0%
64-Entry, 4-way assoc, 2KB Regions	33	1	1	3	143	0.3	0.73%	0.0%
64-Entry, 4-way assoc, 4KB Regions	32	1	1	3	139	0.3	0.71%	0.0%

An RCA, on the other hand, needs more storage for the same number of entries. RCA entries need an address tag, 3 state bits, a line count, a parity bit, and least-recently-used information. From Table 4.2, a 4K-set, 2-way set-associative structure, storage is 70 bits per set, or 35 Kbytes total, irrespective of region size. Depending on the region size, an RCA can require 3.75-8 times as much storage as a RegionScout filter for the same number of entries.

5.2.3 Impact on System Design

Both Region Coherence Arrays and RegionScout Filters require minor modifications to a multiprocessor memory system, such as additional bits in the combined snoop response. Other modifications needed are specific to each implementation:

Region Coherence Arrays need the ability to evict cache lines to maintain inclusion. This is not the case for RegionScout Filters, which maintain inclusion via a non-tagged hash table. In order to do this, the Region Coherence Array needs a communication path back to the cache(s) with which it can send eviction requests and mechanisms to evict lines in a manner ordered with respect to requests from the processor and other processors.

In some systems RegionScout Filters require address decoders to optimize write-back requests. While write-backs trivially do not require a broadcast for maintaining coherence, they are commonly broadcast to locate the appropriate memory controller and simplify ordering. To send write-backs directly to memory, firmware-programmed address decoders are needed to determine the correct memory controller to send the request to (which is nontrivial with the complex multitude of DRAM configurations, interleaving modes, and slot occupancies in a commercial system [30]). Conversely, Region Coherence Arrays can store a memory controller

index with each entry, and this index is always available for write-backs because the Region Coherence Array maintains inclusion over the cache.

5.2.4 Performance

Region Coherence Arrays have four performance advantages over RegionScout Filters, including optimizing instruction fetches, exploiting temporal locality, precise tracking of regions cached by the processor, and no additional latency in sending requests.

First, RegionScout Filters target only requests to non-shared data and do not track data that is clean-shared to optimize instruction fetches (often processors share instructions, and multiprocessor systems commonly do not source clean copies of lines to other processors). Region Coherence Arrays have externally-clean region states that track regions for which lines are shared, but have not been modified by other processors. Instruction fetches do not require a broadcast if the instructions have not been modified; the data in memory is up to date, and instruction fetches do not take shared copies of lines away from other processors.

Second, RegionScout Filters only exploit spatial locality, whereas Region Coherence Arrays can exploit spatial and temporal locality. In other words, RegionScout Filters can only avoid broadcasts for requests to other lines in a region recently accessed by the processor (spatial locality), and cannot optimize requests for a line that was previously cached but evicted before it could be used again (temporal locality). The RegionScout NSRT is small to minimize power consumption and the latency added to snoops. To exploit temporal locality the NSRT must be large enough to map more data than the cache, and regions must remain resident in the NSRT long enough for lines to be evicted and brought into the cache again. Region Coherence Arrays

are accessed in parallel with the low-level cache, and can map two or more times the data in the cache. As a result, a significant portion of the broadcasts avoided are the result of temporal locality.

Third, Region Coherence Arrays precisely track regions cached by the processor, providing a region snoop response that precisely indicates whether other processors are caching lines in the region. Because Region Coherence Arrays are tagged associative structures, information about regions does not alias with information from other regions, allowing them to avoid more broadcasts and snoop-induced cache tag lookups. The performance is not adversely affected if multiple regions map to the same index, provided the Region Coherence Array has sufficient associativity.

Finally, Region Coherence Arrays do not delay sending requests externally. Region Coherence Arrays are accessed in parallel with the cache access so the region state is available on a cache miss. Whether the request must be broadcast or can be sent directly to memory, the request can be sent externally right away. RegionScout Filters, on the other hand, delay sending the external request. To save power the NSRT is not accessed until after a cache miss is detected, at the cost of additional latency for external requests.

5.3 Simulation Results Comparing RegionScout and Region Coherence Arrays

In this section, RegionScout Filters and Region Coherence Arrays are compared quantitatively with simulation results for the same baseline system and workloads. The two implementations are evaluated based on their ability to avoid broadcasts and filter snoop-induced cache tag lookups.

5.3.1 Avoiding Broadcasts and Reducing Broadcast Traffic

Figure 5.2 shows the average percentage of requests sent directly to memory or avoided altogether by RegionScout Filters and Region Coherence Arrays, for a baseline system with 512KB 2-way set-associative L2 caches (Note that the caches were set to 512KB to correlate with earlier work done by Moshovos [5, 33], refer to Chapter 3 for other parameters). The RegionScout Filters have 64-entry, 4-way set-associative NSRTs, and CRHs with varying numbers of entries indexed by the lower address bits. The Region Coherence Arrays are 2-way set-associative with varying numbers of sets. The region size is varied from 128 bytes (two 64-byte cache lines) to 4KB (the physical page size). Figure 5.2 also shows the broadcasts that can be avoided by a theoretically optimal implementation of CGCT, which uses oracle knowledge of the status of regions in other processors' caches to avoid as many broadcasts as possible for that region size.

The RegionScout Filter and Region Coherence Array both substantially reduce the number of broadcasts. However, the Region Coherence Array always outperforms the RegionScout Filter for the same region size. A Region Coherence Array with as few as 1,024 entries outperforms a RegionScout Filter with a 32,768-entry CRH. The Region Coherence Array approaches the oracle line as the structure size and region size increase, whereas the RegionScout Filter appears to approach 55% asymptotically as the CRH size and region size increase.

As the Region Coherence Array's number of sets and region size increase, the amount of spatial and temporal locality it can exploit also increases. The larger region size yields more spatial and temporal locality, with potential lost only for the first access to the region. The larger

number of sets increases the RCA's reach beyond the cache and increases the average lifetime of a region in the RCA. As the region size increases to 1KB and beyond, the RCA's effectiveness begins to drop (along with the theoretically optimal implementation's effectiveness), due to increased false-sharing of regions.

As the RegionScout Filter's number of CRH entries and region size increase there are fewer collisions in the hash table and more spatial locality is exploited. However the limiting factor appears to be the reach of the NSRT. The NSRT is small by design to minimize power consumption and latency overhead; it cannot buffer enough regions to exploit all of the available spatial locality, nor any temporal locality beyond the cache.

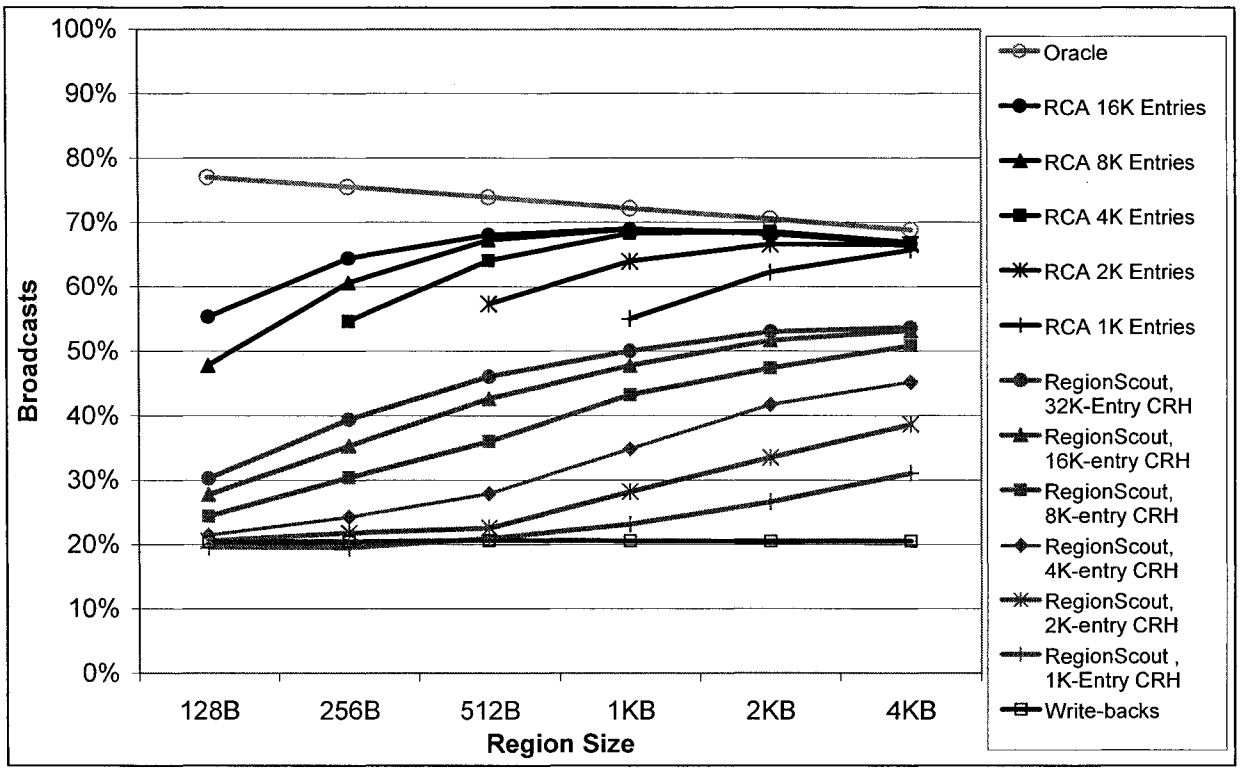

Figure 5.2: Broadcasts avoided by RegionScout filters and Region Coherence Arrays.

The baseline system has 512KB 2-way set-associative L2 caches (Refer to Chapter 3 for other parameters). The RegionScout Filters have 64-entry, 4-way set-associative NSRTs, and CRHs with varying numbers of entries indexed by the lower address bits. The Region Coherence Arrays are 2-way set-associative. For the same region size, the Region Coherence Array consistently outperforms the RegionScout Filter.

Figure 5.3 shows the same data as Figure 5.2, except that now the x-axis shows the logarithm of the number of kilobytes of storage used for each implementation, and there is a curve for each region size instead of for each number of entries. The y-axis is still the percentage of broadcasts sent directly to memory or avoided altogether by the two Coarse-Grain Coherence Tracking implementations. From this graph, we can compare the two techniques based on the amount of

storage they use, and for a given amount of storage select the best region size and implementation.

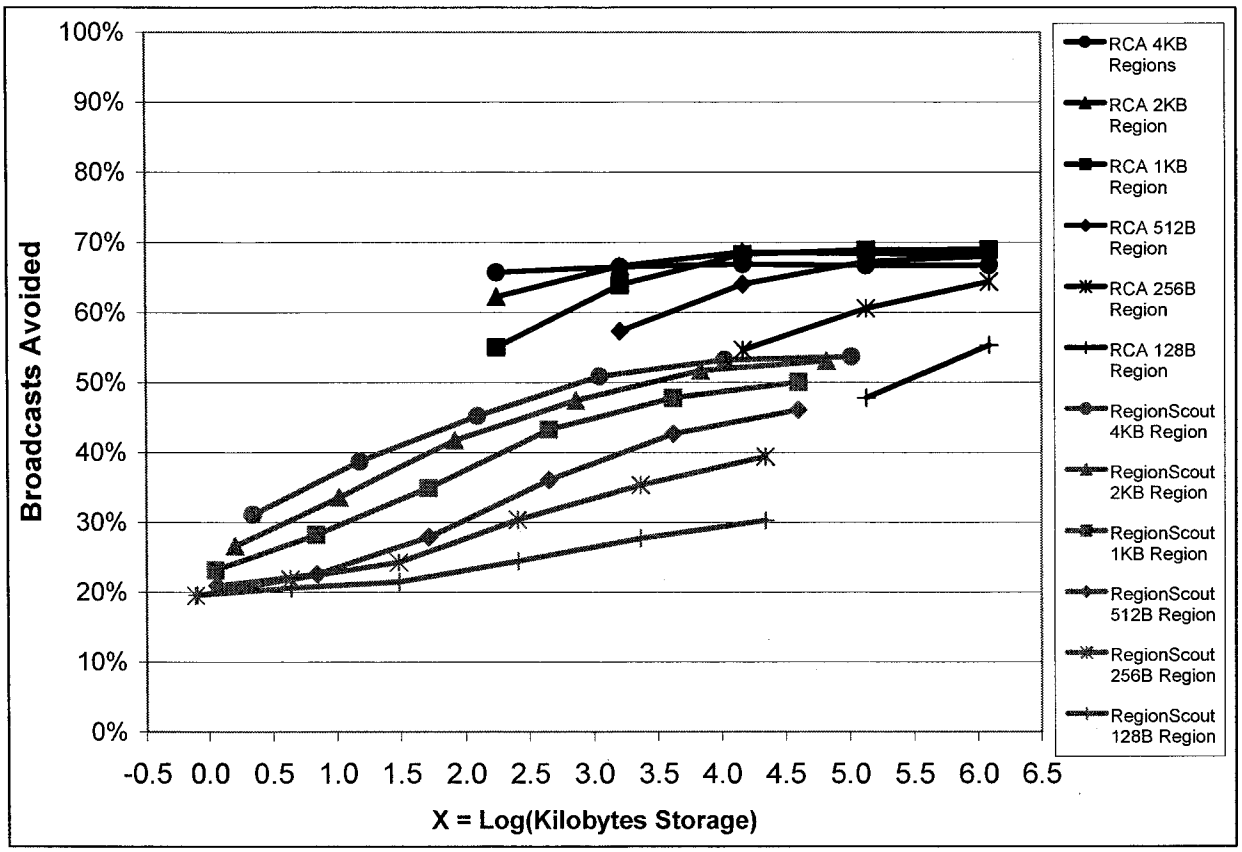

Figure 5.3: Broadcasts avoided by RegionScout filters and Region Coherence Arrays per kilobyte storage.

The RegionScout Filter scales down to less than 1KB while still avoiding broadcasts. However, for equal amounts of storage, the Region Coherence Array outperforms the RegionScout Filter for the same region size. As the amount of storage is scaled down, both structures perform best with large region sizes.

Here, we can clearly see that for equal amounts of storage and large region sizes Region Coherence Arrays consistently outperform RegionScout Filters. Region Coherence Arrays also consistently outperform RegionScout Filters when comparing data points with the same amount

of storage and the same region size. However, the RegionScout Filters have the virtue of scaling down to smaller amounts of storage, providing a reduction in broadcasts for as little as 2-4KB storage.

Figures 5.4 and 5.5 show the average and peak broadcast traffic, respectively. The average broadcast traffic is measured as the number of broadcasts performed during execution of the program divided by the number of processor cycles the program executed. The peak traffic is measured as the maximum number of broadcasts performed during any 10-million cycle interval, divided by the number of cycles. For clarity, these measurements are converted to units of broadcasts performed per 1,000 cycles executed.

RegionScout Filters reduce the average broadcast traffic from nearly 12 broadcasts per 1000 cycles to 6-9 broadcasts per 1000 cycles, reducing the average traffic by more than half in the best case. The Region Coherence Array reduces the broadcast traffic more dramatically, from nearly 12 broadcasts per 1000 cycles to 3-6 broadcasts per 1000 cycles, a nearly 75% reduction.

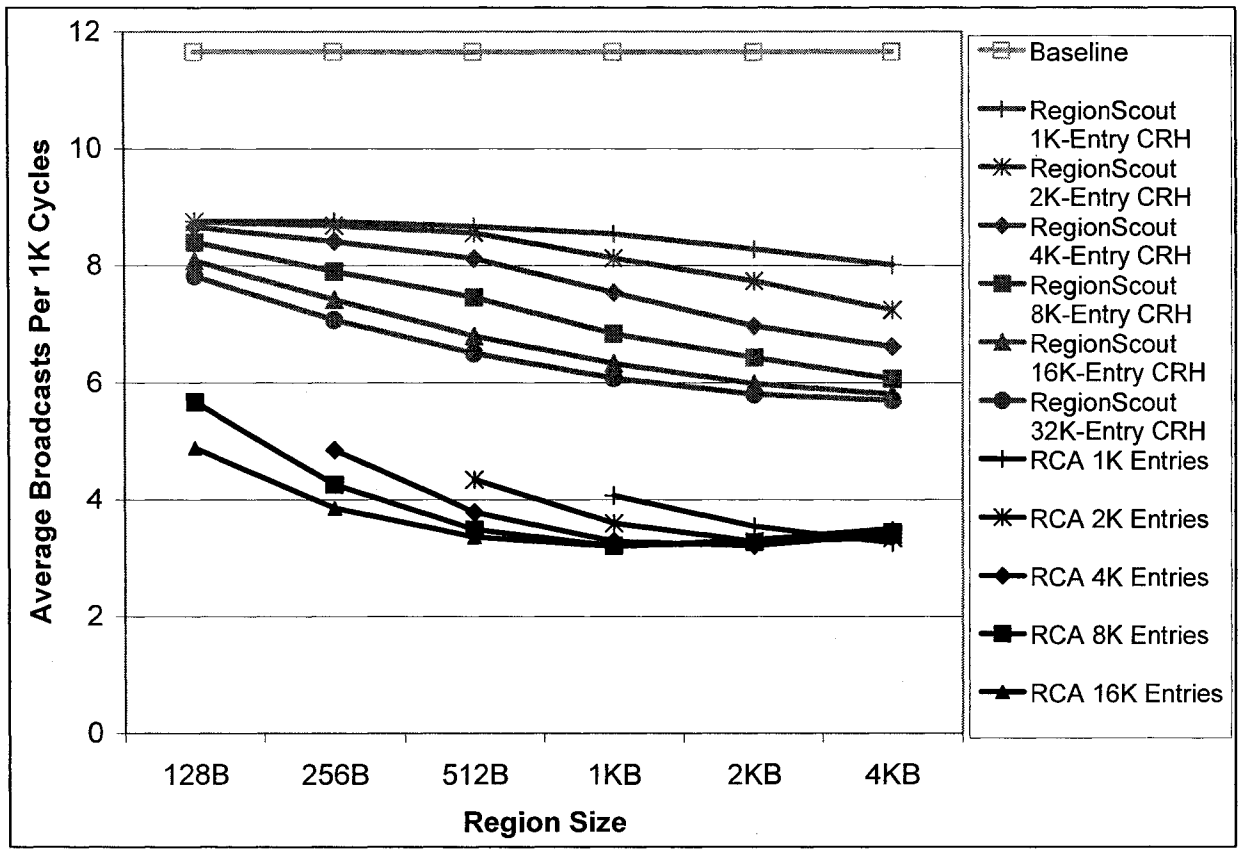

Figure 5.4: Average broadcast traffic comparison.

RegionScout Filters reduce the peak broadcast traffic from nearly 19 broadcasts per 1000 cycles to 10-15 broadcasts per 1000 cycles, reducing the peak traffic by nearly half for large regions and CRHs. The Region Coherence Array reduces the broadcast traffic more dramatically, from nearly 19 broadcasts per 1000 cycles to 7-10 broadcasts per 1000 cycles, an overall reduction of nearly 65%. Region Coherence Arrays can extend broadcast-based systems further by decreasing average and peak broadcast traffic more.

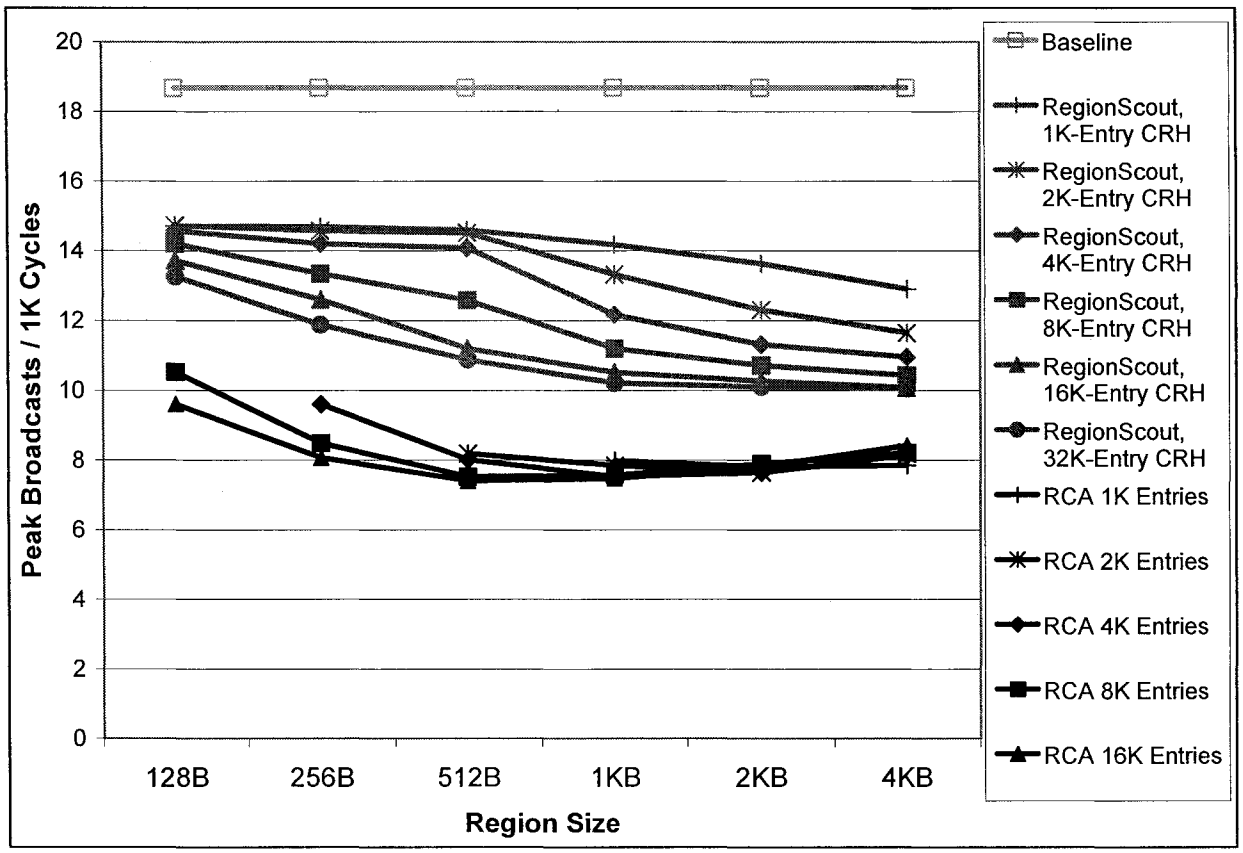

Figure 5.5: Peak broadcast traffic comparison.

5.3.2 Filtering Snoop-Induced Cache Tag Lookups

Figure 5.6 shows the percentage of snoop-induced cache tag lookups from the baseline system eliminated by the two CGCT implementations. For the most part, the reduction in snoop-induced cache tag lookups is a direct result of the reduction in broadcasts. Nevertheless, the CGCT implementations filter additional snoop-induced cache tag lookups.

The RegionScout Filter increases in effectiveness with region size up to an 8K-entry CRH. As the CRH grows in number of entries and region size, there are fewer collisions, and more regions are correctly identified as not cached by the processor. At CRH sizes of 8K-entries and above, the increased reach of a larger region size begins to be offset by the increased probability

of caching a line in the region. The larger the region, the more likely the processor is caching some line in the region, and the lower the chance that an external snoop for a line in that region will be filtered.

At 8K entries, the Region Coherence Array filters more snoop-induced cache tag lookups than all the RegionScout Filter configurations. The reason for this is the precise nature of the information in the Region Coherence Array; no region is identified as cached by the processor unless lines are in fact cached. As the Region Coherence Array is reduced in size, the percentage of snoop-induced cache tag lookups filtered is relatively constant for a given region size. Though the smaller RCA is avoiding fewer broadcasts, it is able to compensate for this by avoiding cache tag lookups for the resultant broadcasts. The effectiveness only drops slightly due to the increasing region size and increased probability of caching lines in the larger region.

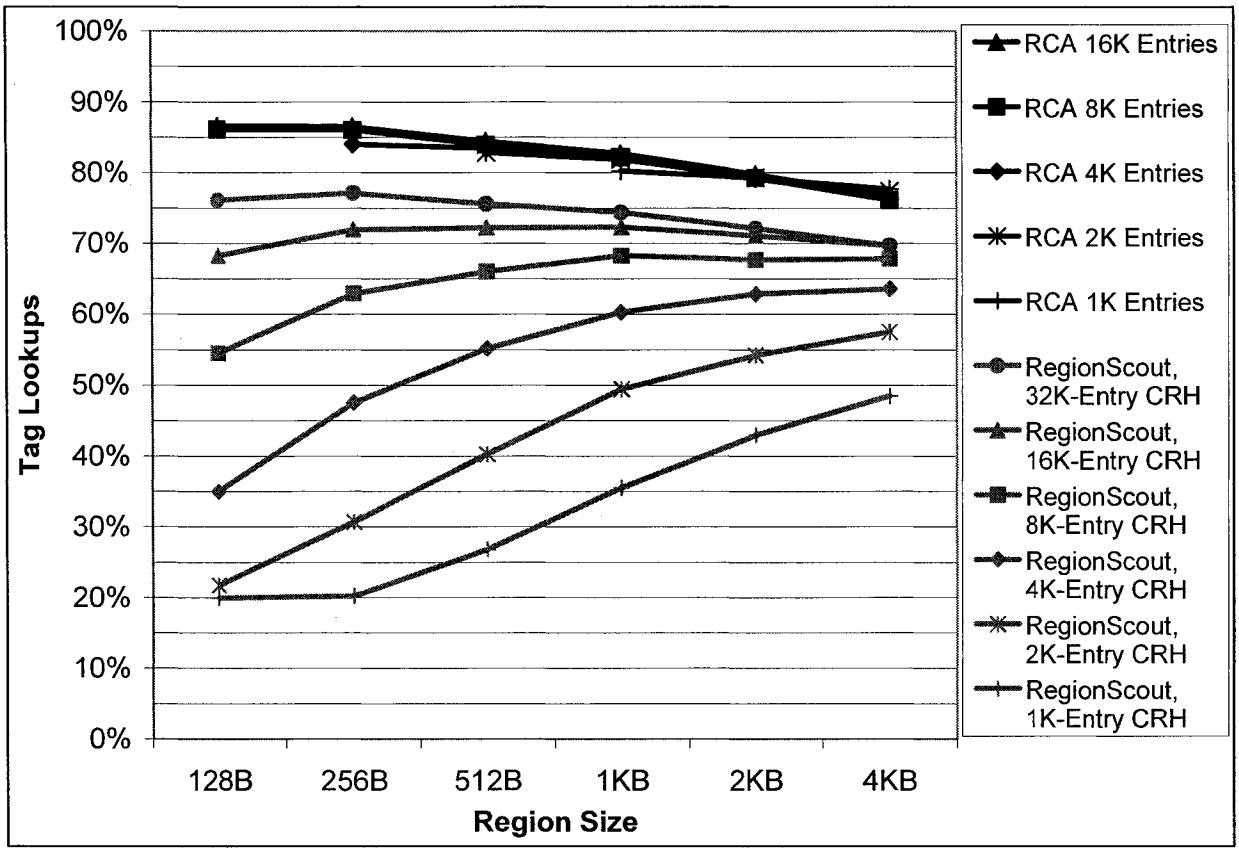

Figure 5.6: Snoop-induced cache tag lookup filtering comparison

A shortcoming of Figure 5.6 is that it only shows the snoop-induced cache tag lookups that are filtered by the two CGCT implementations. One must take into account that the Region Coherence Arrays evict lines from the caches to maintain inclusion, and the cache tags must be looked up for lines to evict. Figure 5.7 shows the net reduction in snoop-induced cache tag lookups, where cache tag lookups are added for lines evicted by the Region Coherence Array for inclusion. For 1,024 entries, the Region Coherence Array has a net effect of filtering only 55-65% of the snoop-induced cache tag lookups. While this is still a significant reduction in cache tag lookups, it is a significant drop in performance over the larger Region Coherence Arrays.

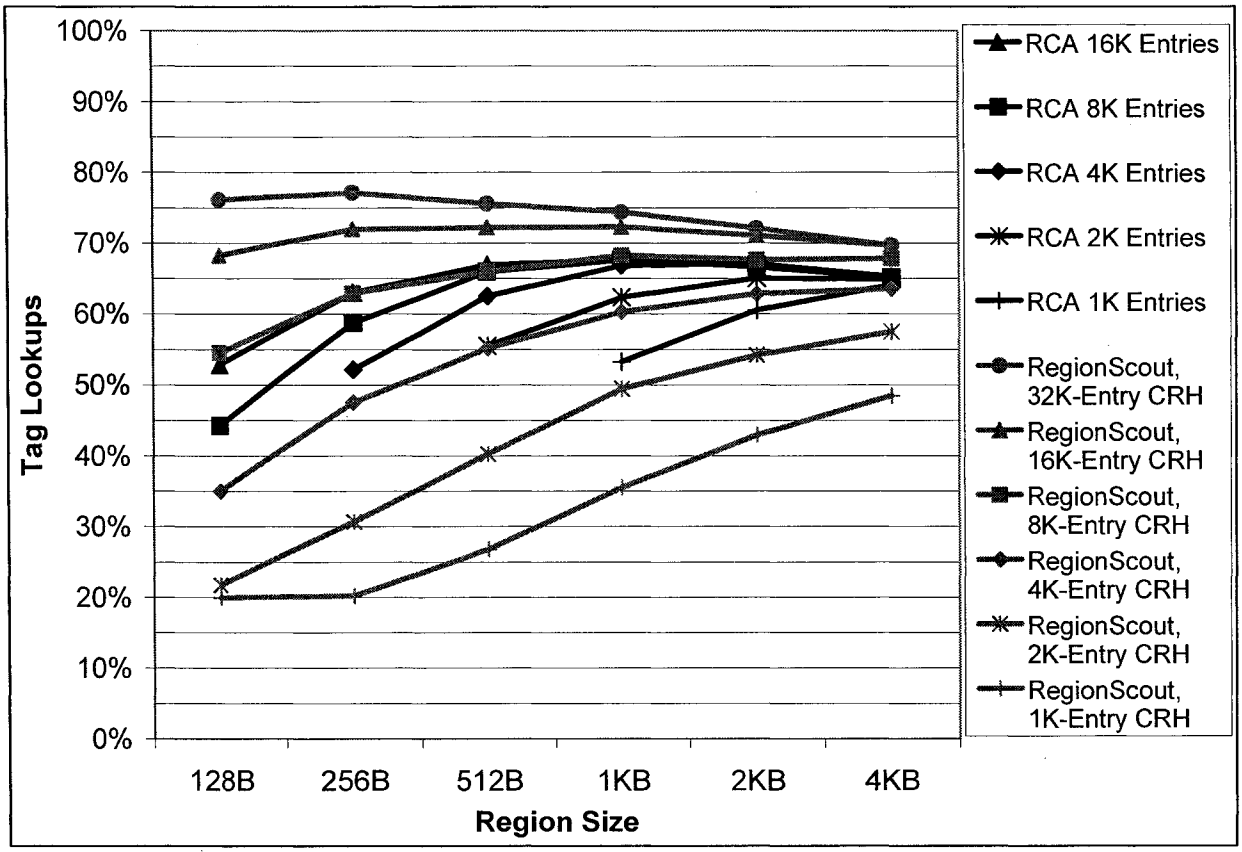

Figure 5.7: Net snoop-induced cache tag lookup filtering comparison

Figure 5.8 shows the same data as Figure 5.7, except the x-axis is now used for the logarithm of the number of kilobytes of storage for each case, and a different curve is plotted for each region size. The y-axis is still the net snoop-induced cache-tag lookups filtered by the two CGCT implementations.

From this graph, the RegionScout Filter almost always outperforms the Region Coherence Array at filtering snoop-induced cache tag lookups for a given amount of hardware storage. The only data point that is an exception to this rule is the Region Coherence Array with 4KB Regions and 1K entries (~4.75 KB of storage). Furthermore the RegionScout Filter reduces cache tag lookups with smaller amounts of hardware, as low as 1-2KB. For smaller combinations of region

size and Region Coherence Array size there are more cache evictions for inclusion, and this reduces the benefit. If not for this, Region Coherence Arrays would likely filter a comparable number of snoop-induced cache tag lookups.

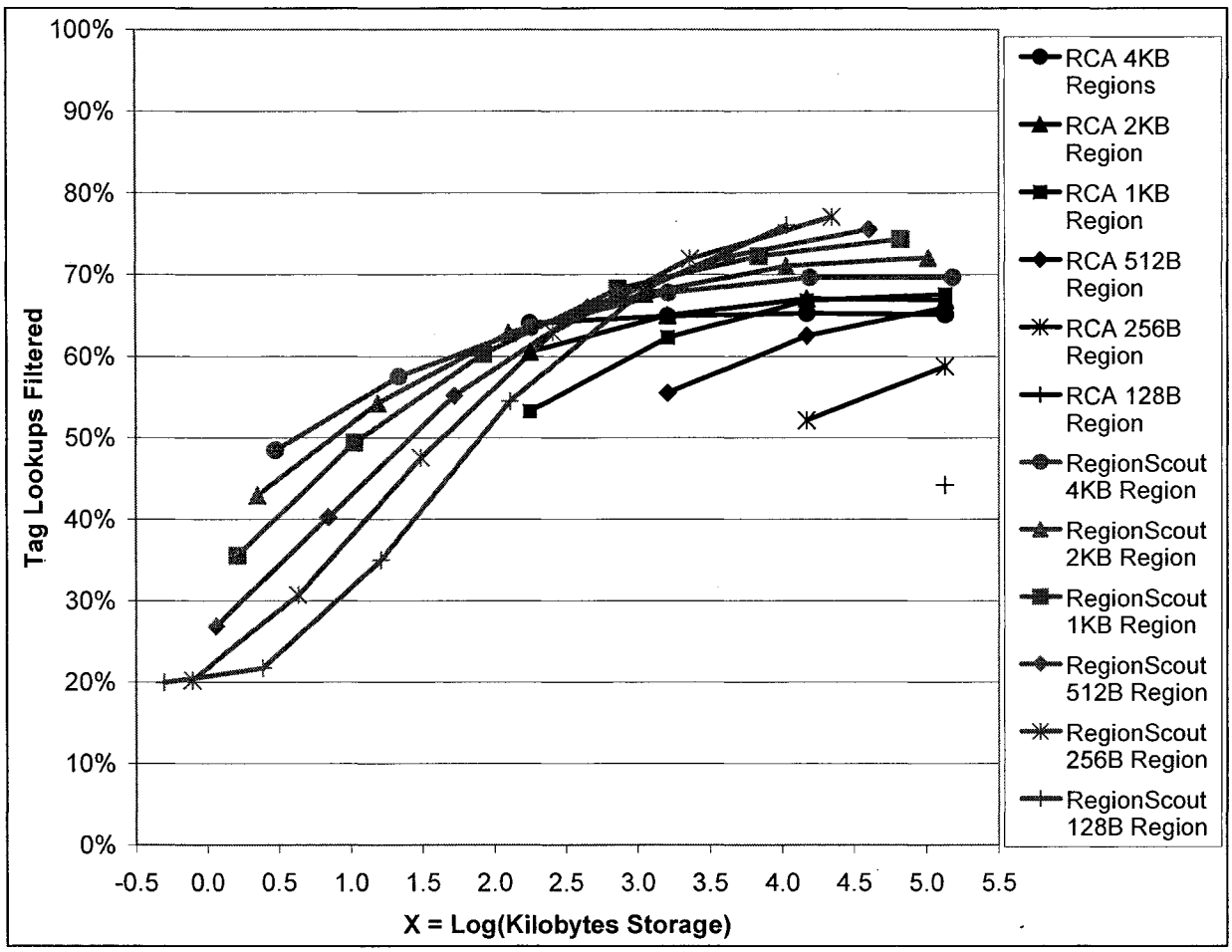

Figure 5.8: Net snoop-induced tag lookups filtered by RegionScout filters and Region Coherence Arrays per kilobyte storage.

The RegionScout Filter scales down to 1KB while still filtering half of the snoop-induced cache tag lookups. The Region Coherence Array performs comparably for small amounts of storage, but does not perform as well as the RegionScout Filter for large amounts of storage. The additional cache tag lookups for inclusion hinder the Region Coherence Array as the amount of storage is decreased.

Figure 5.9 shows the increasing effect of maintaining inclusion for decreasing numbers of entries in the RCA. Note that for all of this data the associativity of the Region Coherence Array is 2-way. As the number of entries is decreased for a given region size, the increase in L2 miss ratio grows quickly and nonlinearly. The RCA with 1K entries has a large increase in L2 miss ratio from the inclusion evictions; over 12% for a 4KB region size. This explains the large discrepancy in snoop-induced cache tag lookup filter rates as the RCA was scaled down.

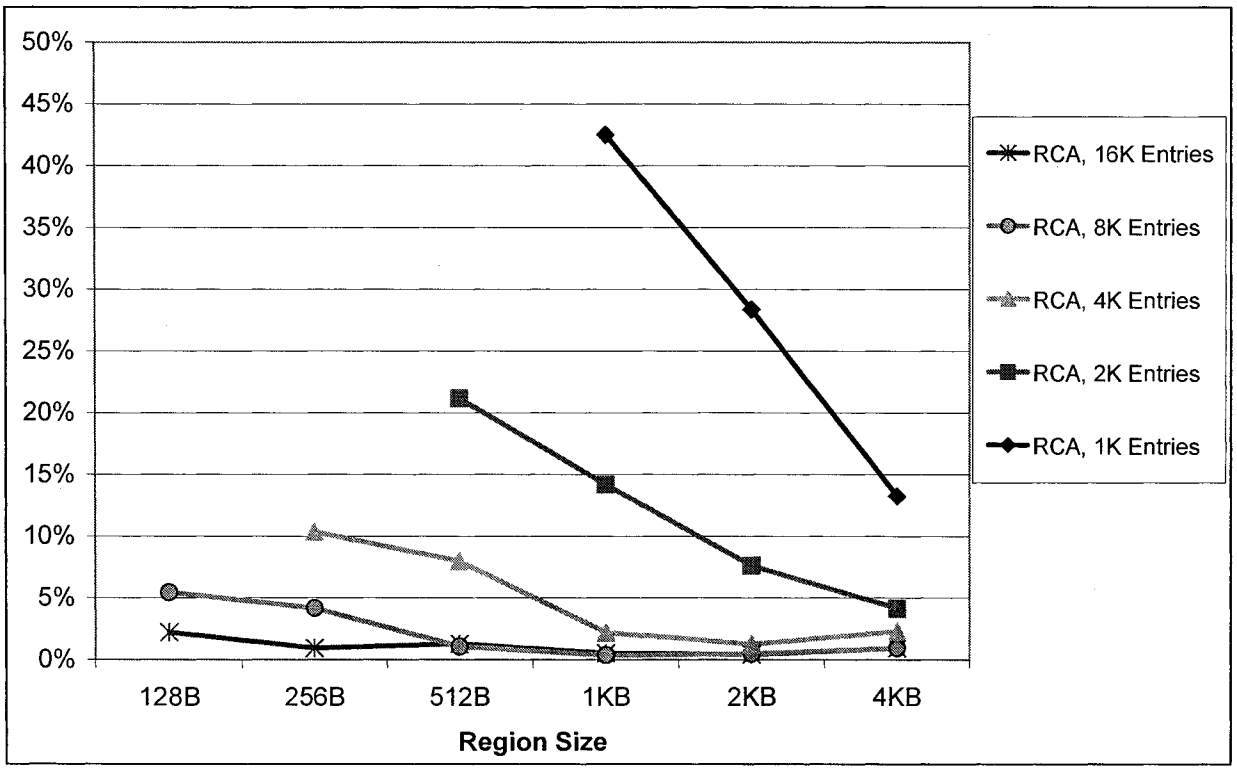

Figure 5.9: Increase in L2 miss ratio for different RCA configurations.
For a fixed region size and decreasing number of entries in the Region Coherence Array, the L2 miss ratio increases nonlinearly. To use small Region Coherence Arrays a large region size must be used; small region sizes quickly become unsuitable due to their impact on system performance.

5.4 Combining Techniques

From the results in the preceding sections, we can conjecture how ideas from both implementations might be combined to develop improved Coarse-Grain Coherence Tracking implementations.

5.4.1 Temporal Locality vs. Latency and Power Consumption

Using a large structure such as a Region Coherence Array to track the external coherence status of regions has the important advantage of exploiting temporal locality in addition to spatial locality. The key is that the structure must map more of the address space than the cache. To save power, the structure would ideally be accessed only after a cache miss has been detected. However, accessing a large structure after detecting a cache miss can add significant latency to external requests. The structure should be small, low-latency, low-power, and have considerable capacity. It may be possible to attain all of these goals with a two-level structure.

A small, fast, and power-efficient structure is used to buffer the external coherence status of the most recently used regions, backed up by a much larger structure that can map more of the address space than the cache. For example, a large Region Coherence Array could be implemented with a small *region cache* for the most recently accessed regions. This cache would have an address tag and region state for each entry. Requests first check the region cache and only check the larger structure if the region is not found. Requests wait until a cache miss is detected before accessing the region cache to save power, and in the common case the region state can be accessed quickly and with little additional power consumption. Temporal locality is exploited by

the larger Region Coherence Array. To improve the hit ratio in the region cache, regions can be prefetched from the Region Coherence Array into the region cache.

Other possibilities exist. A heavily sublined NSRT can be implemented; packing the state for several regions into one entry to extend its reach. The NSRT might also be modified to use a hash index instead of an address tag. When a count in the CRH is zero, it means that no lines are cached from *any* region mapping to that entry. The NSRT could store that hash index instead of an address tag for the region requested, and hence keep information for all regions mapping to a hash index.

5.4.2 Maintaining Inclusion

As the size of the Region Coherence Array is scaled down, the eviction of cache lines for inclusion becomes detrimental to performance. RegionScout Filters have the advantage of not having to evict lines from the cache, and scale down gracefully. However, this comes at the cost of less precise tracking of regions and lost potential for optimization. An important issue is how to achieve the performance of a Region Coherence Array without the cache evictions for inclusion.

Perhaps the two techniques can be combined into a more effective structure. The CRH can be implemented as tagged hash instead of a non-tagged one, keeping an address tag for each entry and two counts. The first count is for lines in the region corresponding to the tag, and the second count is for all other regions mapping to the entry. When both counts are zero, and a processor requests a line in a region, the address tag in the corresponding CRH entry is set for that region. Lines allocated in that region increment the first count only. If lines from other

regions mapping to that same entry are allocated/evicted, the second counter is incremented/decremented for them. For best results, the second counter should be kept at zero as much as possible, so regions with no lines cached can be moved from the first counter to the second, freeing the tag for a region with lines cached. The new hybrid structure requires more area than a CRH, but is less sensitive to data access patterns and physical address mappings.

A complementary idea can be used for Region Coherence Arrays. When regions are evicted from the RCA, the line count can be added to a counter associated with each set in the Region Coherence Array. The regions are only evicted from the tagged area of the structure, and their lines are still represented in the count. External requests that do not match on the tagged entries in the RCA can then check the hash count in the set to provide a conservative region snoop response. This will require a small amount of additional storage over a base RCA implementation, and some potential is lost because the RCA may falsely respond to external requests indicating that lines from a region are cached. However, there is no longer a need to evict lines from the cache to maintain inclusion, no longer a constraint on what data may be simultaneously cached, and the structure can be scaled down to even smaller numbers of entries. Although, there remain implementation issues to resolve, such as how to move lines from the counter in each set when a region is brought back into the tagged portion of the Region Coherence Array.

5.4.3 Targeting Requests to Shared Data

Region Coherence Arrays optimize requests for shared data; RegionScout Filters do not. There is no inherent reason for this, and significant potential to improve scalability by sending instruction fetches directly to memory when the instructions have not been modified.

To optimize requests to shared data, the CRH and NSRT must distinguish between regions that are shared by other processors and regions from which other processors are modifying lines. First, the CRH can be modified to have a separate count in each entry for lines in a potentially modified state. It can then provide a region snoop response indicating whether or not the requested region may have lines cached, and whether lines may have been modified by the processor. This information can be buffered in the NSRT with an additional bit (shared/nonshared). NSRT entries would no longer be discarded if a region became shared; instead they would be maintained until the region is evicted or becomes modified by other processors.

5.5 Summary and Conclusions

This chapter introduced RegionScout Filters, an alternative implementation of Coarse-Grain Coherence Tracking proposed concurrently by others. RegionScout Filters are compared qualitatively and quantitatively to Region Coherence Arrays. Region Coherence Arrays can avoid more unnecessary broadcasts and reduce average and peak broadcast traffic more. However, due to cache evictions to maintain inclusion RegionScout Filters can outperform Region Coherence Arrays at filtering unnecessary snoop-induced cache tag lookups for comparable amounts of storage. In addition, RegionScout filters do not constrain the data that may be simultaneously cached; require fewer modifications to the baseline system; and scale down to smaller amounts of storage. There is potential to combine features of the two techniques to obtain the best of both worlds: RegionScout Filters can be extended to target shared data and

exploit temporal locality, and Region Coherence Arrays can sacrifice some precision to do away with cache evictions for inclusion.

6. Stealth Prefetching

This chapter presents Stealth Prefetching, a new performance-enhancing technique targeted at broadcast-based shared-memory multiprocessor systems. Stealth Prefetching utilizes Region Coherence Arrays to identify large regions of memory that are not shared. Data from those regions can be prefetched aggressively and efficiently, without disturbing other processors and without consuming broadcast bandwidth (i.e., "stealthily").

In Section 6.1 the motivation for Stealth Prefetching is explained. In Section 6.2, the basic concept is described, with an example. Section 6.3 describes our implementation of Stealth Prefetching. This is followed by simulation results in Section 6.4 quantifying the effectiveness and timeliness of Stealth Prefetching, utilization of prefetched data, data network bandwidth overheads, and performance impact. Section 6.5 summarizes findings for Stealth Prefetching.

6.1 Motivation

Modern multiprocessor systems commonly prefetch instructions and data to improve memory system performance [1, 2, 3, 4]. Prefetching is an effective way to overlap memory latency with computation by speculatively fetching data from memory in anticipation of future references. As memory latencies have increased, more aggressive prefetching techniques have been proposed [6, 7, 36]. Unfortunately, most prefetching proposals do not explicitly consider issues specific to multiprocessor systems.

In multiprocessor systems, not only are memory latencies longer and interconnect bandwidth more precious, but prefetching can interact with accesses to shared memory in negative

ways. For example, prefetching data too early (while other processors are modifying the data) leads to detrimental behavior [58]. In multiprocessor systems prefetching must be performed more aggressively to overlap the long latencies, more accurately to avoid wasting bandwidth, and with special care not to downgrade cached copies in use by other processors.

In this chapter, CGCT with RCAs is utilized to identify non-shared regions for prefetching. Data from non-shared regions can be prefetched aggressively and safely (without disturbing other processors). Prefetches for these regions are piggybacked onto demand requests sent directly to memory, and data is fetched from DRAM in open-page mode by the memory controller. RCAs will also be extended to track which lines in a region were used in the past to generate accurate prefetches in the future.

Figure 6.1 illustrates the potential for prefetching non-shared regions. For a set of commercial, scientific, and multiprogrammed workloads, more 60-80% of the remaining lines from a region are touched when the region is brought into the Region Coherence Array in a non-shared state. Stealth Prefetching will exploit this behavior to prefetch regions of data aggressively. However, not all applications and region sizes use most of the lines in a region, so care will be taken to make sure that Stealth Prefetching does not waste copious amounts of data bandwidth.

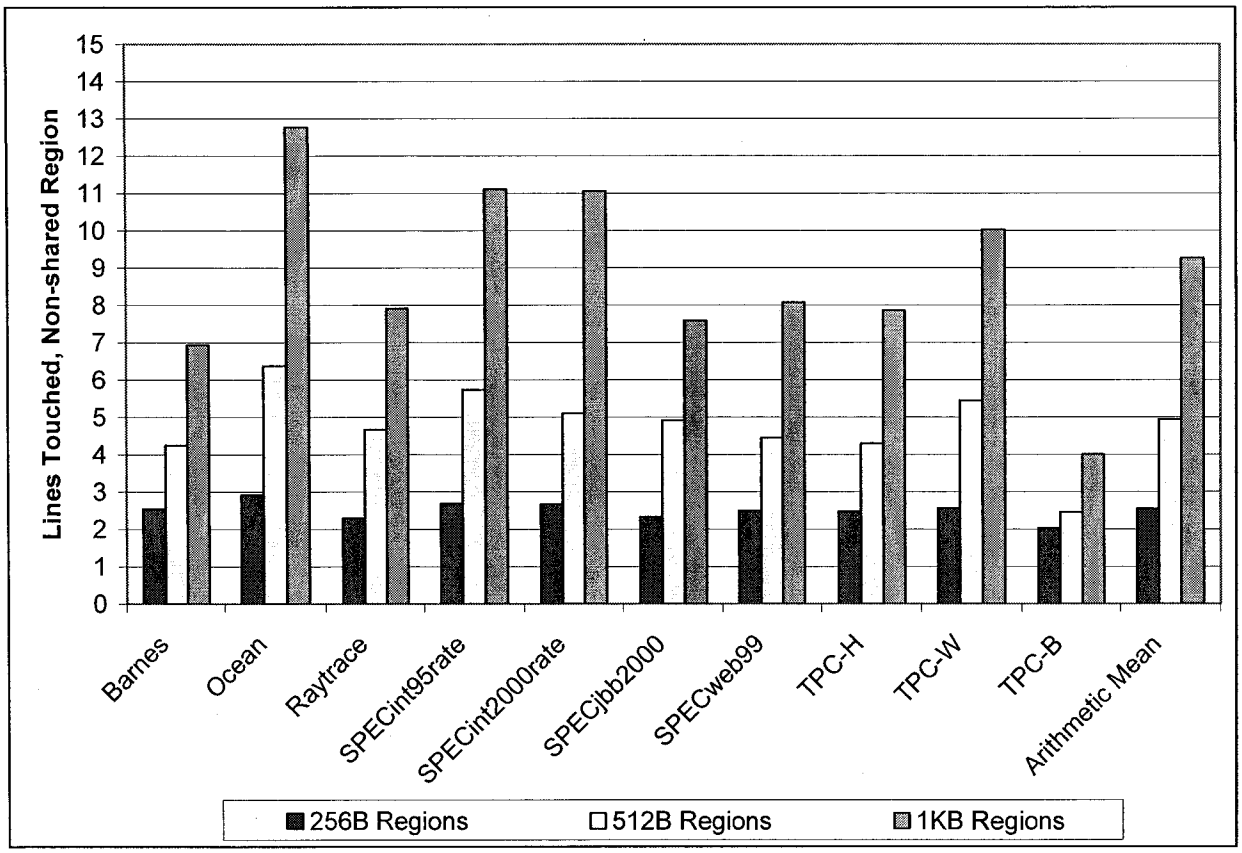

Figure 6.1: Average lines touched from 256B-1KB non-shared regions.

The average number of lines touched while a non-shared region is resident in the Region Coherence Array is shown. Note that one memory access must be performed to bring the region into the RCA; this chart illustrates subsequent accesses that may be prefetched.

6.2 Stealth Prefetching

Stealth Prefetching (SP) is a new technique that utilizes Region Coherence Arrays to aggressively, stealthily and efficiently prefetch data in broadcast-based shared-memory multiprocessor systems. SP is based on the observation that large regions of memory exhibit similar sharing patterns and that the processor accesses most of the lines in non-shared regions. As a result,

accesses to a non-shared region can be treated as a prefetch trigger, with the remaining lines in the region being prefetched to the requesting processor in anticipation of future references.

After a region has been identified as non-shared and a threshold number of accesses to the region have occurred, a prefetch request is sent to memory. This prefetch request is piggybacked onto the direct request that triggered the prefetch, and contains a bit mask of lines to prefetch from memory. By piggybacking prefetch requests on direct requests, they go directly to memory with low latency and without consuming broadcast bandwidth. The memory controller fetches the requested lines from DRAM in open page mode, and sends them back to the requesting processor. The prefetched lines are buffered in a pseudo-invalid coherence state until accessed coherently by the processor, evicted to make room for other prefetched lines, invalidated by external requests to obtain modifiable copies of those lines, or invalidated because the region has been evicted from the Region Coherence Array.

To access prefetched data the processor must first access the Region Coherence Array to determine if the region is present and to globally order the request, the buffer is then checked for the line and if it is present it is moved to the cache. The prefetched lines may be used as long as the region is present in the Region Coherence Array and the lines have not been invalidated by other processors' requests to obtain modifiable copies of the lines. If a prefetched line is requested by another processor, to avoid harming performance the prefetched line is invalidated to allow the requesting processor to obtain an exclusive copy of the line (optionally, the line may be transferred from the buffer to the other processor's cache to reduce latency). Prefetched lines are conservatively invalidated when a region is evicted from the RCA because another processor may obtain exclusive access to the regions and modify lines without broadcasting requests.

For example, consider a broadcast-based shared-memory multiprocessor system with caches, an RCA and a buffer for prefetched data in each processor. Processor *A* performs a load of line *x* that misses in the cache. Line *x* is part of region *Rx*, which is in the RCA of processor *A* in non-shared state. A read request is sent directly to memory for line *x*, along with a bit mask of other lines in *Rx* to prefetch. The memory controller fetches line *x* and the other requested lines in *Rx* from DRAM in open-page mode. It sends this data back to processor *A*, which loads line *x* into its cache in an exclusive state and loads the prefetched lines into the buffer for later use. Subsequent requests from processor *A* that do not require a broadcast can obtain data from the prefetch buffer without an external request. As long as region *Rx* remains in the RCA of processor *A* in a valid state, requests by other processors to obtain a modifiable copy of a line or downgrade the region state must be broadcast, and processor *A* will observe the broadcast and invalidate prefetched lines in the buffer to ensure coherence is maintained.

In effect, the movement of region *Rx*'s data from memory to the prefetch buffer collocated with processor *A* has been decoupled from the request and acquisition of coherence permissions for *Rx*. If subsequent requests to region *Rx* occur at processor *A* (which our empirical findings indicate is the common case) they can be satisfied locally and efficiently because the region protocol can grant access permission and the prefetch buffer can supply data.

Prefetching in this manner is aggressive because large regions of data are prefetched at once; stealthy in that prefetch requests are not broadcast; and safe because it does not interfere with other processors sharing or modifying data, does not prevent other processors from obtaining exclusive copies of lines, and does not pollute caches. It is efficient because lines can be obtained quickly and power-efficiently from DRAM in open-page mode. Finally, it can be

made accurate by extending the Region Coherence Array to track which lines in a region the processor referenced previously.

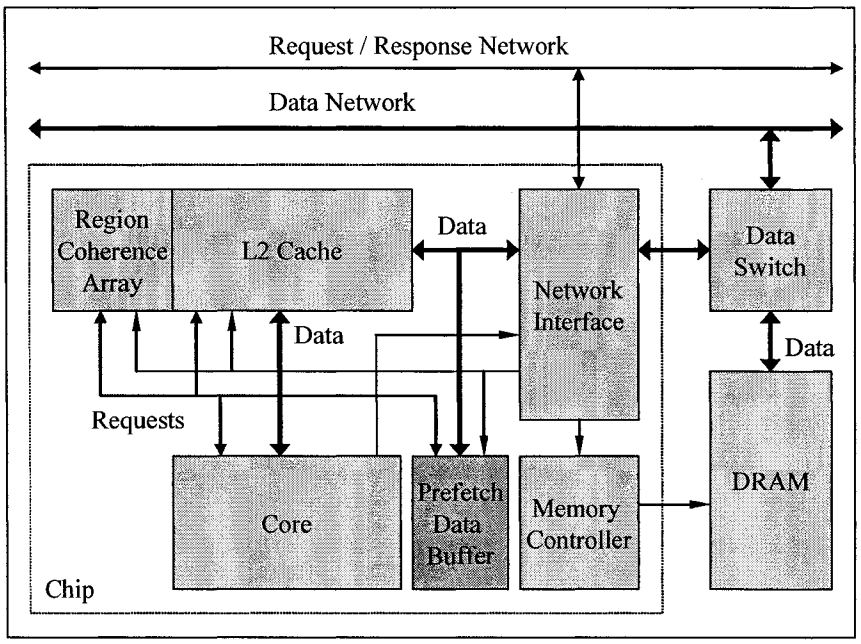

Figure 6.2: System modified to implement stealth prefetching

Figure 6.2 illustrates a system modified to implement SP. The starting point is a broadcast-based shared-memory multiprocessor system that implements CGCT with Region Coherence Arrays. A prefetch buffer is added to store prefetched data until it is accessed by the processor.

6.3 Implementation

Stealth Prefetching can be implemented in a broadcast-based shared-memory multiprocessor system that implements CGCT with Region Coherence Arrays. The implementation consists of a prefetch buffer [47], a protocol for managing prefetched data, a policy for determining when and what to prefetch, some modifications to the RCA, a minor modification to the memory

controller, and a bit mask in the request packets sent directly to the memory controller to inform it to prefetch lines.

6.3.1 Stealth Data Prefetch Buffer

The Stealth Data Prefetch Buffer (SDPB) is a tagged, sectored data array. Each entry contains an address tag, LRU bits, storage for each of the lines in the region, and two bits for each line for the protocol state.

Entries in the SDPB are allocated when a region is prefetched. The newly allocated region is marked as the MRU in the set, and the data for the evicted region (if any lines remain) are discarded. Once lines in the SDPB are accessed by the processor they are moved to the cache and invalidated from the SDPB. The accessed region is marked as the MRU in the SDPB to keep regions with useful data cached. Additionally, the replacement policy can avoid evicting regions for which the data has not yet arrived.

The SDPB does not obtain or maintain coherence permissions over individual lines; it relies on the Region Coherence Array to ensure that accesses to the prefetched data are coherent. Lines from a region must be invalidated if another processor gains exclusive access to a region (e.g., the region is evicted and another processor requests the region afterward), or another processor modifies the line. In this work, prefetched lines from a region are discarded whenever a region is evicted or invalidated and whenever there is an external request for the line.

6.3.2 Protocol

A four-state protocol manages lines in the SDPB (Figure 6.3). The states are *Invalid*, *Pending Data*, *Pending Requested Data*, and *Valid*. All entries start out in the *Invalid* state, indicating the data is not valid and no prefetch in progress.

When a prefetch is initiated, an entry is allocated in the SDPB, and the lines masked for prefetch are set to the *Pending Data* state. From this state, the entry can be upgraded to *Valid* when data arrives, discarded if there is an external request for the line (or the processor flushes the lines), or changed to a second pending state if there is a processor request that hits on the entry and the data has not yet arrived.

The second pending state *Pending Requested Data* indicates that the data has not yet arrived but there is a processor request waiting for it. When the data arrives it will be forwarded to the cache and the buffer entry will go to the *Invalid* state. Entries in this state cannot be discarded; the Region Coherence Array globally ordered the processor request that caused the line to enter this state, and the cache is waiting for the data as it would for data from memory. External invalidates or processor requests to flush the line will hit on the pending state in the cache and will be handled by the conventional cache coherence protocol.

When the prefetch of a line is completed and there are no pending requests for the data, the SDPB entry for the line goes to the *Valid* state. This state indicates that prefetched data is present and may be accessed by the processor. When accessed by the processor, the data is moved to the cache and the line is invalidated in the SDPB.

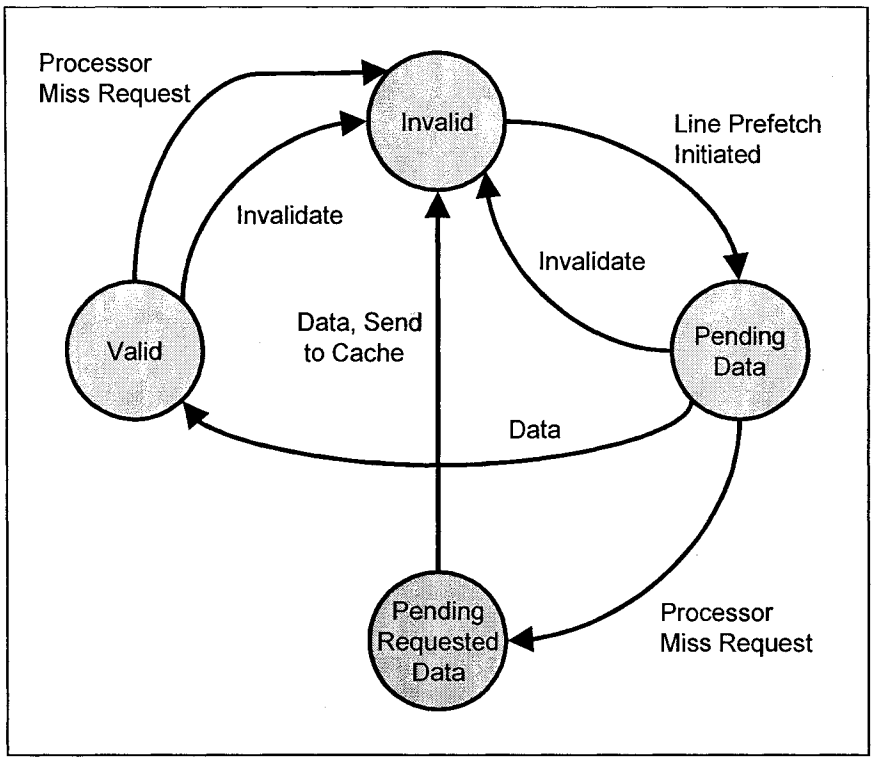

Figure 6.3: Data prefetch buffer protocol

6.3.3 Prefetch Policy

Two key issues are when to first start prefetching a region of data and when to prefetch lines from that region again. If prefetching is performed too aggressively, useless data will be transferred. Conversely, if too many lines in the region must be touched before prefetching the rest of the lines, potential is lost and prefetches become less timely. This section considers the effect of having a threshold number of L2 misses (the number of misses to lines in the region before the rest of the lines in a region are prefetched).

Figure 6.4 depicts the average distribution of lines touched per non-shared region. The X-axis is the number of lines touched, and the Y-axis is the percentage of all non-shared regions touched. Ideally, the graph should resemble a step-function with 100% of the non-shared regions

having all lines used. In that case touching a single line in the region would be an indication that the rest of the lines would be used, and prefetching should be performed aggressively. In reality, not all of the lines are used from 50% of the non-shared regions.

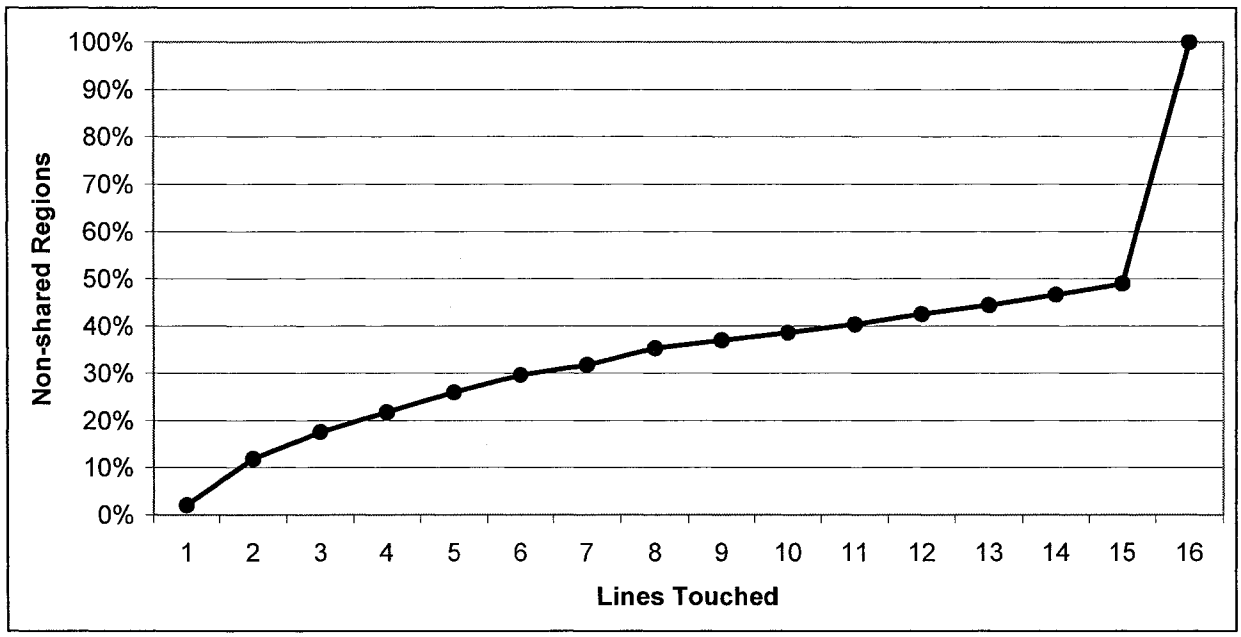

Figure 6.4: Distribution of lines touched per non-shared region.

Some threshold number of L2 misses is necessary to avoid prefetching useless data. To determine the threshold value, one can take the data points from Figure 6.4, and for different threshold numbers of L2 misses (to occur before prefetching) calculate the probability that one additional line will be used, two additional lines will be used, and so on. The probability that one additional line will be used is multiplied by one; the probability that two additional lines will be used is multiplied by two, and so on. The sum of these products is the average number of lines used after a threshold number of L2 misses. We can subtract this number and the threshold from the number of lines in the region to determine the average number of useless lines prefetched. Figure 6.5 displays the result of this computation, along with the average number of lines

touched per non-shared region for comparison. The threshold in the graph starts at two because it takes at least one access to the region to bring it into the RCA and determine its external coherence state, prefetching should begin on a subsequent access.

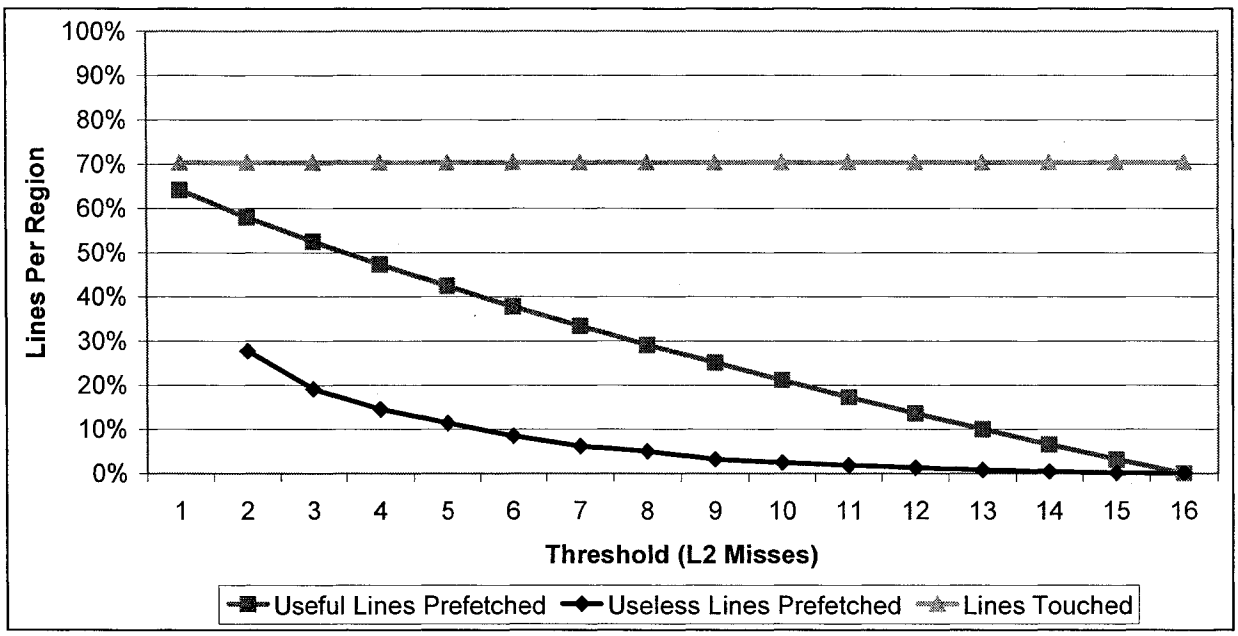

Figure 6.5: Average useful lines prefetched for a 1KB non-shared region with varying thresholds.

With a threshold of two L2 misses, nearly 60% of the lines in a region are prefetched and used, and nearly 30% of the lines in a region are prefetched but not useful. Both the number of useful and useless lines prefetched decrease steadily with increasing threshold. While the useless lines prefetched decreases more quickly at first, there is no improvement in the ratio of useful lines prefetched to useless lines prefetched as the threshold is increased. Figure 6.6 shows this ratio for increasing thresholds. While increasing the threshold number of L2 misses will result in prefetching less useless data, it leads to prefetching even less of the useful data.

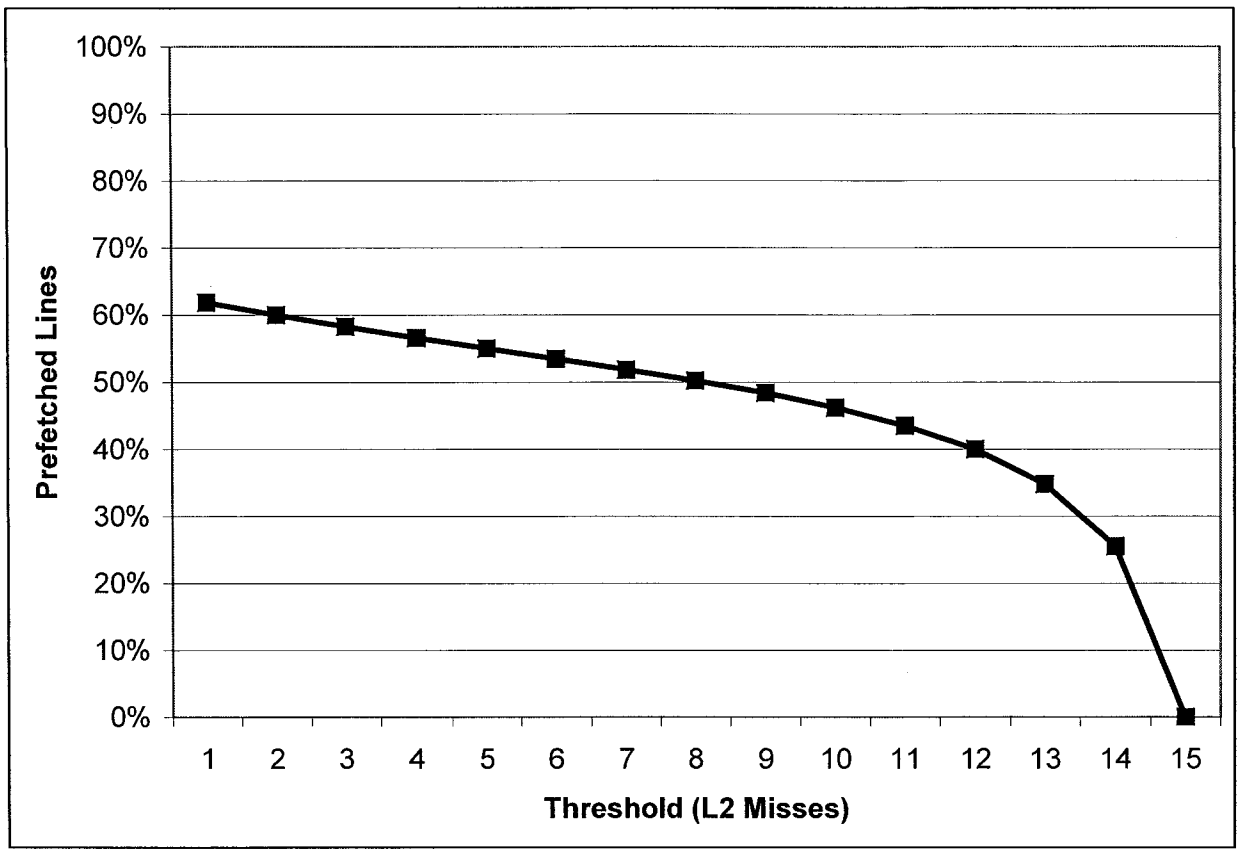

Figure 6.6: Average lines prefetched that are useful for a 1KB non-shared region with varying threshold number of L2 misses before prefetching.

There is a simple explanation for this. If a region contains X lines of which Y are used, the percentage of useful lines in the region is X/Y. Based on Figure 6.2, assume that on average X is less than Y (i.e., $X = Y - \Delta$). If a threshold T lines are touched before prefetching the region, the number of useful lines remaining in the region is $(X-T) / (Y-T)$. By substitution we have $(X-T) / (X-T+\Delta)$; the numerator and denominator decrease equally with increasing T, and the ratio of useful lines remaining to useless lines remaining decreases. The calculation becomes more complex when considering temporal locality, but the relationship is the same. When lines are

fetched into the cache again on average some percentage of the remaining lines in the region will be fetched again, only the ratio may be smaller because some useful data remained in the cache.

Based on these findings, the threshold T was set to the minimum value two to maximize miss coverage (the first request bring the region into the RCA, and the prefetch is piggybacked onto a subsequent request if the region is non-shared). Furthermore, T is also used to control the re-prefetching of lines in a region. A region of data is prefetched initially, after T L2 misses occur. The region is considered for prefetch again after there have been T further L2 misses since the last region prefetch. Only this time, instead of prefetching all the remaining lines, only the lines touched since the last prefetch are prefetched. The RCA keeps track of the lines in the region that have been touched with a bit-mask; this mask is used to reduce the number of useless prefetches. This approach is similar to the density vectors used in prior work [7].

6.3.4 Modifications to RCA

The RCA in each processor chip must be modified to better track which lines are cached and lines which have been cached since the last region prefetch. First, each entry needs a set of presence bits to keep track of the lines that are in the cache to avoid prefetching data that is already cached. These presence bits can be used in place of the line count used to detect empty regions and enable the RCA to more efficiently maintain inclusion over the cache. Second, each entry needs a set of bits to keep track of the set of lines that were referenced earlier, in order to improve prefetch accuracy. The bit mask sent to memory is formed by the logical product of these reference bits, the complement of the presence bits, and the complement of a bit mask of lines resident in the SDPB.

For a 2-way set-associative RCA with 8K sets and 256B regions, the modifications to the RCA increase the size of its sets by 10 bits each (12.5% more storage). For large, 4KB regions that masks are 64 bits, and the additional storage per set is 242 bits, increasing the size of the RCA by a factor of four.

6.3.5 Modifications to the Memory Controller

The memory controllers in the system need to determine when to prefetch a region of data and send it to the processor. It would be detrimental to performance to attempt to prefetch the lines in the region around every demand-requested line. Instead, a bit mask is sent to the memory controller in the message packet of the request that triggered the prefetch. This bit mask is used to communicate when and what to prefetch; the memory controller does not prefetch lines unless instructed.

For reduced latency and power consumption, the DRAM can fetch the requested lines in open-page mode. The designer may further reduce latency and leave the DRAM page open after the initial access to a line in the region in anticipation of a prefetch, at the cost of additional power to leave pages open.

6.4 Results

In this section we quantify the effectiveness, timeliness, performance improvement, data utilization, and data traffic overhead of Stealth Prefetching.

6.4.1 L2 Misses Covered

Figure 6.7 shows the percentage of L2 misses that hit in the SDPB for each workload and different region sizes. In these simulations, the SDPB contains storage for 256 lines (4-128 regions depending on region size) and is 4-way set-associative. The first four bars for each workload illustrate the effect of increasing the region size, and the rightmost bar illustrates the effect of a *perfect prefetcher* (all buffer accesses hit; the buffer is accessed after the RCA determines a broadcast is not necessary to coherently acquire the data). On average, close to 30% of the L2 misses hit in the SDPB and do not suffer the latency of a miss to memory for 1KB regions. Benefiting most is Ocean, for which nearly 70% of the L2 misses are prefetched. The next best performers are TPC-W, for which over 50% of the L2 misses are prefetched.

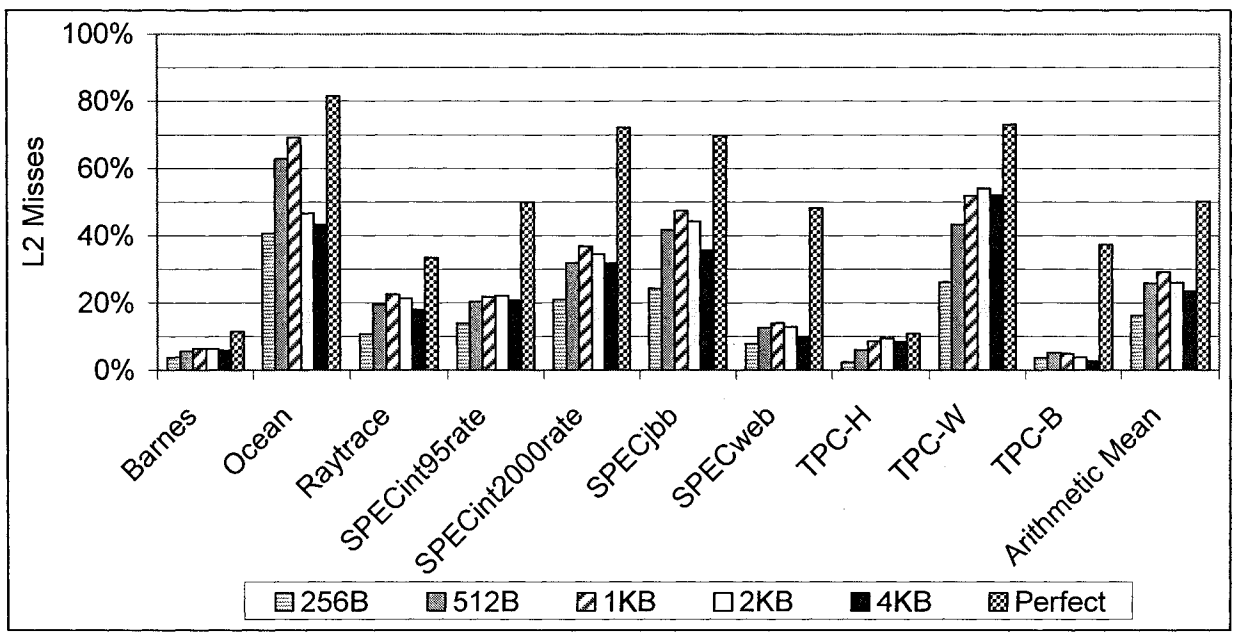

Figure 6.7: Reduction in L2 miss rate with Stealth Prefetching

Figure 6.8 illustrates the rate of L2 misses per instruction. Bars for the baseline miss rate and the miss rate with CGCT are shown for comparison. Note that the miss rate with CGCT alone is

slightly higher than the baseline miss rate due to the RCA maintaining inclusion. Stealth Prefetching is based on CGCT with Region Coherence Arrays, and this increase offsets the some of the benefit of Stealth Prefetching.

Again, Ocean has the largest reduction, followed by TPC-W and SPECjbb. These applications have relatively high miss rates and should gain the most performance from the reduction in L2 misses. Barnes and SPECint2000rate do not have a large miss rate compared to the others and should not benefit as much.

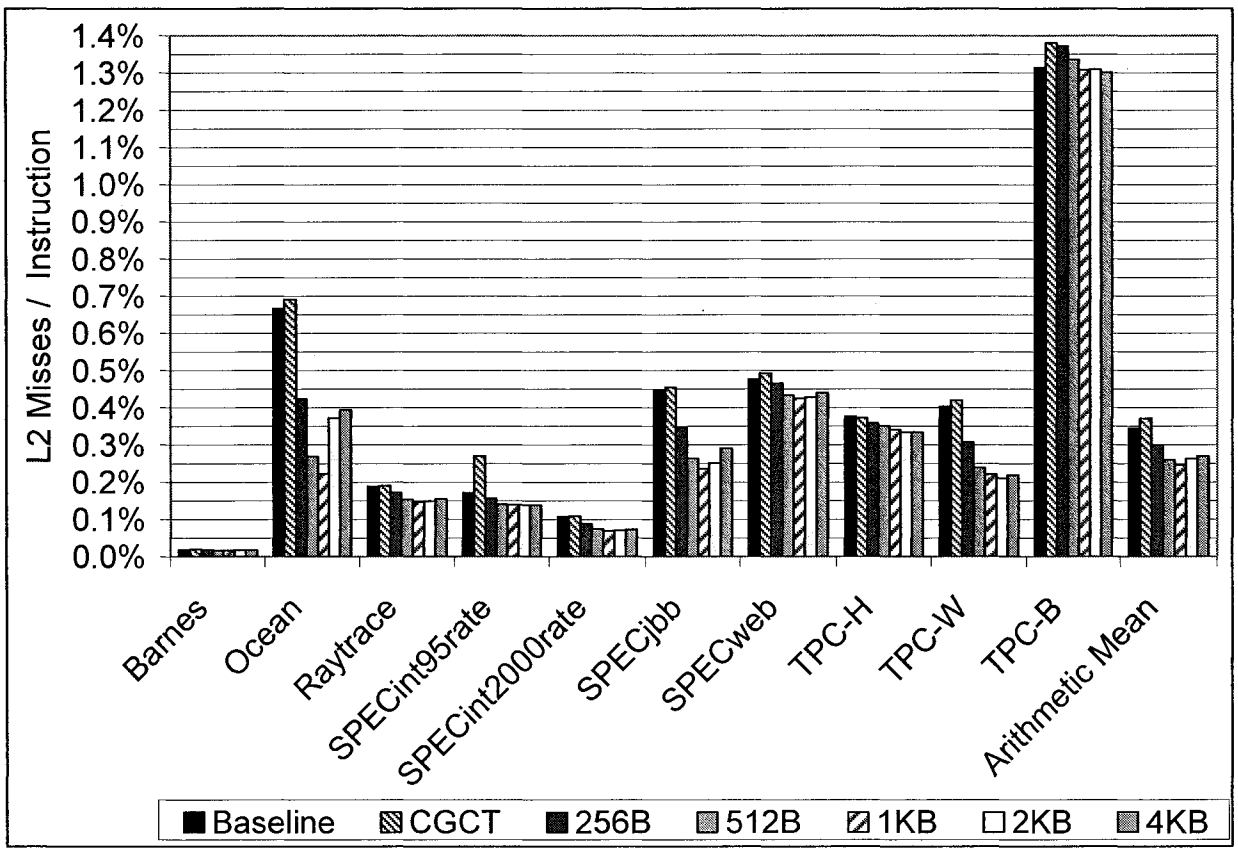

Figure 6.8: L2 misses per instruction with Stealth Prefetch

6.4.2 Performance Improvement

Figure 6.9 contains the normalized execution time of each benchmark, starting with the baseline and CGCT, and adding Stealth Prefetching with different SDPB sizes. Overall performance is improved; execution time is an average of 15% lower than the baseline system for 1KB regions, and an additional 9.4% lower than for CGCT alone. As predicted, Ocean and TPC-W have the largest run time reductions of 46% and 39%, respectively.

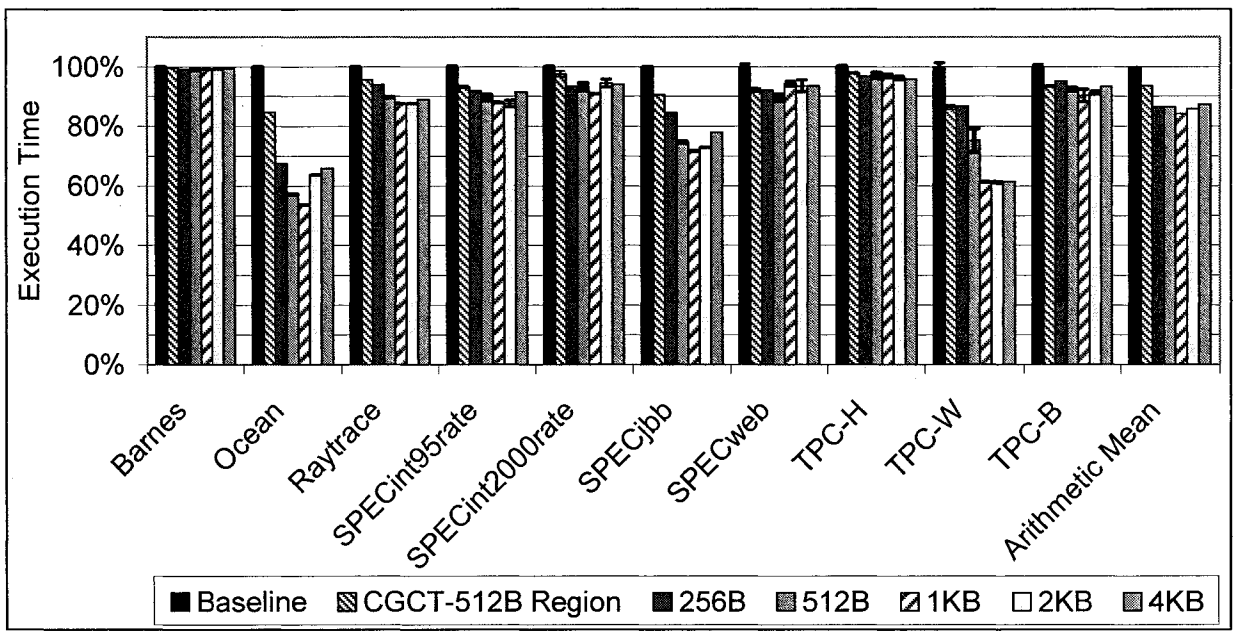

Figure 6.9: Execution time

6.4.3 Data Utilization and Traffic

As with all prefetching techniques, some prefetched data is unused. The average data traffic for each application and region size is shown in Figure 6.10. On average there is less than 20% more data traffic for region sizes of 512B and lower, 27% for more for 1KB regions, and 39-52% more for large regions of 2KB-4KB, respectively. Most of the average data traffic increase is due to

SPECjbb, which benefits significantly from prefetching but just needs larger data buffers to hold data for larger regions. From the arithmetic means, it appears that doubling the region size leads to an approximately linear 10% increase in data traffic, and perhaps a doubling of the prefetch data buffer will provide a similar reduction in data traffic.

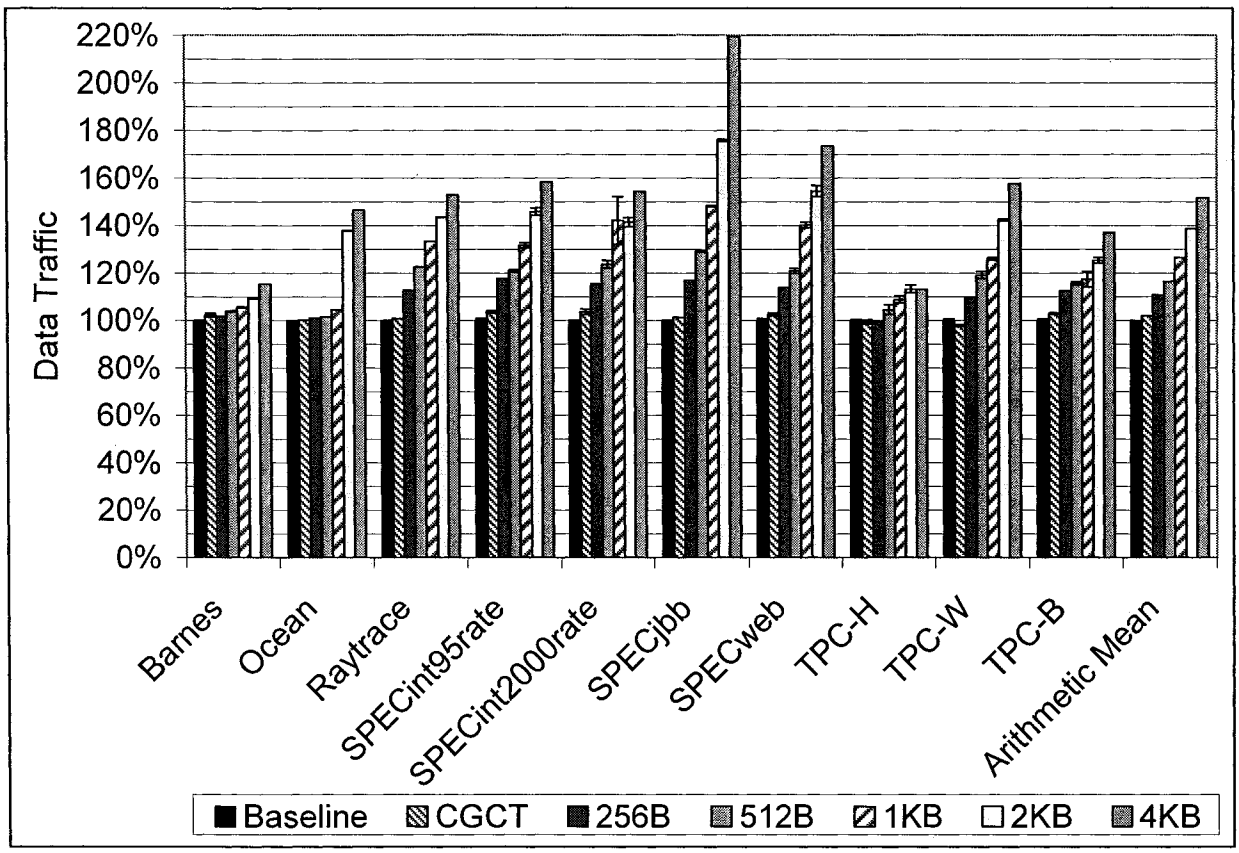

Figure 6.10: Average data traffic overhead

The increase in data traffic is due to data that is prefetched, but not referenced before being replaced or invalidated from the buffer. As shown in Figure 6.11 below, an average of 33-67% of the lines prefetched are used, depending on the region size.

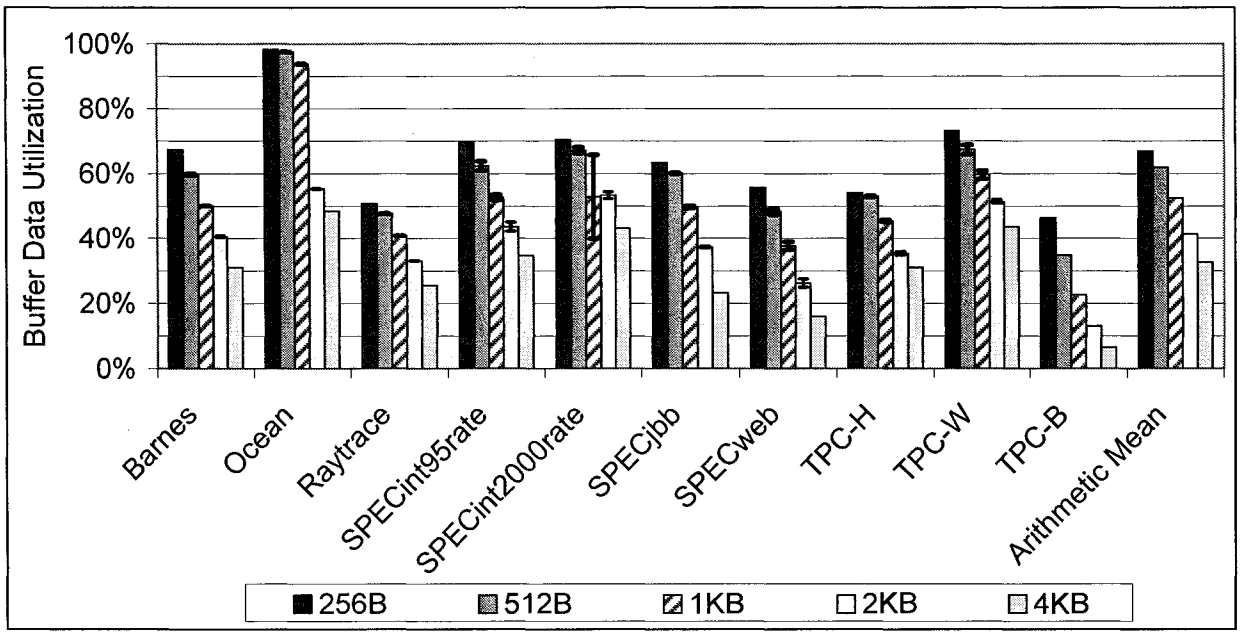

Figure 6.11: Stealth data prefetch buffer utilization

6.5 Summary

This chapter proposed and evaluated Stealth Prefetching, a new performance-enhancing technique utilizing Region Coherence Arrays. Stealth Prefetching is simple to implement, requiring a buffer for prefetched data, a bit mask in the requests sent directly to memory, and logic in the memory controllers to prefetch regions of data on request. Stealth Prefetching is shown to provide data for 29% of the L2 misses on average with 1KB regions and a 256-entry 16-way sectored data buffer (over 50% for some applications). This leads to average run-time improvements over the baseline of 15%, and nearly 10% over the system with CGCT alone. However, there is still work to be done in order to improve prefetched data utilization and reduce data traffic overheads.

7. Power-Efficient DRAM Speculation

Power-Efficient DRAM Speculation (PEDS) is a new optimization that takes advantage of information provided by Coarse-Grain Coherence Tracking with Region Coherence Arrays to avoid speculatively accessing DRAM unnecessarily. The external region state is used to identify requests that are likely to be satisfied with data from other processors' caches. To save power, the memory controllers do not speculatively access DRAM for those requests.

In Section 7.1 the motivation behind PEDS is explained. Section 7.2 describes the basic concept. This is followed by a discussion of the implementation of PEDS (Section 7.3), including extensions to the Region Coherence Array to improve PEDS. Section 7.4 presents simulation results quantifying the effectiveness of PEDS, and its impact on performance. Section 7.5 discusses possible enhancements to improve our implementation of PEDS. Section 7.6 summarizes and concludes the chapter.

7.1 Motivation

In a paper presented at ISSCC 2006 [59], Sun Microsystems revealed that the DRAM power consumption of the UltraSPARC T1 (Niagara) systems running SPECjbb was approximately 60 watts; greater than 22% of the total system power consumption [60]. This is almost as much power as the CPUs consume, making DRAM power consumption an important design consideration in modern multiprocessor systems.

Modern multiprocessor systems commonly access DRAM speculatively to improve performance [1, 2, 3, 4]. The DRAM access is started part-way through the broadcast snoop,

after the request reaches the memory controller, but before it is known whether memory or another processor will source the data. By starting the DRAM access early, part of the DRAM latency is overlapped with the broadcast snoop and the average memory latency is reduced. However, in cases where a modified copy of the data resides in another processor's cache, fetching data from DRAM is useless and wastes power.

Results for a four-processor system running commercial, scientific, and multiprogrammed workloads indicate that approximately 33% of lines read speculatively from DRAM are discarded (25.6% of all DRAM requests, see Figure 7.1). While most read requests benefit from the lower latency of speculatively accessing DRAM, a third of them waste DRAM bandwidth and waste power. Furthermore, these percentages can increase with cache size, as cache misses to memory are replaced with cache-to-cache transfers.

While 33% of lines read speculatively from DRAM are useless, the remaining 66% provide useful data. If DRAM was not speculatively accessed, 66% of read requests would be delayed, and would have latency greater than the total broadcast snoop latency and the total DRAM latency combined. The impact on system performance would be significant.

The problem is less acute (but still significant) for systems that implement Coarse-Grain Coherence Tracking with Region Coherence Arrays, in which nearly half of the read requests are not broadcast. Such requests are *non-speculative DRAM reads*; there is no broadcast and no combined snoop response to indicate that the data will come from another processor. Accounting for these, approximately 15-16% of reads will be delayed if speculative DRAM accesses are not performed.

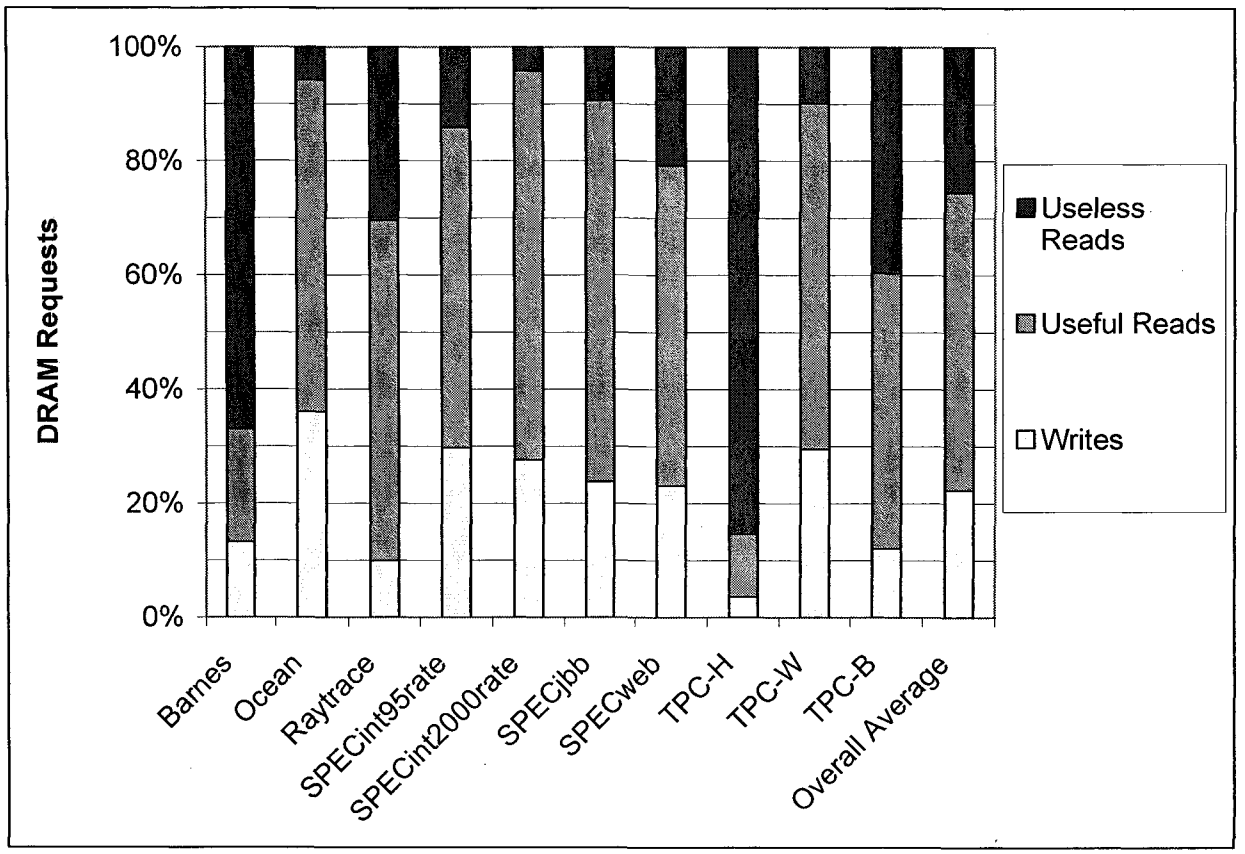

Figure 7.1: Breakdown of DRAM requests into Writes, Useful Reads, and Useless Reads.

On average, two-thirds of DRAM reads benefit from speculatively accessing DRAM (52% of all DRAM requests). However, 33% of DRAM reads speculatively access DRAM unnecessarily, totaling 25.6% of all DRAM requests. The remaining DRAM requests are writes (22.4%).

If it could be determined beforehand that a memory request will be satisfied with data from another processor's cache, the memory controller could avoid speculatively fetching the data from DRAM for that request. This would reduce DRAM activity, contention for DRAM bandwidth, and power consumption. Performance would not be adversely affected because requests that need data from memory could still access DRAM part-way through the snoop. It

would be an effective way to reduce DRAM power consumption if the hardware overhead was kept small.

7.2 Power-Efficient DRAM Speculation

Power-Efficient DRAM Speculation (PEDS) takes advantage of information provided by Coarse-Grain Coherence Tracking with Region Coherence Arrays to detect requests for which data is likely to be sourced from another processor's cache, and to save power avoids speculatively accessing DRAM for those lines.

PEDS adds one bit to memory read requests to inform the memory controller whether it should or should not speculatively fetch the data from DRAM. Requests tagged as bad candidates for a speculative DRAM access are buffered by the memory controller until the combined snoop response arrives to validate the prediction. At that time if the combined snoop response indicates no processor will source the data, the prediction was incorrect and the DRAM read was delayed unnecessarily. The line is fetched from DRAM and sent to the requesting processor. If the combined snoop response indicates that another processor will source the data, the prediction was correct and the request can be discarded. Existing memory controller queues can be extended with more entries and flow control for buffering these DRAM reads while waiting for the combined snoop response to arrive.

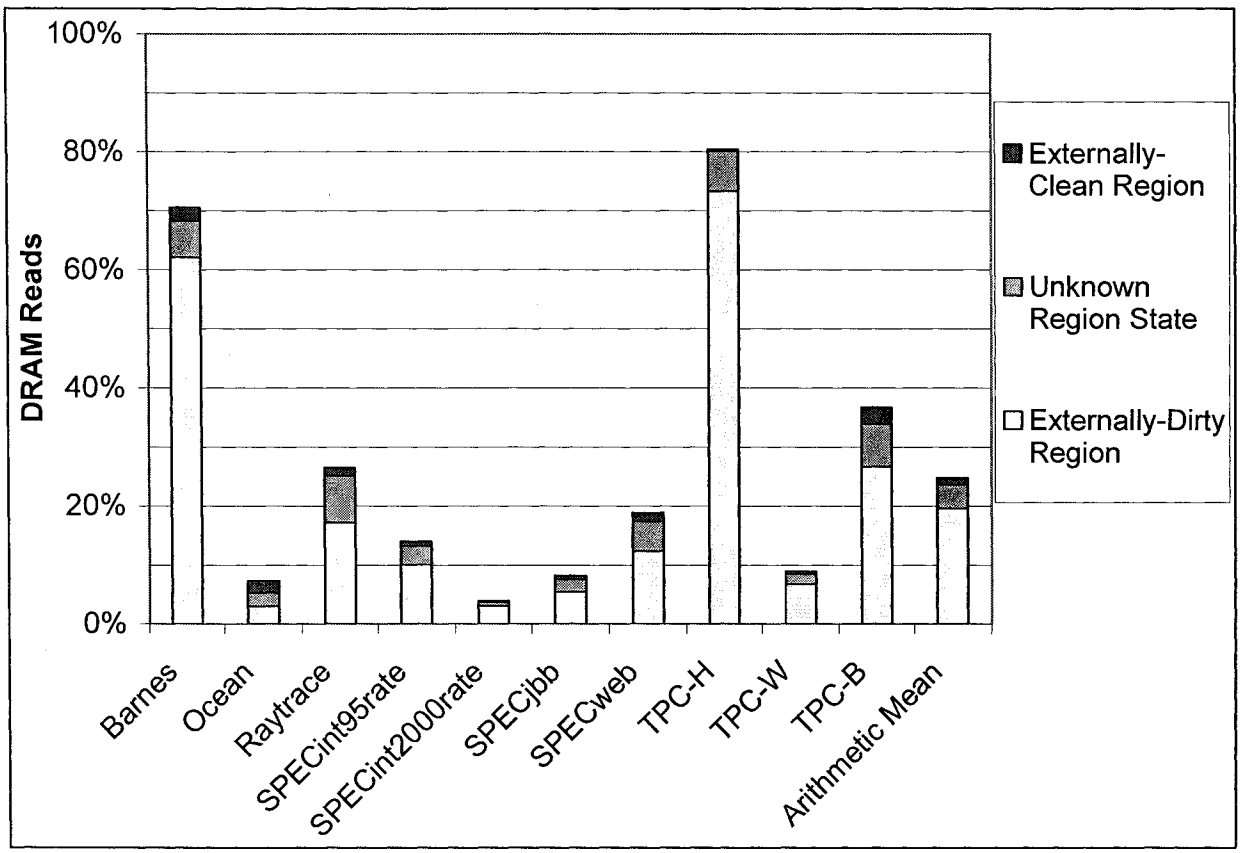

Figure 7.2: Breakdown of useless DRAM reads by external region state.
On average 20% of the DRAM reads broadcast while the region state was externally-dirty accessed DRAM unnecessarily. Another 4% of the reads were broadcast while the external region state was unknown (the external region state is unknown for requests that miss in the RCA). The smallest contribution is reads to externally-clean regions.

Figure 7.2 illustrates the potential of Region Coherence Arrays for detecting read requests that will obtain data from other processors' caches. For each application, the percentage of read requests that obtain data from other processors' caches is shown in Figure 7.2, broken down by external region state. This data was collected for a system with 1MB L2 caches, and a 2-way set-associative Region Coherence Array with 8K sets and 512B regions. Most of the reads that speculatively access DRAM unnecessarily originate from processor requests for which the

region state in the RCA is externally-dirty, or unknown because the region is not present in the Region Coherence Array. This is the potential effectiveness of predicting based on external-region state at the time the request is broadcast.

Figure 7.3 shows the percentage of requests for each external region state for which speculatively accessing DRAM is not useful, and hence the potential accuracy of predicting not to speculatively access DRAM based on the external region state at the time the request was broadcast. Over 75% of the DRAM read requests for which the region state was externally-dirty at the time they were broadcast do not obtain data from DRAM. From Figures 7.2 and 7.3 it is clear that most of the useless DRAM reads can be avoided by not speculatively accessing DRAM for requests when the region state is externally-dirty. Approximately 20% of the DRAM reads can be eliminated, with approximately 7% (20/3) of the useful reads delayed. The next most significant contribution is from regions with externally-unknown region state. Less than 35% of these DRAM reads are unnecessary, meaning that not speculatively accessing DRAM would delay more reads than it would eliminate. Reads from externally-clean regions make up only a small fraction of the unnecessary DRAM reads and have a low probability of obtaining data from another processor's cache; they can be safely ignored.

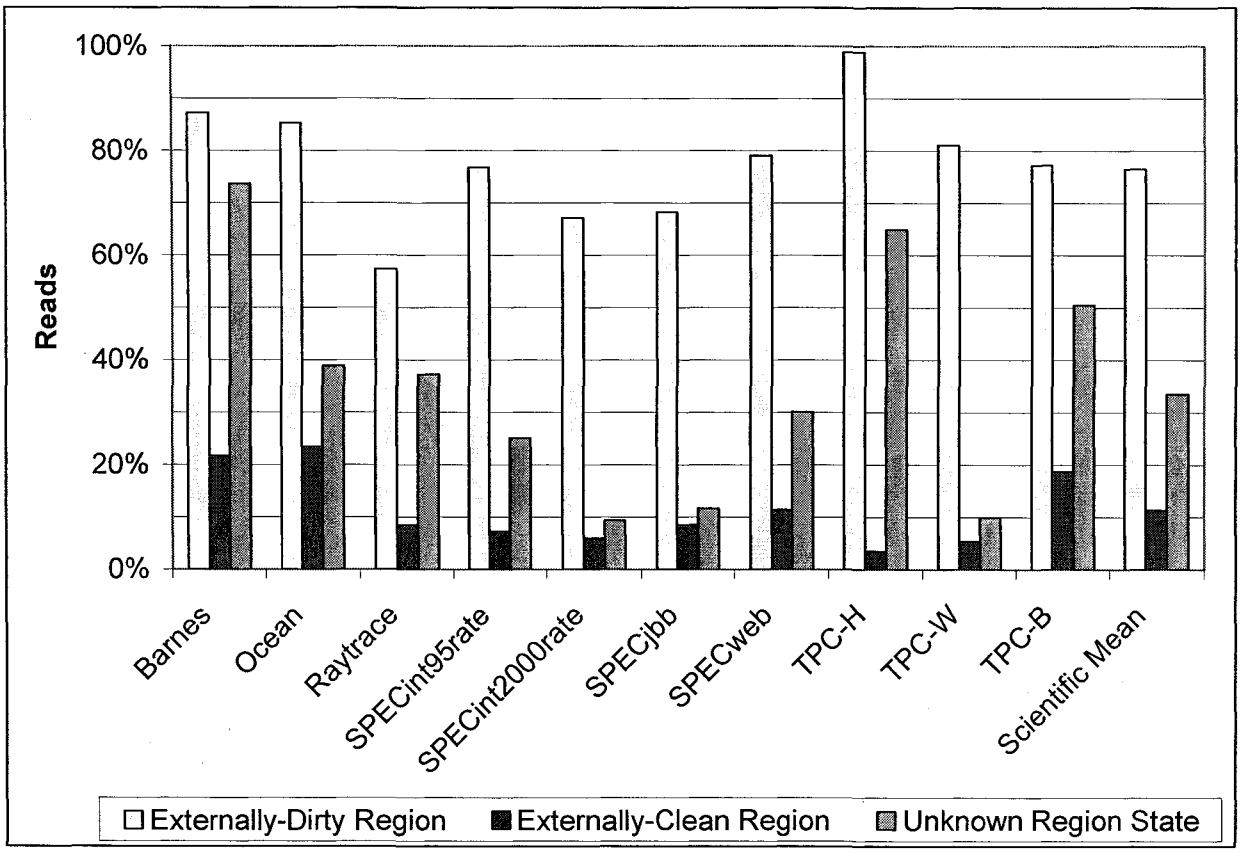

Figure 7.3: Percentage of DRAM reads that are useless for each external region state.

Over 75% of DRAM reads originating from processor requests for which the region state is externally-dirty are useless. For externally-unknown regions (misses in the Region Coherence Array) and externally-clean regions, ~35% and ~10% of the reads are useless, respectively.

7.3 Implementation

In this section I discuss the implementation of PEDS, how an RCA may be extended to improve PEDS, and the hardware overhead involved.

7.3.1 Base Implementation

First, the RCA is accessed in parallel with the low-level cache on a processor request. On a cache miss, the region state from the RCA is used to determine whether to broadcast the request to the other processors in the system, and if so, whether the data should be speculatively fetched from DRAM part-way through the broadcast snoop. Memory read requests are tagged with a bit indicating whether DRAM should be speculatively accessed.

Requests from non-shared regions are not broadcast, are nonspeculative, and can access DRAM right away. If the region state is externally-dirty, there is a high-likelihood that the data will be sourced from another processor's cache, and the read request should not speculatively access DRAM. Depending on the policy chosen, read requests from other region states may be tagged as bad candidates for a speculative DRAM access.

When a memory controller receives a read request that is a bad candidate for a speculative DRAM access, it buffers it until the combined snoop response arrives. Memory controllers already have queues to hold requests for scheduling purposes and to buffer combined snoop responses. If the combined snoop response indicates that another processor will source the data, the request is discarded, otherwise DRAM is accessed and the data is sent to the requesting processor. DRAM reads delayed unnecessarily have a higher occupancy in memory controller queues (due to the long latency of waiting for the combined response and only accessing DRAM afterward); so it may be necessary to increase the size of some of the memory controller queues to avoid contention.

7.3.2 Optimized Implementation

To minimize DRAM traffic and minimize unnecessarily delayed DRAM reads, the prediction accuracy for reads to regions with unknown external region state must be improved. If the external region state is obtained ahead of time, these reads can be more appropriately categorized as reads to externally-dirty or externally-clean regions.

First, recall that Region Coherence Arrays use a form of self-invalidation to maximize their effectiveness. In response to external requests, the RCA will invalidate a region if the processor is not caching any lines from that region. This increases the probability that the processor that initiated the request will gain exclusive access to the region. However, this self-invalidation also throws away information about the external status of the region (information that would be valuable to PEDS). In fact, for many RCA misses there is a tag match, but an invalid region due to self-invalidation.

To preserve external region state information, two states are added to the region protocol used by the RCA: *Invalid-Externally-Dirty (ID)*, and *Invalid-Externally-Clean (IC)*. These states are effectively hints that are used exclusively for PEDS, and do not affect the performance or correctness of the RCA. The first indicates that other processors may be modifying lines in the region, while the latter indicates that the region may only be clean-shared. For both of these states the processor is not caching any lines, and a broadcast must be performed to promote the region to a valid state. The ID state is entered when a region in an externally-dirty state is self-invalidated, or a region is self-invalidated by a request for a modifiable copy of a line (See Figure 7.4). The IC state is entered when a region in an externally-clean state is self-invalidated, or an exclusive region is self-invalidated by an external instruction-fetch. These states can

occasionally become stale, such that they no longer accurately reflect the external status of the region. For example, a processor may have a region in the IC state, and other processors may obtain exclusive access to the region and modify lines in it without informing the processor holding the region in the IC state.

The optimized PEDS implementation uses these new states to better predict whether speculatively accessing DRAM is useful. If a region is in an externally-dirty state (including ID), DRAM is not speculatively accessed. If the region is in an externally-clean region state (including IC), DRAM is speculatively accessed.

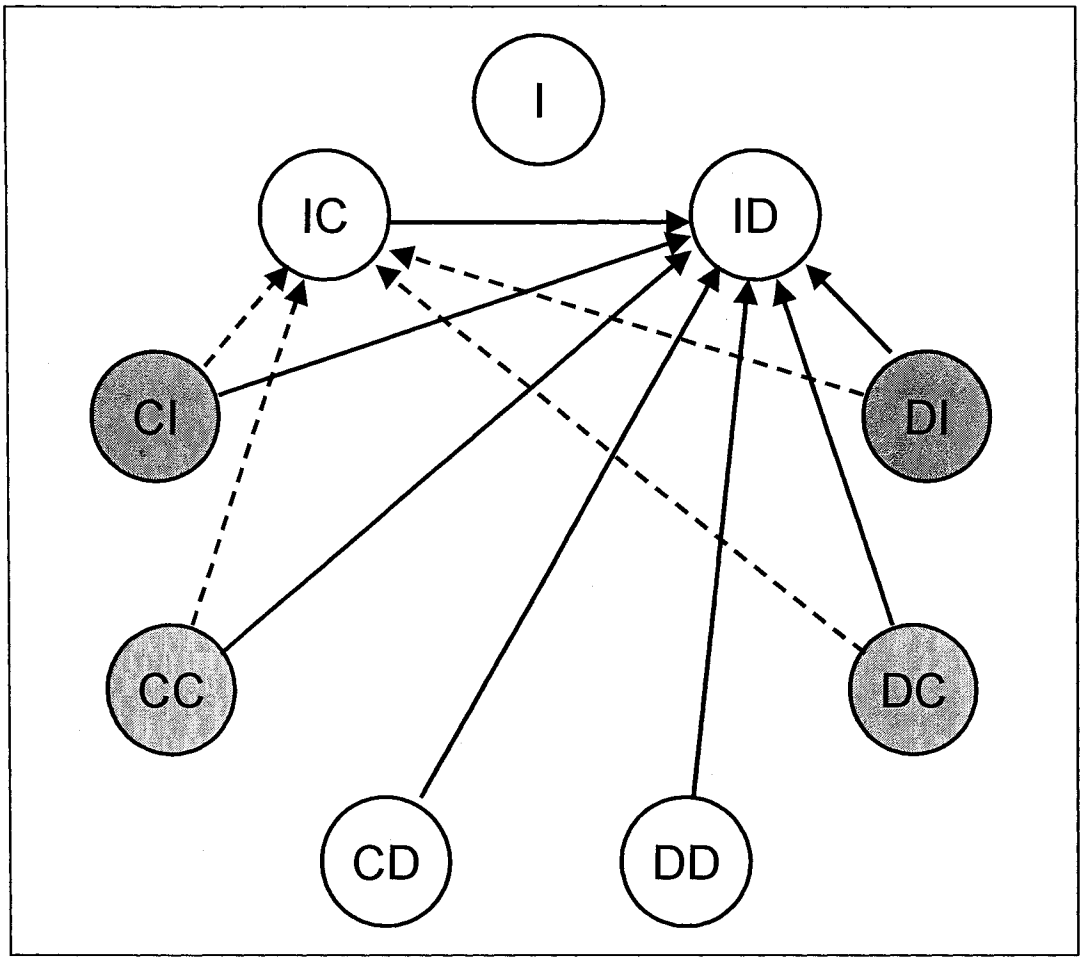

Figure 7.4: New region states for optimized implementation of PEDS.
New states, IC and ID, added to the region protocol to improve PEDS. These are pseudo-invalid states that do not affect the performance or correctness of the RCA, and are entered on self-invalidations of regions. Dashed lines indicate self-invalidations resulting from requests to obtain read permissions to the region (e.g., instruction fetches). Solid lines are writes and data reads.

7.3.3 Hardware Overhead

No additional storage in the RCA is required for the base implementation of PEDS. The only requirement is a small amount of additional logic to take the region state and generate a bit informing the memory controller whether to speculate the request. One additional bit in request

messages is required to tag memory requests as good/bad candidates for a speculative DRAM access. If request-type encodings are available in message packets, this information can be transmitted via a special memory read request type, requiring no additional bits.

The optimized implementation of PEDS requires two additional states in the region protocol, which requires an additional bit in each RCA entry for the region state. However, to avoid this PEDS could be implemented with just the ID state, separating the externally-dirty regions from the unknown ones.

In the memory controller more queue space may be needed to buffer read requests that do not speculatively access DRAM. While the memory controllers already have the necessary queues to buffer such requests, these read requests may occupy memory controller queues for longer periods of time if the prediction is incorrect. Thus more entries may be needed to avoid additional stalls from queues filling. Additionally, the memory controller may limit the number of read requests it will buffer without speculatively accessing DRAM (a high-water mark), and resume speculatively accessing DRAM for all read requests when that number is reached to allow the buffered requests to complete.

7.4 Simulation Results

7.4.1 Reduction in DRAM Reads Performed

Figure 7.5 illustrates the reduction in DRAM reads and the percentage of reads delayed unnecessarily for four different scenarios. For the first experiment, all reads speculatively access DRAM and none are delayed, achieving performance at the cost of maximal power consumption. Following this is the case where reads to externally-dirty regions do not speculatively access

DRAM, a net 20% of the DRAM reads are avoided with 6.9% of the reads unnecessarily delayed. Next is the optimized implementation of PEDS, with the ID and IC states. The optimized implementation avoids over 31% of the DRAM reads, while only delaying 0.3% reads unnecessarily. Last, we have a system with CGCT in which no reads speculatively access DRAM; here net DRAM reads are reduced 33%, and all useful reads except ones from non-shared regions are delayed (15.2%).

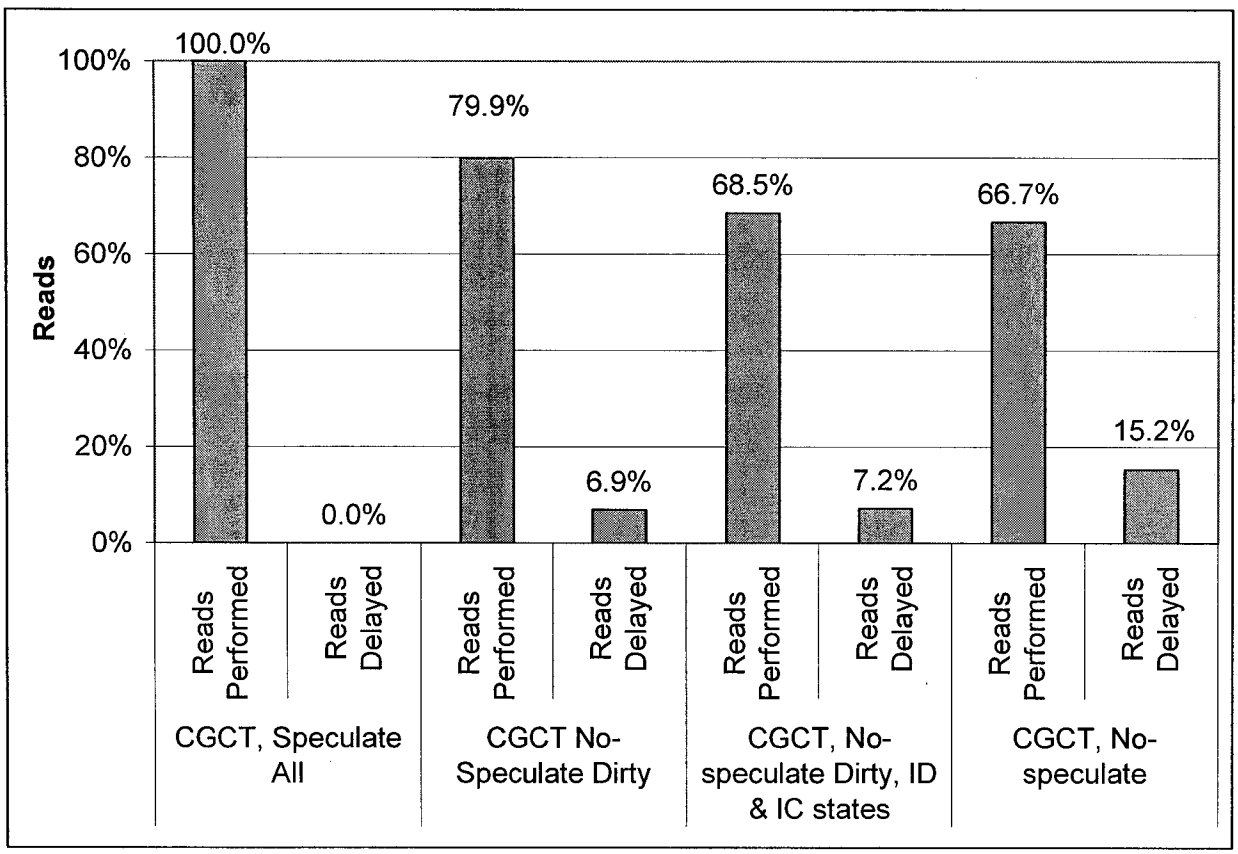

Figure 7.5: DRAM reads avoided and delayed by PEDS.

DRAM reads performed and delayed for the baseline, a system that implements CGCT with an RCA and 512B regions, a system with CGCT and oracle (perfect) speculation, and three different PEDS strategies.

7.4.2 Increased Opportunity for DRAM Power Management

Figure 7.6 illustrates the potential to keep DRAM modules in low-power modes for longer periods of time. DRAM modules do not need to be powered up for memory read requests that are satisfied by data from other processors' caches, so more power can be saved by keeping DRAM modules in low-power modes. On average, PEDS doubles the amount of time between DRAM operations. The applications that benefit most are Barnes and TPC-H. For these applications and datasets, a large percentage of read requests become cache-to-cache transfers, and once these are identified and removed, the time between DRAM requests increases 3-6 times.

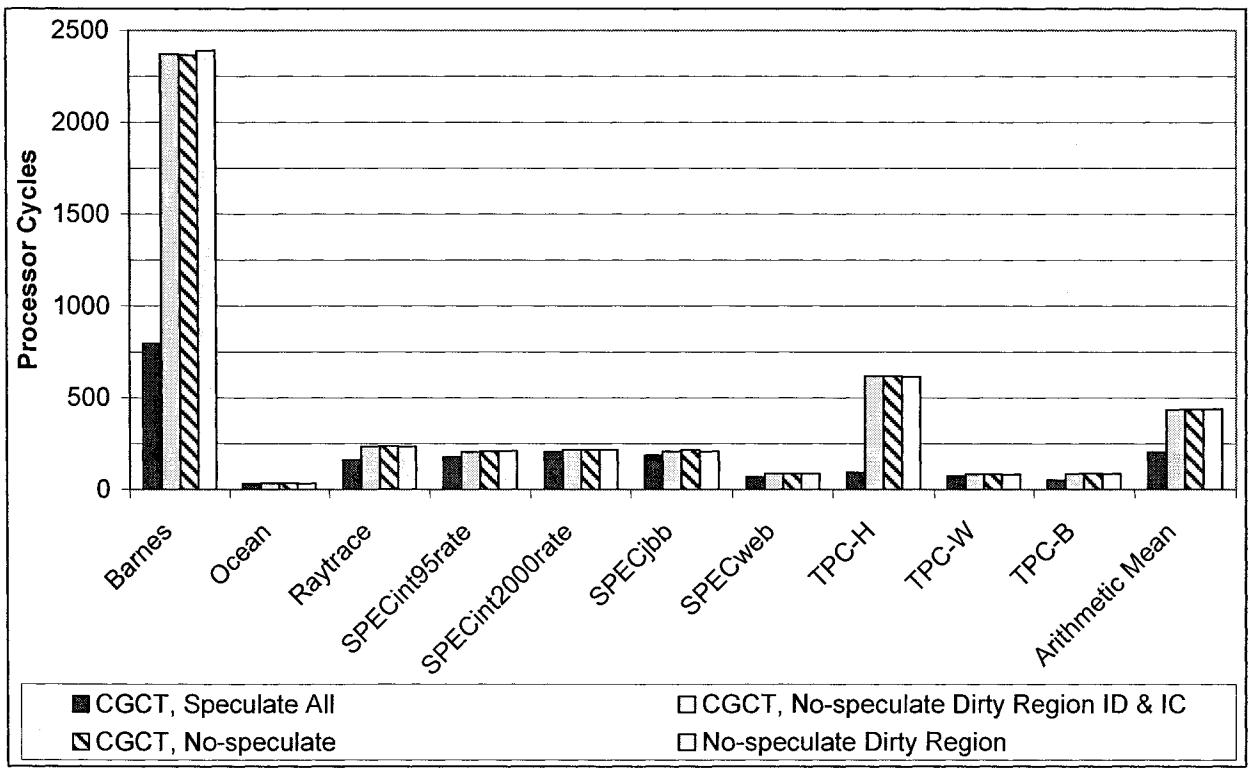

Figure 7.6: Average processor cycles between DRAM requests.

The average time between DRAM requests more than doubles from ~200 to ~450 processor cycles when read requests do not speculatively access DRAM. Subtract 100 processor cycles (66ns) for the DRAM access, and the idle time triples.

To better understand how PEDS reduces the rate of DRAM requests, we created a logarithmic distribution of the intervals between DRAM reads. For each interval, the number of reads arriving at the memory controller that amount of time after the last read was counted.

The normalized distributions are very similar in shape, only there is much less area under the curve when speculative DRAM accesses are inhibited. The behavior of DRAM reads has not changed significantly; there are simply fewer reads. There are fewer reads over any given time period except those on the far right-hand-side of the graph.

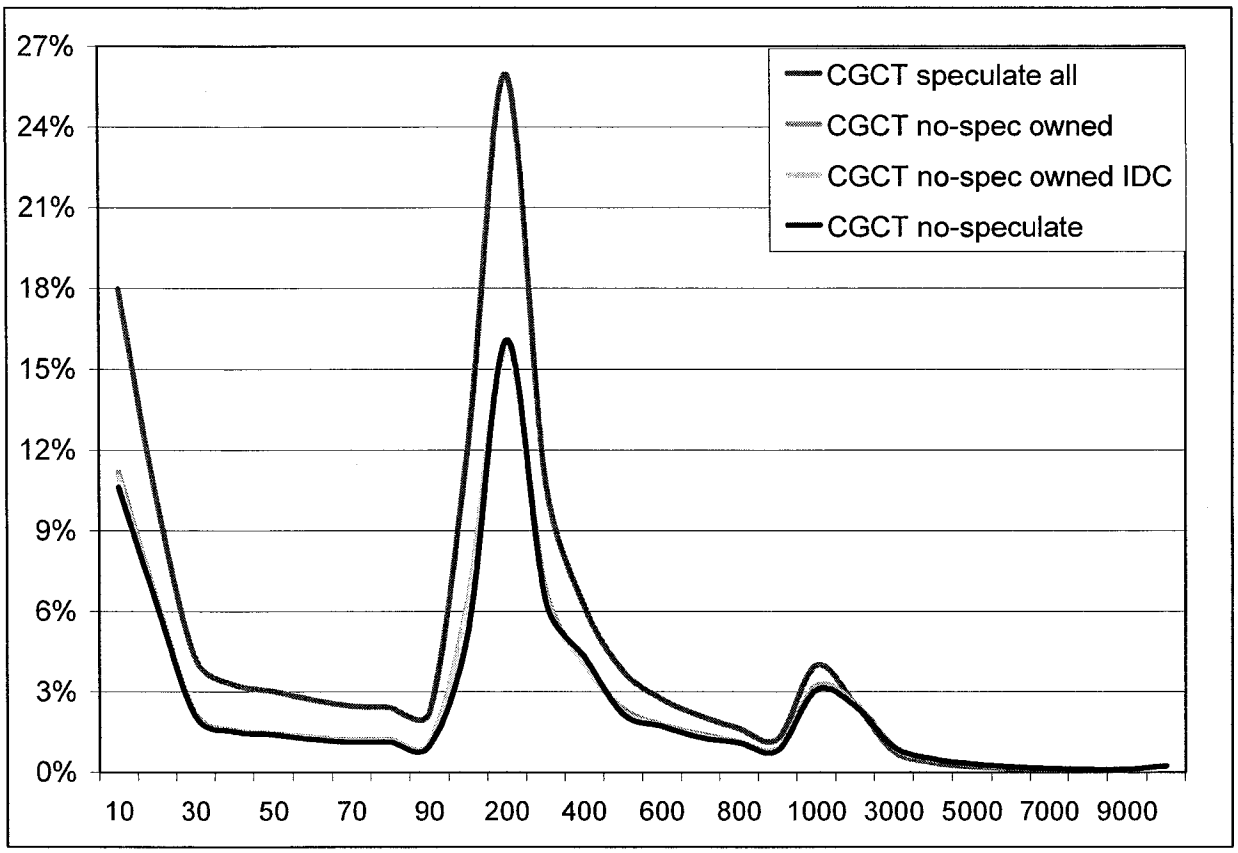

Figure 7.7: Logarithmic distribution of intervals between DRAM reads.
Based on the time period between the arrival of a DRAM read and the arrival of the last DRAM read, reads were sorted by the time-period between them, and used to generate a logarithmic distribution.

7.4.3 Effect on Run-time

Figure 7.7 shows the impact of PEDS on execution time. For each application, the execution time is increased slightly by not speculatively accessing DRAM for read requests. However, due to the improvement in execution time from CGCT, throttling DRAM speculation never increases the execution time beyond that of the baseline.

PEDS has a very small performance impact. Inhibiting speculative DRAM reads to externally-dirty regions degrades performance by less than 1% over a system with CGCT that speculatively accesses DRAM for all read requests. The degradation increases slightly as reads to externally-clean and unknown regions are delayed. The performance averaged across all of our workloads is always better than the baseline and only 2.3% worse than a system with CGCT.

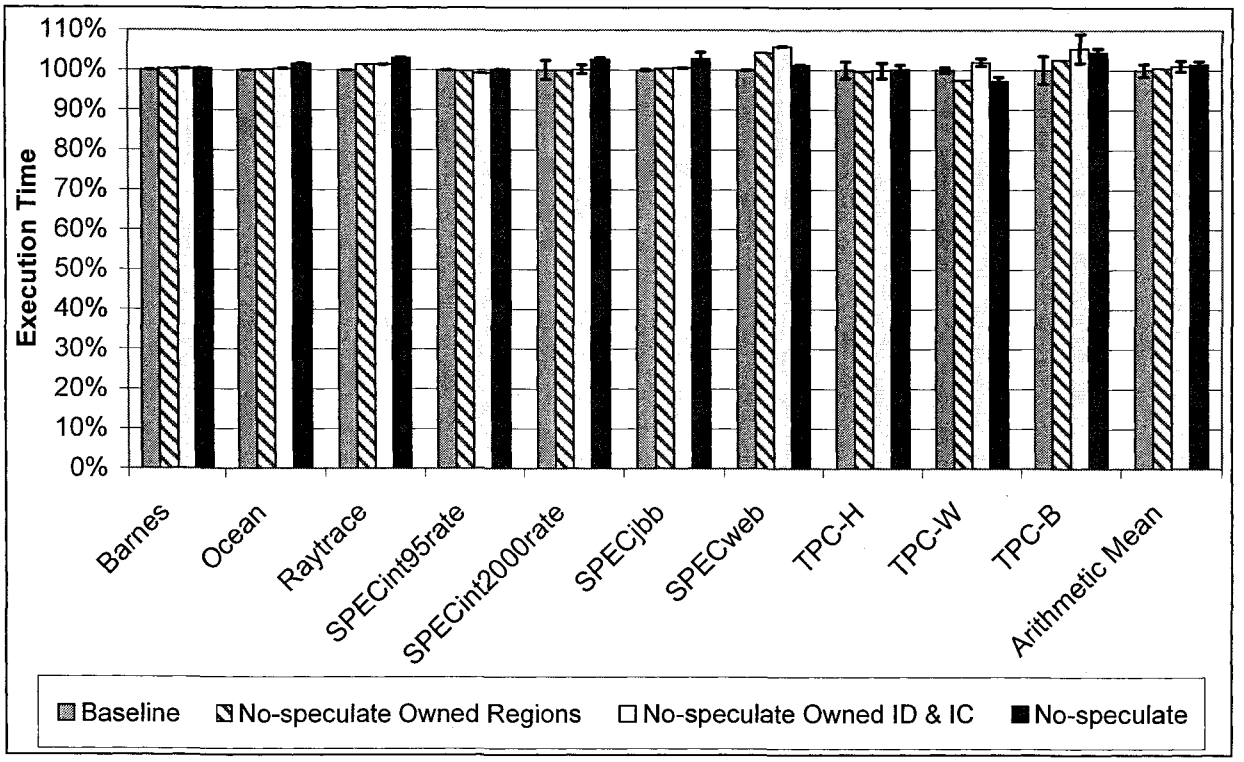

Figure 7.8: Impact of PEDS on execution time.

On average, the DRAM speculation techniques explored here do not significantly affect execution time. Overall execution time is increased 1.4% for the case of no speculative DRAM accesses, and less than 1% for PEDS. Execution time for any individual workload is increased less than 6%.

7.5 Enhancements

In times of low memory traffic, or in cases where a request can be satisfied quickly from data in a row-buffer, the memory controller can elect to access DRAM speculatively for a request anyway. This can improve performance when the power consumption is within acceptable limits, and/or if the power consumption of performing the read is low due to open-page mode operation of DRAM modules.

Though the external region state is often a good indication of whether a speculative read will be useful, this is dependent on application behavior, region size, and cache miss rates. Further, the region state may be an excellent predictor for some regions, but not others such as falsely shared regions. To improve the effectiveness of PEDS, a 1-bit or 2-bit saturating counter can be incorporated into each RCA entry to incorporate the local history of requests to the region into the prediction. This counter is updated by combined snoop responses to read requests from the processor.

Another possibility is to add more bits to read requests to give the memory controller more freedom to prioritize and speculatively access DRAM for requests. For example, while requests for externally-dirty regions are very likely to receive data from other processors, requests to regions for which the external region state is unknown have a much lower probability. Communicating this information to the memory controller can allow it to prioritize requests accordingly: requests to non-shared regions are nonspeculative and can have the highest priority; requests to regions for which the state is unknown can be speculated with a lower priority, and requests to regions in an externally-dirty state can be speculated with the lowest priority (if speculated at all).

Finally, hardware mechanisms used to switch DRAM modules to low-power modes to save power [38] can take into account the speculative nature of read requests. DRAM modules should not be powered up for a read request that has a high probability of being satisfied by data from another processor's cache, and can remain in a low-power mode at the cost of higher latency if the request must eventually acquire data from memory (a resynchronization delay is paid to power-up a DRAM module). PEDS can hence increase the opportunity for powering down DRAM modules.

7.6 Summary

This chapter proposed and evaluated Power-Efficient DRAM Speculation, a new power-saving technique utilizing Region Coherence Arrays. PEDS is very simple to implement, requiring only a bit in request messages and minor modifications in the memory controller to not speculatively access DRAM. By simply not speculatively accessing DRAM for read requests to externally-dirty regions, 20% of the DRAM reads can be eliminated, while only delaying 6.9% of the reads and degrading performance less than 1%. An optimized implementation of PEDS using additional states in the region protocol can reduce DRAM reads 30% with only a negligible increase in reads delayed unnecessarily. More compelling is the potential of PEDS to keep DRAM modules in low-power modes for longer periods of time, and not power up DRAM modules unnecessarily.

8. Future Work

This chapter briefly discusses avenues for future work in the study of CGCT techniques. The first section discusses studies that can be done with a more sophisticated simulation infrastructure (Section 8.1). I then discuss possible refinements to the CGCT implementation presented in this dissertation (Section 8.2), such as sublining the Region Coherence Array. Section 8.3 conjectures on ways that CGCT techniques might be applied to a directory-based system. Section 8.4 discusses work to be done in the area of prefetching with CGCT, including Stealth Prefetching enhancements. Finally, Section 8.5 discusses new applications of CGCT yet to be evaluated. Some of these applications have been studied previously, but can be implemented more efficiently using a Region Coherence Array.

8.1 Remaining CGCT Studies

An important avenue of future research is in the power-saving potential of CGCT. In this dissertation we have only measured performance and scalability improvements. Though the impact of reduced network activity and snoop-induced cache tag lookups on power consumption has been discussed, the net power reduction of these improvements has not been quantified, nor balanced against the additional power consumption of the Region Coherence Array. Currently, we do not have an accurate simulation infrastructure for measuring power in a complex shared-memory multiprocessor system.

8.2 CGCT Refinements

There are a number of refinements that can be made to the Region Coherence Array, its protocol, and the basic CGCT technique to enhance performance. While I have measured the potential for some of these refinements, an exhaustive study has not been completed. Five potential enhancements are discussed in this section

8.2.1 Subregions

Broadcasts remain for accesses to externally-dirty and externally-clean regions. With subregions, access permissions can be acquired at a large granularity (1KB-4KB regions), while coherence is tracked at an intermediate granularity (256B-512B subregions). Each entry in the Region Coherence Array contains one region, with state information for each subregion. This will allow CGCT to either exploit more locality with a large region while maintaining a constant level of false-sharing, or reduce false-sharing with small subregions while still exploiting locality. This will benefit all applications by allowing the region protocol to better suit data with different amounts of spatial locality. However, bits must be added to the coarse-grain snoop response for each subregion.

8.2.2 Prefetching the Region State

In the proposed implementation of CGCT, a broadcast is needed to first acquire coherence information for the region. Some potential is lost here because often the first access to a region does not need a broadcast, and simple prefetching techniques can be used to determine the state of a region early. By broadcasting a region prefetch, there may not be a reduction in average

broadcast traffic (unless there is often a subsequent broadcast for a line in a region that occurs when a region's state is still unknown). However, a region state prefetch request can be piggybacked onto a prior snoop. For example, when performing a broadcast to acquire permission to a region, information can be acquired for an adjacent region, either prefetching the other region in an aligned two-region set, or the next sequential region in memory. The prefetching can be tuned to avoid taking away exclusive access to regions from other processors. By acquiring the state early more demand requests can be sent directly to memory (reducing latency in some systems and helping the memory controller make better scheduling decisions.

8.2.3 Observing Snoop Responses from Other Processors' Requests

Some broadcast-based systems send the combined snoop response to all processors. The snoop responses of other processors' requests can indicate whether the requesting processor will get an exclusive copy of a line, and therefore whether the region state should be downgraded to an externally-dirty state. Currently, the protocol presented in this dissertation conservatively downgrades the region state to externally-dirty on an external read request, based on the assumption that the combined snoop response to external requests is not available.

8.2.4 Adapting the Region Size

In this dissertation, the region size is fixed across the entire run of the program, and across applications. However, different applications are going to benefit most from different region sizes, and in some cases a different region size may be more appropriate for a different execution phase within a single application. There is potential to adapt the region size to the application,

and possibly even to adjust the region size during the run of an application. Figure 8.2 shows the percentage of broadcasts avoided for different region sizes for each application: note that the peak of each curve is not at the same region size.

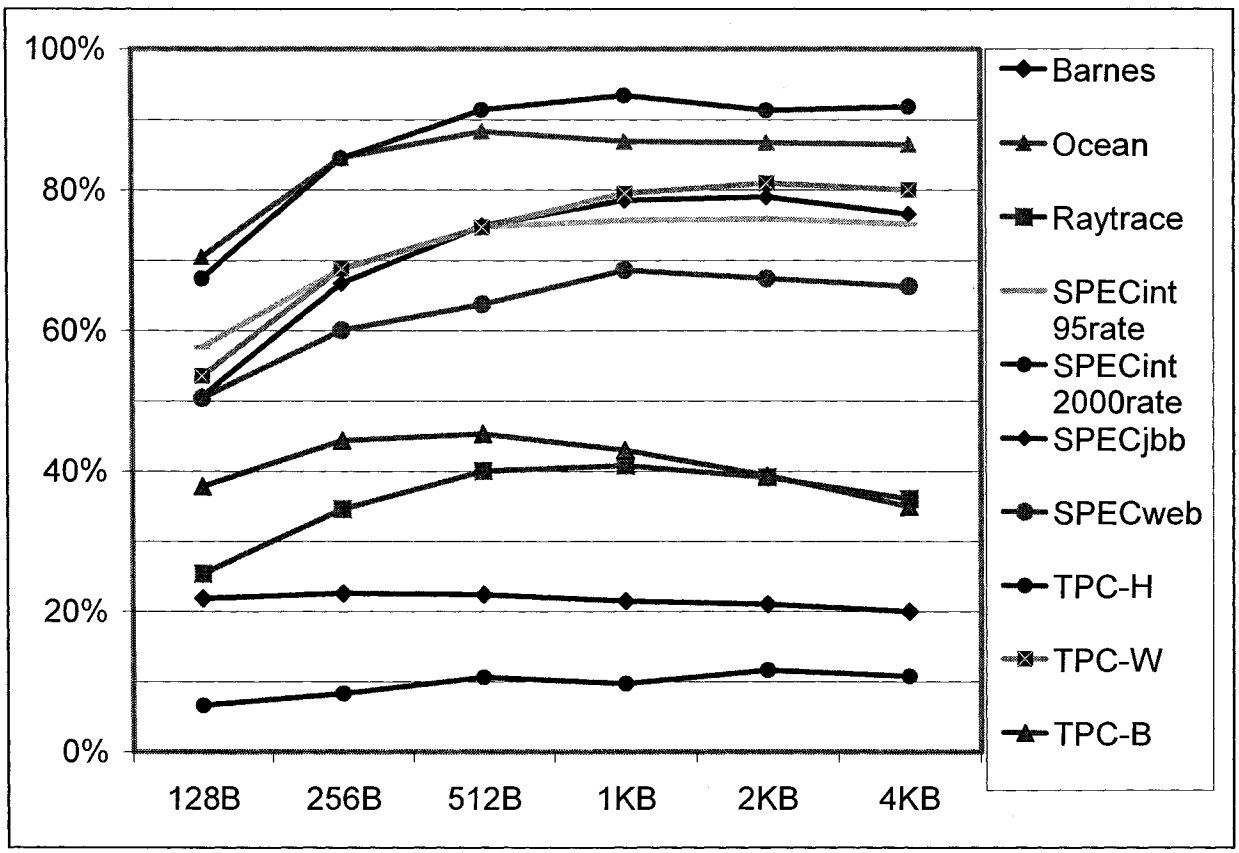

Figure 8.1: Application dependence of optimal region size

For applications such as Barnes, Raytrace, and TPC-B, the optimal region size appears to be 512B or below, whereas applications such as TPC-W, SPECjbb, and SPECint95rate benefit most from 2KB regions.

The multiprocessor system can start an execution with a region size that works well across all workloads. Then, based on the effectiveness of the region size for avoiding broadcasts, it can be increased or decreased accordingly. If a large portion of the broadcasts result from misses in the

RCA, the region size can be increased to exploit more spatial and temporal locality. If broadcasts are not avoided due to externally-dirty region states, and particularly if the region is falsely shared (i.e., broadcasts obtain data from memory, since the line is not cached by other processors), the region size can be reduced.

For implementation, the RCA can be implemented with the capability of supporting a maximum region size (including enough bits in the line count) and a minimum region size (including enough bits in the address tag, and enough entries to not constrain the cache for small regions). For simplicity, the caches can be flushed before changing the region size so that the hardware does not have to search through the cache to merge smaller regions together. To shrink the region size to one half, it is only necessary to evict the lines in the upper half of the region from the cache.

8.2.5 An Active Region Protocol

The region protocol proposed in this dissertation is entirely passive in its operation. It observes the requests made by the processor and other processors to a region and maintains a state for the region that reflects the maximum access permissions to a line held by either. This state is often conservative: (1) evictions of shared/clean lines are silent, and (2) though write-backs are sometimes broadcast, the protocol does not keep track of which lines are cached by other processors and try to deduce when the last one has been written back, and (3) the protocol does not take valid copies of data away from other processors; the region state reflects what is in other processors' caches regardless of whether that data is still in use.

There is potential to optimize more requests than those determined unnecessary in our baseline evaluations. Those evaluations did not take into account whether data was still used, and may therefore have indicated that broadcasts for the data are necessary when in fact they may be avoided. The region protocol can be made more aggressive, with the ability to invalidate lines cached by other processors, or migrate whole regions of data from one processor's cache to another.

8.3 CGCT for Directory-Based Systems

In this dissertation, CGCT has been focused primarily on the task of avoiding unnecessary broadcasts, avoiding unnecessary snoop-induced cache tag lookups, and improving scalability for broadcast-based systems. However, directory-based systems are already scalable and do not perform broadcasts unnecessarily. To apply CGCT to a directory-based system, CGCT will have to be refocused to improve intervention latency.

In a directory-based system, there is a home node for each line address, and a directory in the home node keeps a list of processors that have copies of the data. Requests are first sent to the home node, which forwards them to processors on that list. The penalty is that each request that obtains data from another cache may involve three network hops, one to the home node, a second to the processors sharing the data (sharers), and a third back to the requesting processor. Intervention latency is increased in exchange for scalability.

8.3.1 Targeting Intervention Latency

CGCT may be retargeted to improve intervention latency in directory-based systems by allowing processors to track which other processors are caching lines from a region, and sending a multicast to those processors in parallel with the request to the home node. The directory can be reorganized to summarize sharing information at the granularity of regions, and on the first request to a region it can send the list of sharers for a region back to the requesting processor for future use. The information can be kept in a Region Coherence Array, so that a list of sharers for the region is available on subsequent cache misses. To keep the list up to date, the directory can inform sharers of the region when a new processor begins sharing data and is added to the list. Alternatively, the coarse-grain sharing information can be allowed to become stale, and the directory can forward requests to processors not included in a multicast [49].

Some interconnect bandwidth is wasted communicating with processors that share the region and not the line, but the major implementation issue is ordering. If a multicast is sent to processors that share data, there is no guarantee that the request will be ordered with respect to requests from other processors. Some interconnects may have ordering properties that can be exploited, but not all directory-based systems are built with such interconnects (many are connected into meshes or tori). The directory at the home node is often the ordering point. Requests are ordered once they access and update the directory.

One possibility is to first send requests to the directory, but already tagged with a sharing list so that time is not spent accessing the directory. However, there is still a network hop to the home node. Another possibility is to try and virtually move the home node to one of the nodes sharing the region. That is, logically move management of the sharing list to a different node's

Region Coherence Array. This way, requests are first sent to one of the sharers and may be satisfied with the data they need with only two network hops. The initial requests are not sent to some remote part of the system not sharing the data. Requests to the old home can be forwarded to the new home, and the requesting processor informed of the new home so that it may record it in its Region Coherence Array. If the region is evicted from the new home's Region Coherence Array, management of the sharing list can be moved back to the old home with a message containing the current sharing list. Requests first sent to the new home can be redirected back to the old home, and the requesting processors informed of the change. Processors sharing the region eventually learn where the home node is, and, provided this does not change often, performance may be improved. Finally, to solve the ordering problem, CGCT can be combined with Token Coherence, which uses a token-passing technique to enable broadcasting in systems with unordered interconnects [61]. Using Token Coherence to resolve races and ensure correctness, CGCT can be used to determine whether to broadcast or multicast requests.

8.3.2 Stealth Prefetching for Directory-Based Systems

Stealth Prefetching might be implemented in a directory-based system without modifications to support ordered multicasts as with improving intervention latency. Stealth Prefetching only needs to know that other processors are not sharing a region, and can then go to the home node with a request for a set of lines to prefetch.

8.4 Prefetching with CGCT

Two sources of lost potential in the Stealth Prefetching study are replacements in the buffer and the additional latency of checking the buffer. Though a high percentage of lines prefetched are used, there are still improvements for larger buffer sizes, suggesting that prefetched data can be effectively inserted directly into the cache. This would increase the capacity for prefetched data, and avoid the delay of having to check a second structure after a cache miss (and before sending a request to memory).

When regions are initially prefetched, all remaining lines in the region are prefetched. There is potential to only prefetch addresses forward in memory from the lines initially touched, or to infer the direction in which to prefetch in memory based on the order in which lines are touched and only prefetch lines in that direction. This could reduce the data bandwidth consumed by Stealth Prefetching.

Finally, all memory prefetch operations are performed indiscriminately by the simulated system's memory controllers. In some cases this might overburden the memory controllers. There should be a way for the memory controller to reject or throttle prefetching at times when memory bandwidth is scarce.

8.5 Other Applications of CGCT

This section describes a few potential applications of CGCT and Region Coherence Arrays. Generally speaking, CGCT can enable optimizations that need *a priori* knowledge of the coherence status or location of cache lines in the system. Optimizations that collect or store

information about memory at a coarse-granularity can use an existing Region Coherence Array to reduce implementation overhead.

8.5.1 Improving Existing Prefetch Techniques

CGCT has the potential for enhancing existing prefetching techniques. The CGCT hardware can detect that other processors are sharing or modifying lines and can throttle prefetching of those lines. Conversely, by detecting that lines are not shared, CGCT can identify lines that can be prefetched safely and aggressively. Also, CGCT frees up interconnect bandwidth that might better be used by prefetching techniques to improve performance.

8.5.2 Improving Store Memory-Level Parallelism

As mentioned in Chapter 2, Chou, Spracklen, and Abraham proposed the Store Miss Accelerator (SMAC) to reduce the performance impact of stores that miss in the cache [42]. The SMAC is an associative array that contains information about lines recently cached by the processor. Each entry represents a 2KB region of memory and has a bit for each 64B cache line in the region that is set when an exclusive copy of the line is evicted from the cache. The bit remains set unless another processor requests the line or the entry for the region is evicted from the SMAC. On a store miss, if the corresponding region is present in the SMAC and the bit for the line is set, it is known that an exclusive copy will be obtained from memory. The store data is written to the cache early, before the rest of the line is retrieved from memory, and the store is committed to free space in the processor store queue and write buffer. The updated bytes in the cache are merged with the rest of the cache line when they arrive from memory. This technique can reduce

pressure on processor queues and reduce performance degrading processor stalls from these structures filling up, at the cost of byte-level valid bits in the cache.

This is a potential application of CGCT. The Region Coherence Array proposed in this dissertation is nearly identical in function to the SMAC. Though the implementations evaluated in this dissertation were not sublined, they can be if necessary for performance. By using an existing Region Coherence Array, the cost of improving Store-MLP can be reduced to that of the extra valid bits in the cache.

8.5.3 Optimizing Caching Policies

Also mentioned in Chapter 2, earlier work by Johnson, Hwu and Merten summarized information about cached data at a coarse granularity (called *macroblocks*), and used the information to optimize subsequent data accesses [43, 44, 45, 46]. They proposed adding a tagged hash table (the *Memory Address Table, or MAT*) to each level of the cache hierarchy, to detect and better exploit temporal and spatial locality [43]. Each entry contains saturating counters to record when cached data is reused (temporal locality), and when different bytes within a cache line are used (spatial locality). Based on these counts, levels of the cache are bypassed to avoid replacing useful data with data that has low temporal locality, and only the needed bytes are fetched from memory if little spatial locality is present. Bypassed data is placed in a small associative buffer, like a victim cache [47], allowing reuse for data that has some but not much temporal or spatial locality. This work was extended in [44], where a *Spatial Locality Detection Table (SLDT)* was proposed. The SLDT is a small associative structure that tracks spatial locality across adjacent cache lines, which is later recorded in the MAT for long term tracking. This information is used

to adjust the memory fetch size from a single cache line to multiple adjacent cache lines in a macroblock when significant spatial locality is present.

A similar technique can be implemented using CGCT, adding bits to the storage for each region to detect both spatial and temporal locality. The Region Coherence Array then can track not only the coherence status of cache lines, but their access behavior. This would require only a small amount of additional storage for the detection and a buffer for the data with low temporal locality.

8.5.4 Power and Area Optimized Memory Structures

CGCT may have applications in optimizing memory hierarchy structures for power and area. CGCT can be combined with the Decoupled Sectored Cache technique [19] to have a combined set of cache tags for regions and cache lines to reduce area and power. Also a small structure similar to a Region Coherence Array can be used to filter requests from the processor to the lower cache levels, reducing cache miss latency and avoiding unnecessary cache lookups for requests that miss in the lower cache levels at the cost of a small increase in cache hit latency.

9. Conclusions

This chapter reviews the contributions and results presented in this dissertation (Section 9.1), and presents conclusions regarding Coarse-Grain Coherence Tracking (Section 9.2).

9.1 Contributions and Results

This dissertation proposes Coarse-Grain Coherence Tracking, a new technique for optimizing coherence enforcement in broadcast-based shared memory multiprocessors. CGCT decouples the acquisition of coherence permissions from the request, transfer, and caching of data; tracking coherence status for large regions of memory to optimize the routing of requests, and filter unnecessary cache tag lookups.

An effective implementation of CGCT, Region Coherence Arrays, is proposed and evaluated. Region Coherence Arrays are shown to eliminate over 47-63% of the broadcasts in a 4-processor system running commercial, scientific, and multiprogrammed workloads. Region Coherence Arrays are also shown to filter 70-90% unnecessary snoop-induced cache tag lookups. Region Coherence Arrays significantly improve system efficiency and scalability at a manageable hardware cost. This is achieved by exploiting spatial locality beyond the cache line, and temporal locality beyond the capacity of the cache.

Region Coherence Arrays are compared qualitatively and quantitatively to RegionScout Filters, an alternative implementation of CGCT proposed concurrently by Andreas Moshovos [5, 33]. For avoiding broadcasts, Region Coherence Arrays consistently outperform RegionScout Filters with a comparable amount of hardware storage and/or region size. On the

other hand, RegionScout Filters can filter more net snoop-induced cache tag lookups for comparable amounts of hardware storage. While Region Coherence Arrays can filter more snoop-induced cache tag lookups due to their precision, cache evictions for maintaining inclusion over the cache cancels out this advantage. RegionScout filters do not constrain what data can be simultaneously cached, have no minimum size, and by virtue of not evicting cache lines to maintain inclusion they are a layered extension to an existing system.

Finally, two optimizations enabled by CGCT are proposed and evaluated. Stealth Prefetching moves data close to the processor aggressively, efficiently, and safely. It reduces execution time an average of 16% over the baseline, and approximately 9% over a system with CGCT alone. Power-Efficient DRAM Speculation is implemented, and shown to eliminate 30% of the DRAM read requests and more than double the average time between DRAM operations without hurting performance.

9.2 Coarse-Grain Coherence Tracking

CGCT helps a broadcast-based multiprocessor system achieve much of the benefit of a directory-based multiprocessor system, including low interconnect traffic and low-latency access to non-shared data. It improves scalability while maintaining low intervention latency and can free up interconnect bandwidth that might better be used for other optimizations. Furthermore, there are numerous ways to implement and optimize CGCT, allowing designers to trade off hardware costs, power consumption, complexity, and effectiveness. They can progressively incorporate CGCT into their product lines.

Perhaps most importantly, CGCT can enable many new optimizations. CGCT can be thought of as not just an optimization, but a paradigm under which to design and optimize a multiprocessor system. It can aid existing prefetchers and create possibilities for new prefetchers. It adds a new dimension to the problem of scheduling DRAM accesses. It can be included as part of a design strategy to save static and dynamic power, in the processors, the interconnection networks, and DRAM.

To conclude, CGCT is a promising new way to improve future shared-memory multi-processor systems. Hopefully this dissertation has provided the necessary first step in the investigation of Coarse-Grain Coherence Tracking for this research to continue.

Bibliography

[1] Charlesworth, A. *"The Sun Fireplane System Interconnect"*. In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001).

[2] Tendler, J., Dodson, S., and Fields, S. *"IBM e-server Power4 System Microarchitecture"*, Technical White Paper, IBM Server Group, 2001.

[3] Kalla, R., Sinharoy, B., and Tendler, J. *"IBM Power5 Chip: A Dual-Core Multithreaded Processor"*, IEEE Micro, March-April 2004.

[4] *AMD Eighth-Generation Processor Architecture*, Advanced Micro Devices, Inc. 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Hammer_architecture_WP_2.pdf.

[5] Moshovos, A., *"Exploiting Coarse-Grain Non-Shared Regions in Snoopy Coherent Multiprocessors"*. Computer Engineering Group Technical Report, University of Toronto, December 2003.

[6] Lin, W., Burger, D., *"Reducing DRAM Latencies with an Integrated Memory Hierarchy Design"*. In Proceedings of the 28th International Symposium on High-Performance Computer Architecture (HPCA), 2001.

[7] Lin, W., Burger, D., and Puzak, T., *"Filtering Superfluous Prefetches using Density Vectors"*. In Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors (ICCD), 2001.

[8] Przybylski, S., Horowitz, M., and Hennessy, J. *"Performance Tradeoffs in Cache Design"*. In Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA), 1988.

[9] Smith, A., *"Cache Evaluation and the Impact of Workload Choice"*, In Proceedings of the 12th Annual International Symposium of Computer Architecture (ISCA), 1988.

[10] Smith, A., *"Line (Block) Size Choice for CPU Caches"*, IEEE Transactions on Computers, C-36, 9, September 1987.

[11] Gee, J., Hill, M., Pnevmatikatos, D., and Smith, A., *"Cache Performance of the SPEC Benchmark Suit"*, IEEE Micro, 1993.

[12] Hill, M., and Smith, A., *Experimental Evaluation of On-Chip Microprocessor Cache Memories*. In Proceedings of the 11th Annual International Symposium of Computer Architecture (ISCA), 1984.

[13] Eggers, S., and Katz, R. "*The Effect of Sharing on the Cache and Bus Performance of Parallel Programs*". In Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1989.

[14] Eggers, S., and Katz, R. "*A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation*". In Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA), 1988.

[15] Dubnicki, C., and LeBlanc, T. "*Adjustable Block Size Coherent Caches*". In Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA), 1992.

[16] Veidenbaum, A., Tang, W., Gupta, R., Nicolau, A., and Ji, X. "*Adapting Cache Line Size to Application Behavior*". In Proceedings of the 13th Annual International Symposium on Supercomputing (ICS), 1999.

[17] Liptay, S., "Structural Aspects of the System/360 Model 85, Part II: The Cache". IBM Systems Journal, Vol. 7, pp 15-21, 1968.

[18] Rothman, J., and Smith, A., "*The Pool of Subsectors Cache Design*". In Proceedings of the 13th International Conference on Supercomputing (ICS), 1999.

[19] Seznec, A., "*Decoupled Sectored Caches: conciliating low tag implementation cost and low miss ratio*". In Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA), 1994.

[20] Kadiyala, M., and Bhuyan, L. "*A Dynamic Cache Sub-block Design to Reduce False Sharing*". In Proceedings of the International Conference on Computer Design, VLSI in Computers and Processors (ICCD), 1995.

[21] Anderson, C., and Baer, J-L. "*Design and Evaluation of a Subblock Cache Coherence Protocol for Bus-Based Multiprocessors*". Technical Report UW CSE TR 94-05-02, University of Washington, 1994.

[22] Liu, K., and King, C., "*On the Effectiveness of Sectored Caches to Reduce False-Sharing in Misses*". In Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 1997.

[23] Liu, K., and King, C., "*A Performance Study on Bounteous Transfer in Multiprocessor Caches*". The Journal of Supercomputing, 1997.

[24] Goodman, J., "*Using Cache Memory to Reduce Processor-Memory Traffic*". In Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA, 1983.

[25] Wang, H., Sun, T., and Yang, Q., "CAT – caching address tags: A Technique For Reducing Area Cost of On-Chip Caches". In the Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA), 1995.

[26] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. *"The Stanford DASH Multiprocessor"*, IEEE Computer, March 1992.

[27] Agarwal, A., Simoni, R., Horowitz, M., and Hennessy, J., "*An Evaluation of Directory Schemes for Cache Coherence.*" In Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988.

[28] Laudon, J., and Lenoski, D., *"The SGI Origin: A ccNUMA Highly Scalable Server."* In Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997.

[29] May, C., Silha, E., Simpson, R., and Warren, H. (Eds). "*The PowerPC Architecture: A Specification for a New Family of RISC Processors* (2nd Edition)". Morgan Kaufmann Publishers, Inc., 1994.

[30] Steven R. Kunkel, Personal Communication, 2004-2005.

[31] Ekman, M., Dahlgren, F., and Stenström, P. *"TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors"*. In Proceedings of the International Symposium on Low-Power Electronics Design (ISLPED), 2002.

[32] Moshovos, A., Memik, G., Falsafi, B., and Choudhary, A. "*JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers.",* In Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA), 2001.

[33] Moshovos, A., "*RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence"*. In Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA). 2005.

[34] Cantin, J., Moshovos, A., Lipasti, M., Smith, J., Falsafi, B., "*Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays*". IEEE Micro Special Issue on Top Picks from Computer Architecture Conferences, Jan-Feb 2006.

[35] Zebchuk, J., and Moshovos, A., "RegionTracker: A Case for Dual-Grain Tracking in the Memory System". Computer Engineering Group Technical Report, University of Toronto, February 2006.

[36] Wang, Z., Burger, D., McKinley, K., Reinhardt, S., and Weems, C., "*Guided Region Prefetching: A Cooperative Hardware/Software Approach*". In Proceedings of the 30th International Symposium on Computer Architecture (ISCA), 2003.

[37] Zhang, Z., Torrellas, J., "*Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching*". In Proceedings of the International Symposium on Computer Architecture (ISCA), 1995.

[38] Fan, X., Ellis, C., and Lebeck, A. "*Memory Controller Policies for DRAM Power Management*". In Proceedings of the International Symposium on Low-Power Electronics Design (ISLPED), 2001.

[39] Delaluz, V., Sivasubramaniam, A., Kendemir, M., Vijaykrishnan N., and Irwin, M. "*Scheduler-Based DRAM Energy Management*". Design Automation Conference (DAC), 2002.

[40] Hur, I., and Lin, C., "*Adaptive History-Based Memory Schedulers*", In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2004.

[41] Hur, I., and Lin, C., "*Adaptive History-Based Memory Schedulers for Modern Processors*", IEEE Micro Special Issue on Top Picks from 2005 Conferences, January/February 2006.

[42] Chou, Y., Spracklen, L., and Abraham, S., "*Store Memory-Level Parallelism Optimizations for Commercial Applications*". In Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO), 2005.

[43] Johnson, T., Hwu, W., "*Run-time Adaptive Cache Hierarchy Managements via Reference Analysis*". In Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), 1997.

[44] Johnson, T., Merten, M., and Hwu, W., "*Run-time Spatial Locality Detection and Optimization*". In Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO), 1997.

[45] Johnson, T., Connors, D., and Hwu, W., "*Run-time Adaptive Cache Management*". In Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS), 1998.

[46] Johnson, T., Connors, D., Merten, M., and Hwu, W., "*Run-time Cache Bypassing*". IEEE Transactions on Computers, 2000.

[47] Jouppi, N., "*Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*". In Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA), 1990.

[48] Martin, M., Harper, P., Sorin, D., Hill, M., and Wood, D., "*Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors*". In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), 2003.

[49] Bilir, E., Dickson, M., Hu, Y., Plakal, M., Sorin, D., Hill, M., Wood, D., "*Multicast Snooping: A New Coherence Method Using a Multicast Address Network*". In Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA), 1999.

[50] Cain, H., Lepak, K., Schwartz, B., and Lipasti, M. "*Precise and Accurate Processor Simulation*". Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads, pp. 13-22, 2002.

[51] Keller, T., Maynard, A., Simpson, R., and Bohrer, P. "*Simos-ppc Full System Simulator*". http://www.cs.utexas.edu/users/cart/simOS.

[52] "*UltraSPARC IV Processor*", User's Manual Supplement, Sun Microsystems Inc, 2004.

[53] Gharachorloo, K., Gupta, A., and Hennessy, J. "*Two Techniques to Enhance the Performance of Memory Consistency Models*". In Proceedings of the International Conference on Parallel Processing (ICPP), 1991.

[54] Alameldeen, A., Martin, M., Mauer, C., Moore, K., Xu, M., Hill, M., and Wood, D. "*Simulating a $2M Commercial Server on a $2K PC*". IEEE Computer, 2003.

[55] Cantin, J., Lipasti, M., and Smith J., "*Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking*". In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA), 2005.

[56] Kroft, D., "*Lockup-free Instruction Fetch/Prefetch Cache Organization*". In Proceedings of the 8th Annual International Symposium on Computer Architecture (ISCA), 1981.

[57] Lebeck, A., and Wood, D. "*Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors*". In Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA), 1995.

[58] Jerger, N., Hill, E., and Lipasti, M., "*Friendly Fire: Understanding the Effects of Multiprocessor Prefetching*". In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2006.

[59] IEEE Micro International Solid-State Circuits Conference (ISSCC), http://www.isscc.org/isscc/.

[60] Laudon, J., *"UltraSPARC T1: Architecture and Physical Design of a 32-threaded General Purpose CPU"*, In Proceedings of the ISSCC Multi-Core Architectures, Designs, and Implementation Challenges Forum, IEEE Micro International Solid-State Circuits Conference (ISSCC), February, 2006.

[61] Martin, M., Hill, M., and Wood, D., "*Token Coherence: Decoupling Performance and Correctness*". In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), 2003.

[62] Tang, C., "*Cache System Design in the Tightly Coupled Multiprocessor System*". In Proceedings of the AFIPS National Computer Conference, 1976: 749-753.

[63] Censier, L., and Feautrier, P., "*A New Solution to Coherence Problems in Multicache Systems*". IEEE Transactions on Computers, 27:12 (1978).

[64] Sweazy, P., and Smith A., "*A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus*". In Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA), 1986.

Appendix A. Background Information

In this appendix we provide supplemental background information on cache coherence and cache coherence mechanisms. Section A.1 briefly describes cache coherence. This is followed by a discussion of broadcast-based cache coherence (Section A.2), problems with broadcast-based cache coherence (Section A.3) and directory-based cache coherence (Section A.4).

A.1 Cache Coherence

At least as early as 1976, it was observed that attaching caches to individual processors in a multiprocessor system would violate the logical view of memory expected by programmers, unless the caches communicated to keep copies of shared data updated [62]. This logical view was first called *cache-transparency* because hardware caches are intended to be invisible to software. Later, cache-transparency was dubbed *cache coherence,* and it was stated that cache coherence is maintained if "the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address" [63]. Essentially, memory requests to a given address must appear to execute in a total order, an order which is consistent with the program order of each process, and a read operation must always return the last value written in that order.

Cache coherence is a key logical property of multiprocessor systems. It provides the illusion that all processors are accessing one large, fast memory; though the implementation is far more complex. The programmer expects that when data is written to an address in memory, subsequent reads to that address return the same data. In order to maintain this property only one

processor must be allowed to modify a given piece of data at a time, and if data is modified, copies of that data in other processors' caches must be updated or invalidated (thrown away).

A.2 Broadcast-Based Cache Coherence

Broadcast-based cache coherence protocols (also known as *snooping protocols*) maintain cache coherence by broadcasting certain memory requests to all the processors in the system. Read requests that miss in a processor's local cache are broadcast to determine if there is a cached copy elsewhere that has been modified and is more up-to-date than the data in main memory. Write requests that miss in the cache (and certain write requests that hit in the cache) are broadcast to update cached copies in other processors with new data (write-update protocols), or to invalidate them (write-invalidate protocols) because the data they contain is now stale.

In its basic implementation, broadcasting is implemented via a bus that connects the processors and memory modules. Accesses to certain addresses are broadcast on the bus, and all the other processors observe (snoop) the request and check their caches for valid data at the requested address. The other processors then respond by asserting a signal on the bus, indicating whether they have a clean or modified copy of the data. The response (snoop response) indicates to the requesting processor and memory whether other processors are sharing or modifying the data. Based on the type of request and response, the requesting processor allocates a line in the cache with an appropriate coherence protocol state, and waits for the data to be sent over the bus. Depending on whether there is a modified copy in one of the other processor's caches, the data either comes from that processor's cache (cache-to-cache transfer) or from memory.

An example of a broadcast-based coherence protocol is write-invalidate MOESI [64]. There are five basic states, Modified, Owned, Exclusive, Shared, and Invalid. *Invalid* means that the data in the cache is not valid and cannot be used; the processor must obtain new data from memory or another processor's cache. *Shared* means that the processor has a readable copy of the data, but cannot write it because other shared copies may exist. *Exclusive* means that the data has not been modified, but the processor has the only cached copy and therefore may modify it without informing other processors (this state is entered when a processor performs a read and no other processor had a cached copy). *Modified* and *Owned* indicate that the data is dirty/modified and the data in memory is stale. A cached copy enters the modified state when a write is performed, and once in this state it must be written back to memory before it can be evicted from the cache. A cached copy changes from modified to owned when another processor reads the data, indicating that other shared copies of the data may exist, but the data still needs to be written back to memory before it is evicted.

Table A.1 shows the basic MOESI state transitions. The processor can request to read or write data in a cache line, or evict the line from the cache to make room for other data. Other processors broadcast requests to obtain readable or writable copies of cache lines or to upgrade a readable copy to a writable one. Unlike read and write requests from other processors, upgrade requests do not require a data transfer; only the invalidation of any other readable copies.

Table A.1: MOESI States and State Transitions

	Processor Requests			External Requests		
	Read	Write	Evict	Read	Write	Upgrade
Invalid	Miss	Miss	-	Miss	Miss	Miss
Shared	Hit	Upgrade →Modified	→Invalid	Miss	→Invalid	→Invalid
Exclusive	Hit	Hit →Modified	→Invalid	Send Data →Shared	→Invalid	-
Owned	Hit	Upgrade →Modified	Writeback	Send Data →Shared	Send Data →Invalid	→Invalid
Modified	Hit	Hit	Writeback	Send Data →Owned	Send Data →Invalid	-

Broadcasting allows processors to find cached copies of data quickly, by communicating directly with the other processors in the system. In systems with multiple memory controllers, broadcasting is a simple way to locate the correct memory controller for the requested data. In addition, broadcasting is a simple way to order memory requests, because the broadcast network is the ordering point, and conventionally all processors observe all broadcast requests in the same order.

A.3 Problems with Broadcast-Based Cache Coherence

While broadcasting is a quick and simple way to find cached copies of data, locate the appropriate memory controllers, and order memory requests, it consumes considerable bandwidth, both in the system interconnect and the cache tag arrays. The amount of request traffic increases both with processor speed and the number of processors in the system. Broadcast bandwidth has become a limiting factor to multiprocessor system scalability and performance.

A byproduct of broadcasting is that as the system grows, the broadcast latency increases. The distance between processors grows (in both relative and absolute terms), and that increases the round-trip latency of requests through the broadcast interconnect. Hence, the time between sending a request and receiving the corresponding snoop response increases as the system is scaled up. There is more contention for the broadcast interconnect, resulting in requests being delayed.

Broadcasting also consumes considerable amounts of power, both in the broadcast network and the cache tag arrays [5, 32, 33, 34]. As the system grows to incorporate larger numbers of faster processors, the broadcast interconnect increases in size and complexity, more caches must be checked for the requested data, and each cache must perform more snoop-induced cache tag lookups.

Ultimately, broadcast traffic, memory latency, and power consumption limit the scalability of broadcast-based multiprocessor systems. To continue scaling up these systems, broadcast traffic and memory latency must be minimized, and this should be done without increasing power consumption

A.4 Directory-Based Cache Coherence: An Alternative to Broadcasting

A well known alternative to broadcast-based cache coherence is directory-based cache coherence [26, 27, 28]. Systems that implement directory-based cache coherence contain a distributed hardware table called a "directory". For each line of memory, the directory contains a list of processors sharing that line. Each processor node contains a portion of the memory and the directory information associated with it. Requests are first sent to the processor node containing

the directory for the requested line (the *home node*). The directory is accessed to obtain the list of processors sharing the line, and the request is then forwarded to the processors on that list. These processors check their caches for the requested data, and send their responses (including data) to the requesting processor. Note that three network hops are needed to obtain data from another processor's cache (unless that cache belongs to a processor in the home node), and this is called a "three-hop transfer".

Directory-based systems do not broadcast requests; they forward requests to only the processors that have the requested data. Hence, they have very low request traffic and scale to very large numbers of processors. An ordered broadcast network is not required; they can take advantage of unordered interconnects such as meshes and tori to achieve higher bandwidth. However, three-hop transfers penalize requests to shared data, requiring the request to first travel to another processor node and access the directory (which may be stored in DRAM) before going to the processors that are caching the data. Directory-based systems essentially trade latency for scalability, and hence can be disadvantageous for workloads that are sensitive to intervention latency.

Appendix B. Broadcast Protocols vs. Directory Protocols

There is an ongoing debate as to whether the systems of the future should incorporate broadcast-based cache coherence or directory-based cache coherence. Broadcast-based coherence protocols have the benefit of low intervention latency (by allowing processors to communicate directly with other processors sharing data) at the cost of high interconnect traffic. Directory-based coherence protocols have the benefits of low interconnect traffic and scalability (each request is sent only to a home node and the subset of processors sharing the data). However, directory-based protocols penalize intervention latency by first sending requests to the home node, possibly requiring three network hops to obtain data from another processor's cache. The relative performance of these two types of systems depends on the relative importance of interconnect bandwidth and intervention latency.

Most agree that for small general purpose systems with 2-8 processors a broadcast-based cache coherence protocol is a simple and feasible solution. Most also agree that to scale to hundreds or thousands of processors, a directory-based cache coherence protocol is the only feasible solution. The debate centers on the mid-range, for multiprocessor systems with approximately 16 to 64 processors that are commonly used for commercial applications such as transaction processing and decision support systems. Directory-based system supporters will recommend a directory-based system, citing scalability, cost, and low-latency access to non-shared data. Broadcast-based system enthusiasts will point to commercially available broadcast-based systems with 64 or more processors, the high amount of sharing in commercial workloads, and recent advances in the scalability of broadcast-based systems.

There are academic studies supporting both points of view, but broadcast-based systems and directory-based systems are difficult to fairly and directly compare; there are different interconnection topologies, different protocols, and different sets of optimizations that can be employed. The debate will likely be played out in the market, where teams from different companies work hard to make the best possible system with available resources and real time constraints. The result is uncertain –in the future it may not be possible to support the bandwidth requirements of a large number of processors with a broadcast protocol, or it may be that directory protocols will only find a secure place in large-scale scientific systems.