ABSTRACT

KUEBEL, ROBERT MARTIN. The Performance of Token Coherence on Scientific Workloads. (Under the direction of Dr. Gregory T. Byrd).

Broadcast snooping and directory protocols are, by far, the most common coherence protocols in research and commercial systems. These protocols represent two extremes of cache coherence protocol design with seemingly incompatible goals. Directory protocols produce scalable systems by reducing network bandwidth requirements at the cost of increasing latency. Snooping based systems allow low latency at the cost of increased bandwidth. Recently, a promising class of coherence protocols called Token Coherence has been shown to outperform directory and snooping protocols by attempting to combine the best characteristics of both protocols. The concept of token counting allows the protocol to safely multicast requests on an unordered network. This avoids indirection like a snooping system but allows the system to scale by eliminating the need for an ordered network. Additionally, Token Coherence promises to be easier to implement, requiring nothing more than reliable message delivery from the network, and provides a simple set of rules to guarantee correctness.

Token Coherence was developed to improve the performance of multiprocessors running "commercial" applications including web and database servers. The fact that token coherence was designed with a specific class of applications in mind raises questions about its ability to perform under different circumstances. Without a more thorough investigation of the performance of Token Coherence, it is unclear whether its success on commercial applications is representative of its performance on other workloads. The goal of this thesis is to evaluate the performance of Token Coherence using a subset of the Splash2 benchmark suite. Also, variations of Token Coherence described in the literature but whose effects on performance were not published are examined.

This work shows that Token Coherence is not dependent on the peculiarities of commercial workloads and can improve the performance of scientific applications. In fact, Token Coherence performs well despite that assumptions under which it was conceived are not necessarily true on all applications. In addition, some optimizations made to Token Coherence specifically for commercial workloads do not have a significant positive benefit for scientific workloads.

The Performance of Token Coherence on Scientific Workloads

by

Robert M. Kuebel

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2005

Approved By:

Dr. Yan Solihin	Dr. Vincent W. Freeh

Dr. Gregory T. Byrd
Chair of Advisory Committee

Biography

Robert M. Kuebel was born in Akron, Ohio. He received a B.S. in Computer Engineering from University of Cincinnati in 2003. Currently, he resides in Raleigh, North Carolina with his wife Kari.

Acknowledgements

I would like to thank the members of my Advisory Committee, especially Dr. Byrd, for their guidance during this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For any particular problem, there may be countless solutions that provide adequate results. Unfortunately, solutions that provide satisfactory results over the set of all problems are rare. Consider the dozens of sorting algorithms. In terms of the number of comparisons, a particular algorithm may perform quite well for a given input, but drastically worse for another. Quicksort, for example, executes $O(n \log n)$ comparisons on average, but $O(n^2)$ in the worst case. In the case of coherence protocols in Non-Uniform Memory Access (NUMA) architectures, there have been few protocols that provide reasonable performance on a wide variety of workloads.

Broadcast snooping [3, 7] and directory protocols [11, 13, 22] are, by far, the most common coherence protocols in research and commercial systems. Both types of protocols have been heavily researched and many optimizations [8, 24] and hybrids [20] have been proposed. Research has also generated alternative protocols [6, 19] that have been shown to outperform directory and traditional snooping protocols. Unfortunately, these alternatives require specialized networks that, in practice, may be more difficult to implement than the relatively simple network requirements of directory and snooping protocols. Recently, a promising class of coherence protocols called Token Coherence [16, 18] has been shown to outperform directory and snooping protocols. Additionally, Token Coherence promises to be easier to implement. It requires nothing more than reliable message delivery from the network and provides a simple set of rules to guarantee correctness.

Token Coherence was developed to improve the performance of multiprocessors

running "commercial" applications including web and database servers. The designers took advantage of the behavior of commercial applications—namely, coherence races occur infrequently [18] and cache-to-cache transfers are frequent [5]. This allows the protocol to make trade-offs that reduce the cost of non-racing messages even at the increased cost of race detection and handling. Other aspects of commercial workloads may favor Token Coherence. For example, commercial applications typically exploit thread level parallelism. Servers often handle each new request in a separate, independent thread. These threads are only required to synchronize with other threads when accessing non-static shared data. Finally, when the data being served is static or changes to that data are rare, each thread can cache a copy of the data without suffering frequent invalidations. Token Coherence[1] has been shown [18] to significantly improve the performance of commercial applications on next-generation hardware, but results for other workloads have not been published.

The fact that Token Coherence was designed with a specific class of applications in mind raises questions about its ability to perform under different circumstances. Does the performance of Token Coherence depend on the features exhibited by commercial applications? Does the assumption that coherence races are rare break down under different workloads? If so, what is the impact on performance?

Traditionally, researchers of parallel architectures have used "scientific" workloads to measure the impact of proposed changes. These applications usually include technical computing tasks such as fluid dynamics, N-body problems and matrix manipulations [27]. The characteristics of scientific applications are quite different from commercial applications and potentially contain behaviors detrimental to the performance of Token Coherence. The majority of scientific applications rely on data parallelism to achieve scalability. Another important difference between these two classes of benchmarks is the frequency of different sharing patterns. For example, a database server may exhibit migratory data frequently, while a image rendering program may tend to have more widely shared data. Because of their differences from commercial workloads, scientific applications are the focus of this evaluation.

Without a more thorough investigation of the performance of Token Coherence, it is unclear whether its success on commercial applications is representative of its performance on other workloads. The goal of this thesis is to evaluate the performance of Token

[1]Token Coherence is a class of coherence protocols, not a particular implementation. Throughout this thesis, "a protocol that implements Token Coherence" will be shortened to "Token Coherence."

Coherence using a subset of the Splash2 benchmark suite. Also, variations of Token Coherence described in the literature but whose effects on performance were not published will be examined.

This work will show that, for the chosen parameters, Token Coherence is not dependent on the peculiarities of commercial workloads and can improve the performance of scientific applications. In fact, Token Coherence performs well despite that the assumption that coherence races are rare does not hold for the studied applications. However, some optimizations made to Token Coherence specifically for commercial workloads (a migratory data optimization and "persistent read requests") do not have a significant positive benefit for scientific workloads.

The remainder of this work will be divided as follows. Chapter 2 describes existing coherence protocols and the motivation for Token Coherence. Chapter 3 describes the methods used to simulate the baseline system and three variations of Token Coherence. Chapter 4 gives results and analysis. Finally, Chapter 5 summarizes this work.

Chapter 2

Prior Work

The ideal coherence protocol is one that uses nearly zero bandwidth (or more practically, no more bandwidth than necessary) and has near zero latency. Of course, no real protocol can achieve even one of these goals, let alone both. Snooping coherence and directory coherence represent the two extremes: each approaches the ideal protocol in one aspect. Snooping coherence is known for low latency and directory coherence uses little bandwidth. Unfortunately, approaching one ideal trait comes at a cost—sacrificing ideality in the other. This is shown graphically in figure 2.1. The curve between Directory and Snooping represents a line of equal performance[1] and the origin represents the ideal protocol. The performance characteristics of a system are not the only concern of designers. A manageable level of complexity must be maintained for a systems to be designed and verified in reasonable amount of time. If the effort of design and verification of all components of a coherence system could be quantified with a single number called "complexity," the ideal protocol would have near zero complexity. Some protocols depend on certain network characteristics (e.g. total ordering of messages) to operate correctly. Therefore, required network features will be the fourth dimension for comparison in this work. Table 2.1 gives a relative measure of these four characteristics for the protocols discussed here.

These characteristics are used to compare existing protocols and show how research has progressed toward a ideal protocol. The drawbacks of common protocols are discussed along with some previous attempts at overcoming those limitations. Two recent proposals

[1] Ignore for the moment differences in network topology and bandwidth.

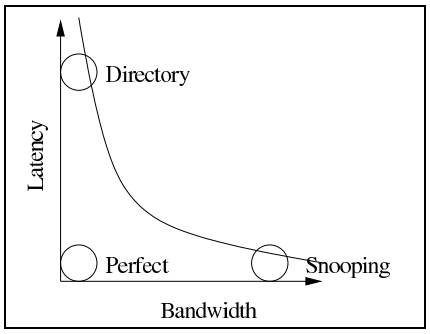


Figure 2.1: The latency-bandwidth trade-off in coherence protocols.

Attribute	Directory	Snooping	BASH	Multicast Snooping	Token Coherence[2]
Latency	High	Low	Low	Low	Low
Bandwidth	Low	High	Fairly High	Moderate	Moderate
Protocol Complexity	Moderate	Low	High (2 protocols)	Low	Low
Network	Any	Ordered	Ordered	Ordered	Any

Table 2.1: Requirements of five coherence protocols

most directly related to this work are Bandwidth Adaptive Snooping [20] and Multicast Snooping [6]. Each attempts to combine the best features of snooping and directory coherence. Token Coherence is the most recent attempt at achieving a "best of both worlds" protocol. The concept of Token Coherence and three implementations are also described in this chapter.

2.1 Snooping and Directory Coherence

Snooping coherence and directory coherence are, by far, the most commonly implemented coherence protocols. In some ways, these two schemes are duals. Snooping coherence provides low latency responses (in terms of the number of network traversals) at the cost of bandwidth, while directory coherence uses much less bandwidth at the cost of higher latency. Because of these trade-offs, snooping coherence is dominant in small scale systems and directory coherence excels in large systems where bandwidth is more precious. This section discusses the advantages and drawbacks inherent to each of these protocols.

[2]TokenM, a multicast form of Token Coherence, is depicted here.

2.1.1 Snooping Coherence

A memory system is said to be "coherent" if a read to any memory location always returns the value of the last write to that location. Therefore, to maintain coherence, every coherence protocol must provide a method to determine the order of transactions for each memory block. In snooping coherence systems, the ordering point is a totally-ordered network such as a shared-wire bus or broadcast tree. In the simple case, the network is a single access bus, allowing no overlapping of transactions. Every coherence message must be received, in the same order, by every processor in the system. Therefore, as the number of processors increases, the rate of coherence messages and the bandwidth requirement of the bus also increase and the bus quickly becomes a bottleneck. Scaling a snooping network to a large number of processors is difficult for two reasons: (1) the physically longer wires and the capacitance of each device on the bus add resistance to the high-frequency signals needed for inter-processor communication and (2) the bandwidth requirement grows quickly as processors are added.

Despite being bandwidth hungry, snooping coherence provides two major advantages. First, it has the ideal property of allowing any request to be satisfied in two traversals[3] of the network. The first traversal broadcasts the request to all nodes. The second allows the node designated by the protocol to provide a response such as a block of data. Also, no explicit acknowledgments are needed because the network ensures that every processor sees each request. For example, a processor issuing a ReadX request can be sure all other processors have eliminated their copies of the requested block as soon as the ReadX is observed on the bus. In contrast, directory systems often require each sharer to send an InvalAck message to the requestor. The requestor cannot assume all copies of a block have been eliminated until all acknowledgments have been collected. The second advantage of snooping coherence is design simplicity. Humans find it much easier to reason about a totally-ordered system than unsynchronized systems.

Several methods have been proposed to cope with bandwidth requirements of a single shared bus. Systems like the Sun E10000 [7] use multiple pipelined, ordered address buses for requests and a separate, unordered data response network. This eases the burden on the single broadcast bus in two ways. First, large data packets do not travel on the same network as the smaller request packets. Since the requests are totally ordered by the address

[3]Some transactions, such as a write-back to memory, may only require one bus transaction.

bus, the cache controllers can reconstruct the order of data responses and thus maintain coherence. Second, the single address bus is split into four physical buses. Each handles a distinct segment of the address space and allows multiple requests to execute in parallel.

2.1.2 Directory-Based Coherence

Directory-based systems distribute the task of ordering transactions. Instead of a single ordering point, main memory is distributed among all nodes in the system. Each node tracks the state of its assigned blocks in a per-node "directory." The directory contains, for each block, state information similar to cache state information and a list of sharers. When a processor requires access to a block, a request is forwarded to the "home" node (the node to which the desired block is assigned). Upon receiving the request, the home node can respond with data if the block at the directory is up to date. If the block is dirty (i.e. there exists a cached copy that has been written since memory was last updated), the request must be forwarded to the current owner. This indirection occurs quite frequently and is the source of directory coherence's additional latency.

The cost of looking up a block's state at the directory is often recovered by the reduction of contention at various bandwidth limited resources (e.g. network links and L2 cache tags). The directory is required to know exactly which nodes in the system must observe any particular transaction. Therefore, by forwarding requests to only those nodes, any transaction will use the smallest amount of bandwidth possible. Nodes that are not concerned with a transaction never receive notice that is has occurred. In addition to conserving bandwidth, directory systems allow multiple transactions to take place simultaneously. Each directory acts independently and in parallel with others. This allows directory systems to scale to much larger configurations than snooping coherence systems.

Directory systems are generally not restricted to a specific network topology. Advanced directory protocols, such as the SGI Origin 2000 protocol [11], make no assumptions about the ordering of messages, while systems like the AlphaServer GS320 [10] exploit a more restrictive network model to eliminate the race cases that exist in unordered networks. Most directory-based systems are implemented using a more general interconnect such as a 2D grid or torus. The difficulty of determining transaction order, avoiding deadlock and starvation and still providing correctness on such a network complicates directory protocols. Therefore, snooping systems tend to be easier to design and verify.

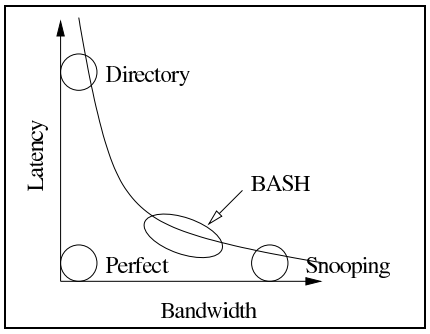


Figure 2.2: The latency-bandwidth trade-off in BASH.

2.2 Bandwidth Adaptive Snooping

Martin, et al. saw an opportunity for a single protocol that could dynamically adjust its bandwidth usage to achieve the best performance. The result was the Bandwidth Adaptive Snooping Hybrid (BASH) protocol [20], that combines a snooping system and directory system. The goal is to create a single protocol that can scale from small systems to very large systems. It has been shown [20] that a snooping protocol can outperform a directory protocol when bandwidth is plentiful. The reverse is true when bandwidth is limited. Typically measures of bandwidth (e.g. cross sectional bandwidth) of a machine grow with the number of processors. However, as the number of processors increases, the average number of packets passing through each network link also increases. Eventually, the amount of bandwidth available to each processor begins to diminish. For this reason, bandwidth is a more precious commodity in larger systems. BASH constantly adjusts its network bandwidth usage to use most of the available bandwidth without saturating the network. This allows BASH to perform as well as either a snooping or directory protocol under a wide range of program behaviors and machine sizes.

The default mode of BASH is to handle requests like a snooping system: all requests are broadcast. As network contention increases, the protocol probabilistically increases the number of messages sent using the built-in directory protocol. Migrating to a directory based protocol when the network is congested can increase performance because these protocols use mostly unicast messages. Unicasting instead of broadcasting like a snooping protocol allows more requests to traverse the network in parallel, thus giving a boost to performance. Also, a network with fewer messages in-flight is likely to be less

congested and thus have a lower end-to-end latency. A simplified directory at each node handles the unicast requests. The advantage of BASH is its ability to tune itself to the "sweet-spot" on the latency-bandwidth tradeoff curve. Figure 2.2 shows the region of the latency-bandwidth curve favored by BASH.

The adaptation mechanism BASH uses is simple. Each node in the system measures the utilization of its own network link(s) as an estimate of the total network utilization. The more often the local link's utilization is greater than a given threshold, the more likely the next message sent to the network will be a unicast according to the directory protocol. The decision to broadcast or unicast is made on a message by message basis.

BASH modifies and combines the Sun E10000 (snooping) and the AlphaServer GS320 (directory) protocols. These systems and the hybrid take advantage of a totally ordered interconnect to maintain coherence. While BASH still requires an ordered network, protocol design is more complex because of the need for two protocols: one broadcast snooping and one directory.

2.3 Multicast Snooping

Multicast snooping is an attempt to combine the low latency of a broadcast snooping system with the scalability of a directory based system. The protocol [6] takes advantage of the fact that most coherence transactions require only a small fraction of processors in the system to be involved. Consider the bandwidth versus latency trade-off made in a directory based system with explicit acknowledgments. During a ReadX operation in a directory based machine, the requestor must access the home directory before it knows which processors require invalidations. This conserves bandwidth by sending messages to only the necessary processors, but accessing the directory to determine the exact set of invalidations to send increases latency. In a broadcast snooping system, the directory is not needed because all processors see each transaction and each can decide locally whether it needs to act upon that transaction. This reduces latency, but increases required bandwidth because each transaction must be seen by all processors. If the processors involved could be determined by the requestor, a system with latency similar to a snooping system and scalability like a directory-based machine could be constructed. Figure 2.3 shows Multicast Snooping's operating point relative to directory and snooping in terms of latency and

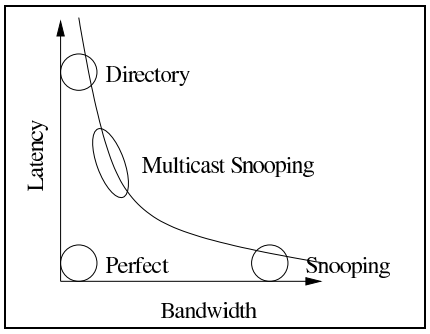


Figure 2.3: The latency-bandwidth trade-off in Multicast Snooping.

bandwidth.

The Multicast Snooping protocol behaves much like a snooping protocol with a few exceptions. The protocol defines a message to get a shared copy of a memory location, GETS, and a message to get an exclusive copy of location, GETX. Each request includes a set of processors to which the message must be relayed. A simplified directory at memory will acknowledge transactions positively if the predicted set of processors is sufficient. Otherwise a partial or complete failure acknowledgment message will be sent to the requestor. Failed requests are retried with the set of processors suggested by the failure acknowledgment. To avoid deadlock, failed transactions are sent using a broadcast after n attempts. To succeed, the destination set of GETX messages must contain at least all current sharers and the current owner. GETS messages must be received by at least the current owner for success.

The key to multicast snooping is correctly predicting the set of processors that need to be involved with a certain transaction, the multicast "mask." The predictor used in Multicast Snooping is called StickySpatial(k). Each processor maintains a table direct-mapped to the machine's address space. Each entry in the table contains a tag, multicast mask and a last invalidator field. A memory block B corresponds to the predictor index B$'$, which is equal to B modulo the number of indices in the table. When a request is completed successfully, memory or the owner of a block returns the current multicast mask. This mask contains exactly the set of sharers. If the tag at B$'$ corresponds to the address of the completed request, the multicast mask at B$'$ is logically OR'ed with the returned mask. Otherwise the mask is set to the returned mask and the tag is updated. When an invalidate is received by a processor, the last invalidator field is set to the requestor.

When a processor sends a GETX B message, the destination set is calculated by logically OR'ing the requestor, directory, the multicast masks at indices B′ − k, ... , B−1, B′, B′ + 1, ... , B′ + k. When a GETS B message is sent, only the requestor, directory and the last invalidator field at B′ are OR'ed. Merging the destination sets of blocks with in the radius k attempts to exploit spacial locality. That is, if a block's mask is insufficient to complete a request, the neighboring blocks may contain related data and therefore have the correct bits set in the mask.

Multicast Snooping's bandwidth conservation allows it to outperform broadcast snooping. Also, Multicast Snooping's avoidance of indirection allows it to outperform directory coherence. The disadvantage of Multicast Snooping is its reliance on a complex, network topology similar to, but less restrictive than, Isotach [23] networks. Multicast Snooping requires a total logical order of request messages. To meet this requirement, a k-ary fat-tree with N root nodes, arbitrarily ordered uplinks and totally ordered downlinks is used. A logically separate, unordered data network supplies responses.

2.4 Token Coherence

Token Coherence [16], like BASH and Multicast Snooping, is an attempt to create a "best of both worlds" protocol by combining the best features of snooping and directory coherence while avoiding their drawbacks. The key observation in the development of Token Coherence was that coherence races occur infrequently. In the commercial workloads studied in [18], an average of 3% of messages suffered interference from racing messages. This insight allows Token Coherence to make design choices that favor non-racing messages, despite having a substantial negative impact on race detection and handling. Typically, because race detection is applied to every transaction, a complex protocol or restrictive network is required. Token Coherence requires neither.

Token Coherence was developed to increase the performance of commercial applications such as web and database servers on medium scale systems. Technology trends indicate that medium scale systems, common among commercial users, in the near future will enjoy highly integrated nodes with L1 and L2 caches, cache controllers, memory controllers and network interface on the processor die much like the Alpha 21364 [22]. Often, these highly integrated nodes employ a mesh network with multiple links per node. There-

fore, systems with structure similar to the AlphaServer GS1280 [9] or the IBM BlueGene/L [1] are the target of Token Coherence.

Token coherence layers a "performance protocol," which optimistically performs transactions as if no races could occur, over a "correctness substrate," which detects and corrects races and starvation in the performance protocol. The idea is to allow the fast (but not necessarily correct) performance protocol to be backed by a correct (but not necessarily fast) correctness protocol that guarantees forward progress. The two protocols are independent of each other. Thus, Token Coherence is said to decouple performance and correctness. Traditionally, one protocol has met both objectives. The advantage is that different sub-protocols can be used in different machine configurations. Three performance protocols (TokenB, TokenD and TokenM [16]) are discussed in this work. TokenB is a broadcast protocol intended for use in systems were bandwidth is plentiful. TokenD is similar to a directory system and can be used were bandwidth is less abundant and the increased latency of the directory lookup can be tolerated. TokenM is a compromise between the two and attempts to avoid indirection at the directory and avoid broadcasting by using destination set prediction similar to Multicast Snooping.

The decoupling of performance and correctness is made possible by the token counting concept. Each memory block in the system has an associated set of T tokens, where T is at least the number of processors in the system. One token is designated as the "owner token" which may be "clean" or "dirty". Tokens assigned to one block are never used with another block and tokens are never created or destroyed, only passed between processors or between processors and main memory. Each cache line in the system contains a tag, valid-data bit, $\lceil \log_2 T \rceil$ bits to hold the number of tokens and two additional bits to designate whether or not the owner token is held and whether or not it is dirty. Since tokens can travel without data, the valid-data bit is needed to determine whether the data stored in the line can be used. The valid-data bit is set when a message arrives with data and at least one token. The bit is cleared when the block contains zero tokens. A clean owner token indicates that the copy of the block in main memory is up-to-date. Therefore, the owner token is set to DIRTY when a block is written. When memory holds the owner token, the token is set to CLEAN.

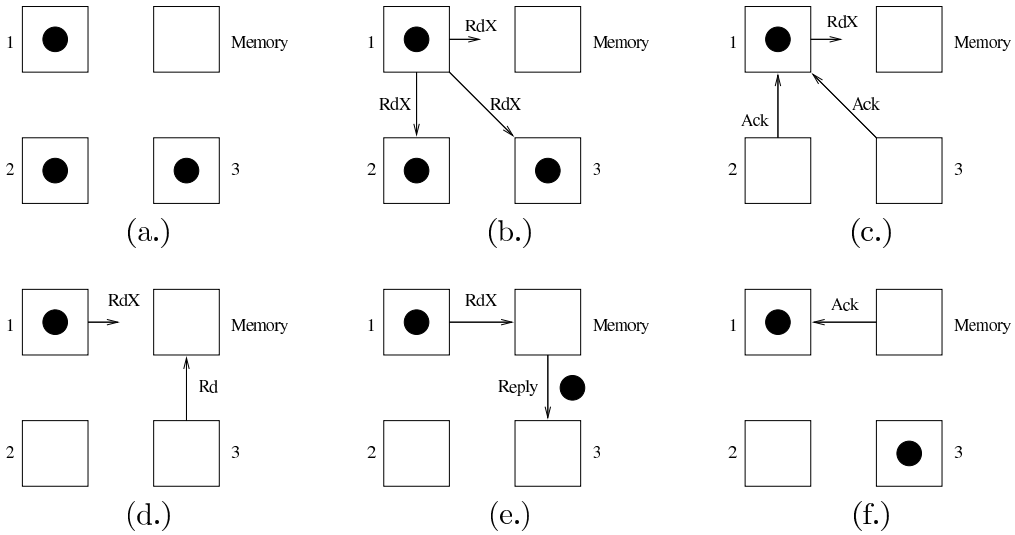
No processor can read a block without holding at least one of the block's tokens and valid data. Since the number of tokens is at least the number of processors, it is possible for any number of processors to cache a block. No processor can write the block without holding

all of the block's tokens and valid data. This ensures no two processors simultaneously write to the same block. In a sense, the block itself is the ordering point for requests. There is not a centralized or stationary ordering point as with previous protocols. The number of tokens for a block held within a processor determine the state of that block. For example, a MOESI coherence scheme can be achieved using token counting in the following way. A cache line is in the INVALID state when it contains no tokens. It is in the SHARED state if it contains less than T tokens and does not contain the owner token. The OWNED state is reserved for cache lines that contain the owner token, but less than T tokens. A cache line is in the EXCLUSIVE state if it contains all the tokens but has not written the line and MODIFIED if it has been written. Entering the MODIFIED state sets the owner token to DIRTY. To ensure dirty data is never discarded, any message that contains a dirty owner token must also contain data. These rules ensure correctness, no matter the performance protocol. In fact, any performance protocol will result in correct operation.

When a processor does not have enough tokens to perform a read or write to a block A, it will send a "transient request" for A's tokens. The performance protocol defines how transient requests are handled. Any other processor may issue requests for some or all of A's tokens at the same time. Since the network does not act as an ordering point, the requests will race for tokens. Therefore, transient requests may fail to retrieve sufficient tokens. If the request fails to return the required number of tokens, the system will retry the request for a period of time. Notice that a failed transaction does not complete and correctness is maintained. If the message has not succeeded by the end of that period, the correctness substrate takes over and activates a "persistent request." Figure 2.4 shows why a traditional broadcast snooping protocol fails to operate correctly in the same situation.

2.4.1 Persistent Requests

When the performance protocol fails to complete a request within a certain amount of time, the correctness substrate is activated. Two persistent request schemes are relevant to this work. The first is somewhat heavy-handed in that it always collects all the tokens for a block despite the fact that the requestor may not require every token to complete the request. Moreover, this method removes tokens from nodes that would not otherwise have to relinquish their right to read the block. The second method addresses this problem by collecting fewer than all tokens when a load caused the persistent request. Both methods



(a.) Initial state. Circles represent valid cached copies. (b.) Node 1 broadcasts a READX request. (c.) Nodes 2 and 3 receive the READX, invalidate their copies and send ACKNOWLEDGMENT messages to 1. (d.) Node 3 issues a READ request. (e.) Memory responds to node 3's request with data. (f.) Memory responds to node 1's request with a second copy of data. Node 1 has received an ACKNOWLEDGMENT from each node and believes incorrectly that is has an exclusive copy of the block.

Figure 2.4: An example race on a unordered broadcast network.

described here rely on a centralized arbiter to activate and deactivate persistent requests. Each home node contains an instance of the persistent request arbitration logic. It is important to note that the arbiter can activate at most one persistent request at a time.

The persistent request mechanism works as follows. A node determines that a request has not been completed after several transient requests. The node sends a persistent request message to the home node of the requested block. The home node places the request in a queue if a persistent request has already been activated by the home node. When the persistent request reaches the head of the queue, the home node broadcasts persistent request activation messages to all nodes in the system. When a node receives the activation request, it forwards tokens for the requested block to the requestor as directed by the persistent request policy. By default, all tokens are sent. Also, any tokens received after the activation message are forwarded to the requestor and any transient requests for the block are ignored. Upon receipt of an activation message, a node records the newly activated persistent requests in a table. The table must hold a record for each active persistent request in the system. Therefore, the size of this table grows with the maximum number of home nodes in the system times the maximum number of persistent requests active per processor. To keep the table to a reasonable size, each home node is limited to one outstanding persistent request.

When the requestor receives sufficient tokens to complete the transaction, a persistent request deactivation message is sent to the home node. The home node forwards a deactivation request to all nodes and removes the persistent request from the queue. In an unordered network, nodes must acknowledge the activation and deactivation requests to assist the arbiter in ensuring that the those messages are properly ordered.

The default response to persistent requests is to send all tokens corresponding to the requested block to the requestor. This requires all nodes in the system to surrender the right to read or write the block. This can cause unnecessary cache misses, so a second, more conservative approach was developed. "Persistent read requests" are activated when a load fails to complete within the allotted time. When a node receives a persistent read request activation message, it forwards all but one non-owner token to the requestor. This allows a node that was able to read the block before the persistent request (with the exception of a node holding only the owner token) to maintain that right after responding to the request. However, there is risk involved with this scheme. If a node causing a persistent read request intends to modify the data shortly after the load completes, it will have to

Current State	Response to READ	Response to READX
MODIFIED	One Token with Data, Transition to OWNED	All Tokens, Transition to INVALID
EXCLUSIVE	One Token with Data, Transition to OWNED	All Tokens, Transition to INVALID
OWNED	One Token with Data	All Tokens, Transition to INVALID
SHARED	None	All Tokens, Transition to INVALID
INVALID	None	None

Table 2.2: Transient request responses.

contact each sharer again to collect all of the tokens. That is, the token "prefetching" effect of normal persistent requests is lost. Normally, read-modify-write operations such as this are completed quickly, but blocks requiring persistent requests are usually highly contested and therefore, the home node will tend to have several persistent requests in the queue. This causes the write operation to wait for several persistent requests to complete before it is activated.

2.4.2 Performance Protocols

The next three sections discuss different performance protocols. In general nodes respond to transient requests as described in table 2.2.

TokenB

TokenB is a broadcast performance protocol intended to avoid indirection latency and the dependence on an ordered interconnect. Indirection can be avoided because each node will receive each request and therefore no state lookup in a central directory is needed. Like all other performance protocols, the reliance on an ordered interconnect is relieved by the correctness substrate. TokenB will be most successful in systems with a large amount of available bandwidth per processor.

TokenD

TokenD is very similar to directory coherence. It is a bandwidth efficient protocol that requires indirection at the home node to find the current sharers and/or owner of a block. Introducing indirection may seem contradictory to the goals of Token Coherence, but TokenD performs two distinct functions. First, it allows the evaluation of Token Coherence as a scalable protocol. Second, it lays the groundwork for TokenM, a multicast performance protocol.

Just as in directory coherence, TokenD sends all requests to the home node of the requested block. The home node forwards read requests to the current owner and write requests to all sharers. Failed transient requests are retried by sending another message to the home node. When a request times out, the persistent request method described above is activated. Aside from existence of tokens, the most significant difference between TokenD and a traditional directory protocol is the maintenance of directory state.

TokenD's directory structure is called a "soft-state" directory because, as opposed to traditional directory coherence, the state is not guaranteed to be completely accurate. Because the correctness substrate ensures that no request is incorrectly satisfied, subtle differences in the actual state of the system and the state reflected in the directory do not cause errant operation. The soft-state directory contains a list of sharers, a "pending" list and the current owner for each block. Upon receipt of a READ request, the home node forwards the request to the current owner and the nodes in the pending list. The union of the sharing list, the pending list and the current owner are the target of forwarded READX messages. Requestors are added to the pending list when a request is received and removed when the directory receives a "completion message." When a node completes a cache miss, it sends a completion message containing the address of the block and the new state of the cache line to the directory. The directory clears the sender from the pending list and updates its state based on completion information. If the new state is MODIFIED, the directory clears the sharing list and sets the current owner to the sender. If the new state is SHARED, the sender is added to the sharing list.

TokenM

TokenM is designed to combine the best features of all coherence protocols. It requires no ordering in the interconnect, avoids indirection in most cases and uses multicast

messages therefore conserving bandwidth. TokenM extends TokenD in the same way that Multicast Snooping extends directory coherence. As in TokenD, a soft-state directory is placed at each home node. Additionally, a destination set predictor [17] is attached to each processor. On a cache miss, the destination set predictor provides a set of nodes likely to require notification of the pending transaction. The request is immediately forwarded to those nodes. The request, with the predicted destination set appended, is also forwarded to the directory. The directory compares the predicted destination set with the current state of the directory. The request is forwarded the nodes node already notified by the requestor. The operation of the sharing and pending lists is the same as with TokenD.

2.5 Optimizing for Migratory Data

Researchers (e.g. [8, 24]) have observed that read-modify-write operations in invalidation-based coherence protocols may incur unnecessary network transactions. These sequences are common inside of critical sections where a variable protected by a lock is read then written by same processor. Cox [8] describes two criteria for identifying migratory blocks: (1) there exist exactly two cached copies of the block and (2) the processor currently attempting to write the block was not the last processor to successfully write to the block.

Consider this sequence of events (for simplicity, assume a broadcast bus connects all processors). Processor A has exclusive access to memory location X and performs a write. Then, processor B attempts to read X. This causes B to place a READ request on the bus. When A receives this message, it responds by placing a copy of the data on the bus. A and B transition to the shared state. Now, processor B attempts to write X, but must eliminate the shared copy at A by broadcasting a READX message. After A has accepted the invalidation message, B can write to X. If block X could have been identified as migratory beforehand, processor A could send X and invalidate its own copy upon receiving the READ request from B. This reduces latency by eliminating the latency of the second request.

Martin used a modification of this scheme [16]. In general Token Coherence provides no way of finding the number of sharers of a block[4]. Therefore, the migratory optimization used in Token Coherence is based on a somewhat looser, less precise definition

[4]TokenD and TokenM could use the directory, but that is not guaranteed to be accurate.

of "migratory." The migratory data optimization for Token Coherence is as follows. Any block in the MODIFIED state that receives a READ request is immediately categorized as migratory. The MODIFIED copy is then invalidated and the data is forwarded to the requestor. Upon receipt of the block, the requestor transitions to the MODIFIEDMIGRATORY state. This state denotes that the block is dirty but has not yet been written by this processor and allows misidentified blocks to return to non-migratory status. Both reads and writes from the local processor to a MODIFIEDMIGRATORY block will hit, but a write will cause the block to enter the MODIFIED state. Because Token Coherence cannot explicitly count the number of sharers, this method runs the risk of identifying too many blocks as migratory and causing cache misses that would have not otherwise occurred.

It is important to note that the accurate identification of migratory blocks is vital to the success of migratory data optimizations such as described above. If non-migratory blocks are considered migratory, performance of other sharing patterns will deteriorate. Producer-consumer and widely-shared are two common sharing patterns that could suffer performance degradation if blocks are misidentified as migratory.

Chapter 3

Experimental Methodology

This chapter describes the tools and techniques used in the evaluation of Token Coherence. Since the goal of this work is to evaluate the performance of several Token Coherence variations, not to accurately determine the absolute runtime, some features that have no or little effect on the performance of the coherence protocol can be simplified. In short, the modeled system is more precise from the CPU-cache interface toward the memory system and less precise toward the CPU core. Figure 3.1 displays this graphically.

3.1 Simulator

The full system, event driven simulator VirtuTech Simics[15] was used to simulate a system based on the "Serengeti" (part of Sun Microsystem's Sun Fire server line) and its attached peripherals such as SCSI controllers, CDROM and Ethernet network interface. The simulator can model a variety of platforms ranging from large UltraSPARC multiprocessors to Alpha servers to ARM-based evaluation boards. In addition, Simics simulates many of the peripherals typically attached to these machines including frame buffers and Fibre-Channel controllers. Though Simics can operate as a purely functional simulator, it is quite extensible and allows the user to compile his own extensions as shared libraries. These libraries are called modules. In this way, a cache model, coherence controller and interprocessor network controller can be attached to each processor to create a cycle accurate simulator at the cost

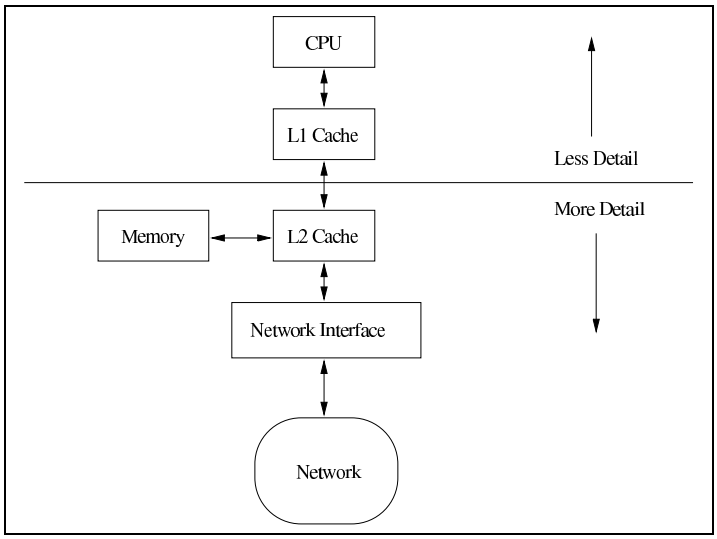


Figure 3.1: Level of detail in simulated systems.

of simulation speed. The Serengeti model seems to be the best suited to multiprocessor simulation and thus thus was the obvious choice for the target platform of this study. The Serengeti systems use UltraSPARC-III processors.

As a full system simulator, Simics models most kernel and user-visible aspects of a system. The advantage is that unmodified operating systems and user programs can be simulated easily and real world bottlenecks can be studied in much greater detail than a physical machine allows. Unfortunately, there are at least two major drawbacks to this method. Simulation run times are increased by operations that may be irrelevant to the user. For example, operating system daemons and other user-level applications may spuriously interrupt the program the user wants to study. The second drawback is variability. Simics can produce deterministic results, but the interleaving of OS code, interrupt handlers and other system perturbations can cause variations in the runtime of benchmarks with different initial conditions.

Alameldeen and Wood studied performance variation [2] with a focus on commercial workloads. They defined the "range of variation" to be the difference of the worst and best observed runtime as a percentage of the mean runtime. The commercial applications were shown to have a significant amount of variation across 20 runs of each benchmark, as much as 14% in the case of Slashcode. The Splash2 version of Barnes and Ocean were also evaluated and had significantly lower ranges of variation, scoring 0.59% and 1.13%

respectively. This suggests that scientific applications exhibit less variability than commercial workloads. In fact, during the course of this work, range of variability between runs of the same program was generally below 2%. Such minor variations do not warrant the simulation overhead of averaging results from a series of runs.

To allow the user to model a custom memory interface, each memory object (e.g. main memory and caches) in the system has an associated "Timing Interface" [26]. The interface is a set of functions written by the user and loaded into the simulator via the module mechanism. When the simulated system is configured, memory objects that should be accurately modeled are attached to an instance of the user's model. When a processor wants to access a memory object, the operate() function of the timing model is invoked. This allows the user's memory model to stall the processor until the requested transaction has completed. Specifically, the operate() function determines how many cycles the processor should stall and returns that number. When the timing model returns 0, the memory transaction is considered complete and the processor advances to the next instruction. Simics allows multiple outstanding memory operations when used with a processor model that supports overlapping memory references. Also, it and allows the user to specify a custom memory consistency model.

3.2 Processor Model

Simics provides two UltraSPARC-III models, in addition to allowing the user to create his own microarchitectural model. The first built-in model is an out-of-order processor that allows the user to adjust parameters such as the ReOrder Buffer size, fetch bandwidth and commit bandwidth [25]. Simics also provides a simple in-order model. This processor executes, with the exception of memory operations, exactly one instruction per cycle. Branch prediction is perfect so there are no speculative operations or rollbacks. When the processor encounters a memory instruction, the timing interface is invoked as described previously. This work is not concerned with measuring IPC rate or microarchitectural effects allowing the simple in-order model to be used. Moreover, each thread (with minor differences for the master thread) in the benchmarks studied here executes the same code. Therefore, simulating a more aggressive processor will speed up program execution between cache misses and will do so at approximately the same rate on all processors.

While a simplified processor model allows for more efficient simulation, some higher order effects are lost. A superscalar, out-of-order processor model would allow multiple outstanding memory requests per processor, increasing the rate of coherence requests per cycle.

3.3 Protocols

A slight variation of the Stanford DASH [13] directory protocol was used as a baseline for the evaluation of three Token Coherence protocols. This section will discuss the implementation of these protocols and highlight any differences from the description of these protocols in the literature. To study the effects of the previously discussed migratory data optimization, the each protocol can be configured to run with or without that optimization.

3.3.1 DASH

To provide a reference to existing coherence protocols, all Token Coherence protocols in discussed in this work are compared to a variation of the Stanford DASH protocol[13]. The DASH protocol is a directory based invalidation protocol that uses three directory states and a sharing list to identify which, if any processors, currently share a block of memory. The caches in the DASH system mirror the directory in their use of three states: DIRTY, SHARED and INVALID. The DASH prototype had four processors per node and the directory's sharing list contained one bit per cluster. To follow the previous Token Coherence studies, this work uses one processor per node and a full map directory.

The network used in this evaluation provides point-to-point ordering, a feature not present in the DASH prototype. This ordering removes many corner cases of the protocol where Negative Acknowledgments (NACKs) would be used to correct awkwardly ordered messages (i.e. invalidation requests arriving at node that has not yet received valid data). Finally, the DASH prototype provided release consistency which is not supported by the chosen processor model. Therefore, the consistency model provided by all memory systems in this work is sequential consistency.

A release consistent machine allows the data response to a READX message to be used immediately, instead of waiting for all INVALACK messages to be collected. This

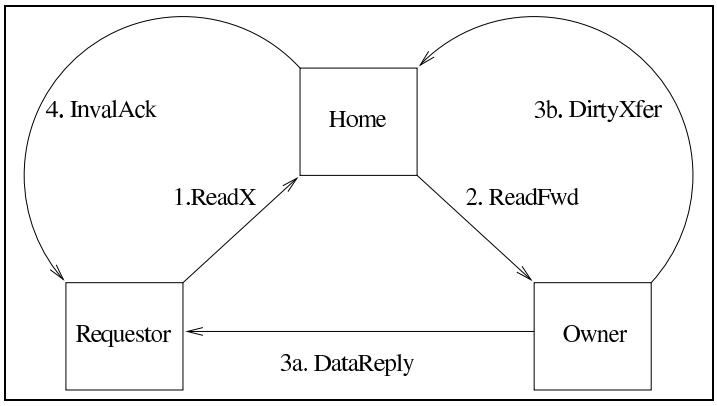


Figure 3.2: A ReadX request to a block in the Dirty state.

reduces the effective latency of READX requests. This is important in DASH because data will return to the requestor in three hops while INVALACKs require four hops. Refer to figure 3.2. The requestor suffers a write cache miss and sends a request the the home node. The home node forwards the message to the current owner. When the owner receives the forwarded request, a DIRTYXFER message is sent to the directory to update the current owner and a DATAREPLY is sent to the requestor. Now, the requestor has an up-to-date block, but the directory must acknowledge that is has updated the current owner of the block. A naive implementation of this optimization violates sequential consistency. However, if the processor stalls at the next cache miss and waits for the pending INVALACKs for the previous miss before continuing, sequential consistency can be preserved [12].

3.3.2 Token Coherence Protocols

The three Token Coherence protocols, including the correctness substrate and migratory data optimization, were reproduced faithfully according to Martin's dissertation [16]. However, the model does differ slightly from the description in section 2.4. The model used here is the MOSI version of token counting also described by Martin [16, 18]. The advantage of the MOESI protocol is that the EXCLUSIVE state can be evicted with a dataless message. A MOSI protocol cannot distinguish between an exclusive clean and exclusive modified block and therefore, must include data when the block is evicted. Due to the large caches modeled in this study evictions are exceedingly rare. Therefore, the performance difference between the MOESI and MOSI counting schemes is negligible.

Benchmark	Parameters
barnes	16k particles, 3 time steps
cholesky	tk29.O
lu	512-by-512 matrix, contiguous
ocean	258-by-258 matrix, contiguous
radix	1M keys, max key 524288, radix 1024
water-n2	512 molecules
water-sp	512 molecules, cutoff radius 6.2 Å

Table 3.1: Benchmark parameters

3.4 Benchmark Applications

The Splash2 benchmark suite is probably the most widely used evaluation method for parallel architectures. The suite [27] was developed for the study of shared address-space multiprocessors and consists of twelve applications. Seven of those applications were chosen for this study. The subset was selected to reduce the total simulation time and still retain the wide variety of sharing patterns and work distribution methods. The graphics applications (radiosity, raytrace and volrend) were eliminated. Barnes and fmm are both N-body simulations (3D and 2D respectively) and have similar runtime characteristics. Thus, the shorter running fmm is redundant and removed. The default parameters for each application were used with the exception of radix where the number of keys was doubled. These are listed in table 3.1. Except for choosing the "contiguous" versions of lu and ocean, no additional effort was made to distribute data in an intelligent manner.

Each application was compiled with GCC version 3.3.3 using the "-O3" optimization level. A subset of the PARMACS [4] macros for portable parallel programming written by Alexis Vartanian allows the applications to be linked with the Solaris native Pthread libraries. The Solaris system call processor_bind() was used to ensure a one-to-one mapping of threads to processors.

3.5 Simulation Parameters

The models used in this work are used to expose the differences in coherence protocols. As such, they are tuned to expose the differences in how coherence misses are

handled. Larges caches are employed to minimize the effect of capacity and conflict misses. Also, the caches are allowed to warm before statistics are collected. This reduces the number of cold misses measured. A perfect L1 cache with contents exactly equal to that of the L2 is implicitly modeled by allowing local memory accesses that would hit in the L2 to complete in zero time. It is assumed that a L1 cache would have a high hit rate for the problem sizes studied here.

The simulated machines presented here follow the previous Token Coherence evaluations [16, 18], which, in turn, were guided by the Alpha EV7 [22]. Figure 3.3 gives a high-level sketch of a node. Sixteen nodes with integrated L1 and L2 caches, cache controllers, memory controllers, network interface and optionally a directory and destination set predictor are connected through a 2D torus operating at 3.0 GB/s. This is an aggressive network implementation in terms of network bytes per instruction. Networks supporting the bandwidth requirements of broadcasting have already been built for systems up to 64 processors [7], much larger than the system studied here. Therefore, system bandwidth is not considered a bottleneck. However, due to the nature of persistent requests (requiring many network traversals to complete) network latency is more likely to be a performance limiter. Like previous work, it is assumed the the network is capable of multicast routing [14, 21] and therefore broadcasting and multicasting will be handled efficiently. Network contention is modeled only at the receiving end-points. With multicast routing, multiple routes to any node and a large amount of available bandwidth, contention is assumed to be minimal at intermediate hops in a message's path. All network paths are point-to-point ordered.

All memories in the system (main memory, cache tags, directories, etc.) are 4-way interleaved by block address. The DASH system allows memory and directory accesses to occur in parallel if both resources are free. Similarly, the Token based systems can access the directory and memory simultaneously.

The destination set predictor used in the TokenM protocol is Martin's Group [17] predictor. The predictor is a 16k-entry table direct mapped to physical addresses. Each entry contains a tag, rollover counter and a 2-bit saturating counter for each remote node in the system. Nodes whose corresponding counter is 2 will be predicted as a member of the destination set. An entry is allocated in the predictor when tokens are received from another processor's cache. Upon receiving token requests and responses from other processors the table is updated in the following way. If the address of the message hits in the

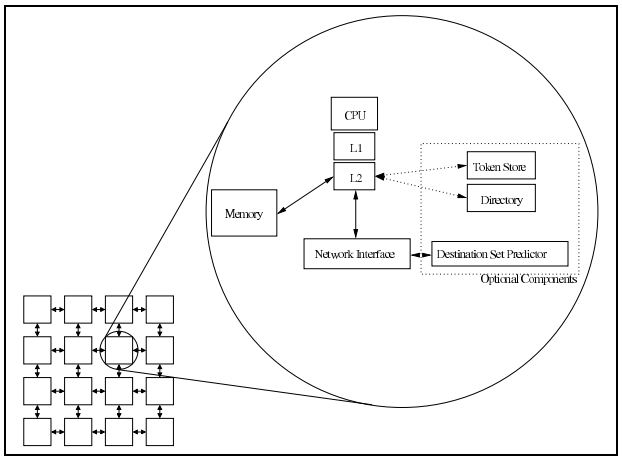


Figure 3.3: Detail of one processing node.

table, the remote processor's counter and the 5-bit rollover counter is incremented. When an entry's rollover counter reaches zero, each 2-bit counter in the entry is decremented. This prevents nodes that are no longer active in sharing a block to be removed from the predicted destination set.

A high rate of requests for a block can cause large queuing delays at contention points. The main memory is especially susceptible to these delays because of its longer latency. The properties of Token Coherence allows the memory controller to reduce the rate of requests issued to the main memory RAMs, thus reducing queuing latency. The following optimizations were made to the Token Coherence memory controller. Since transient requests can be retried, it is common for a processor to have more than one identical request queued at the memory controller. The first optimization is to merge repeated requests into one. Persistent requests exist to ensure the requestor gains access to sufficient tokens to complete its request. Transient requests for the same address as an incoming persistent request are purged from the memory controller's queue. This reduces the latency of persistent requests. Finally, if the memory queues become so long that the requestor is likely to reissue its request, incoming requests are dropped. To avoid deleting tokens and preventing deadlock, persistent request messages and writebacks are never dropped.

Processor	
Processor Model	16 × 1 GHz, Single Issue, In-Order
Caches	
L1	0 ns hit latency, 4 MB, contents always equal to L2
L2	6 ns hit latency, 4-way set associative, 4 MB, random replacement, 4-way interleaved
Network	
Topology	2D torus, point-to-point ordered
Link Latency	15 ns end-to-end
Channel Width	4 bytes
Channel Frequency	750 MHz
Max. Throughput	3.0 GBs
Memory	
Latency	80 ns, 4-way interleaved
Token Store and Directory	
Latency	20 ns, 4-way interleaved

Table 3.2: Simulation parameters.

3.6 Measurement Techniques

All statistics reported were collected during the parallel section of each application. The caches are allowed to warm during the initialization phase of the program. The statistics counters of the simulator are reset at the beginning of the parallel section but the contents of the caches, directories and destination set predictors (if applicable) are maintained. Hints in Splash2 source codes were used as a guide to place timing and processor binding code.

Chapter 4

Results

This chapter describes the results of experiments designed to evaluate the performance of several Token Coherence variants on a subset of the Splash2 benchmarks. First, the effectiveness of the migratory data optimization is examined for DASH and TokenB. Then TokenB, TokenD and TokenM are each evaluated relative to DASH. Finally, because the cost of persistent requests is a potential bottleneck in Token Coherence, a method of reducing the frequency of persistent requests, persistent read requests, is evaluated.

4.1 Rate of Retries and Persistent Requests

This section will give some insight the following sections by describing the behavior of the benchmarks with respect to persistent requests. Due to the bandwidth and latency cost of persistent requests, the rate at which they occur has a major impact on the performance of Token Coherence. Moreover, because each home node can only activate a single persistent request at a time, the distribution of persistent requests across home nodes and the shared data space are also important. If a large portion of persistent requests target a single home node, that node will quickly become a bottleneck.

Commercial workloads have been shown to maintain persistent request rates on the order of 0.2% with retry rates of 2–5% [18]. Table 4.1 describes the rates of persistent requests and retries on seven scientific benchmarks. Clearly, the rate of persistent requests

Benchmark	Persistent Request Rate	Retry Rate
Barnes	7.89	12.4
Cholesky	6.46	8.85
LU	9.91	14.5
Ocean	27.5	40.8
Radix	16.5	18.3
Water-n2	11.3	14.7
Water-sp	18.0	26.3
Apache†	0.29	4.25
OLTP†	0.21	2.43
SPECjbb†	0.07	2.40

†: Results from [18].

Table 4.1: Persistent requests and retries per 100 cache misses for TokenB.

is much higher in the workloads studied here. The benchmarks exhibit persistent request rates of 6.5–27.5% with retry rates up to 40.8%. Though the rate of persistent requests per cache miss is quite high, what is more important is the rate of persistent requests to each home node. Because a persistent request cannot be activated until all previous persistent requests to the same home node have been satisfied, unevenly distributed persistent requests can cause a single home node to become overloaded and the queue of persistent requests at that node will grow considerably.

This raises the question: "Are persistent requests evenly distributed?" Figure 4.1 shows the distribution of persistent requests across addresses and home nodes. Clearly, the target addresses of persistent requests are not evenly distributed. In fact, 10% of persistent request addresses account for 40–80% of all persistent requests. The high frequency of persistent requests to these blocks causes persistent requests to the same block to queue at the home node, drastically increasing the miss latency. The problem of queuing persistent requests is exacerbated by the fact that persistent requests are poorly distributed among home nodes. Nearly all persistent requests are handled by three or fewer home nodes.

4.2 The Migratory Data Optimization

All previously published evaluations of Token Coherence have used the migratory data optimization described in Chapter 2. This study explores different workloads from

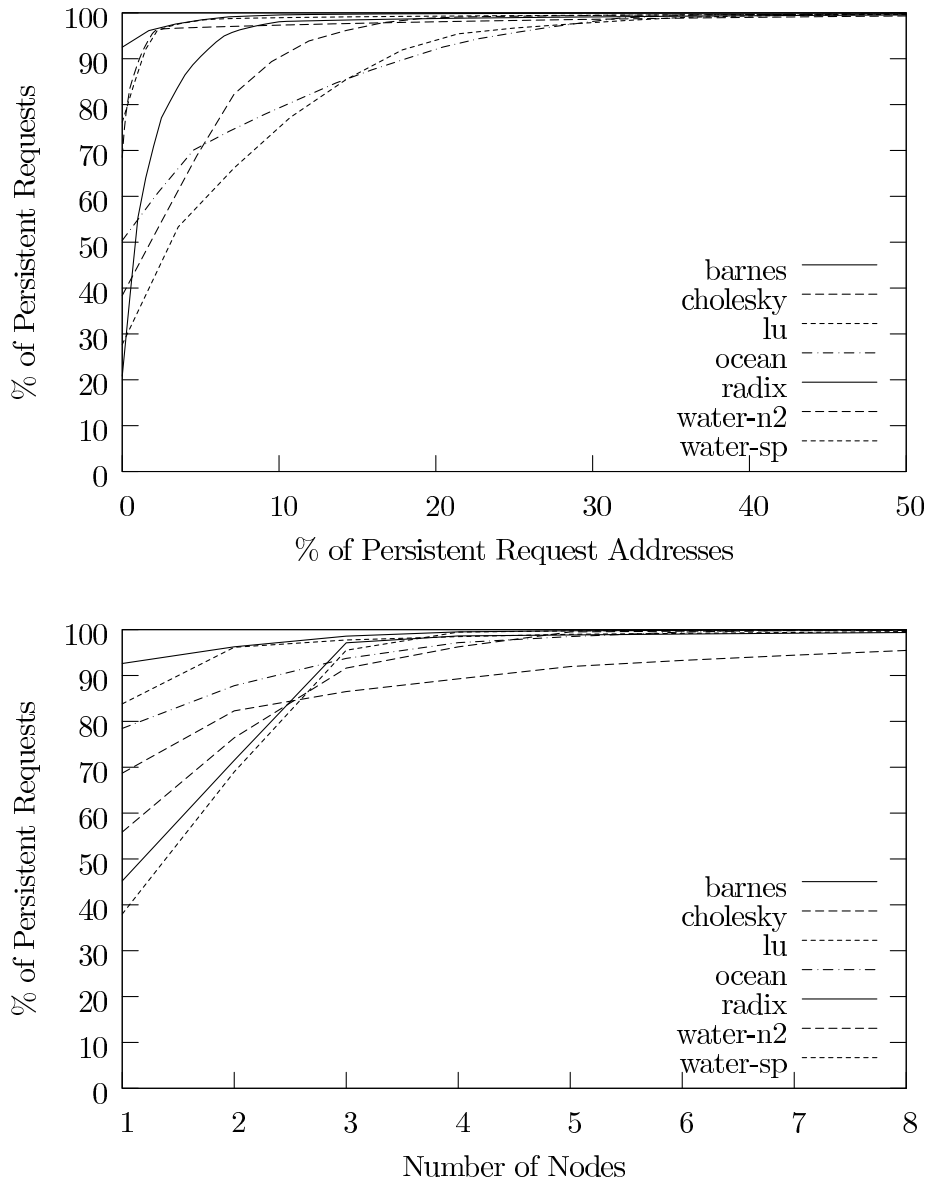


Figure 4.1: Cumulative percentage of persistent requests versus (a.) cumulative percentage of persistent request target addresses and (b.) cumulative number of target home nodes. For clarity only a portion of the horizontal axis is shown.

that body of work. Also, no results have been published that quantify the performance increase of the optimization. Therefore, the optimization must be re-evaluated here.

The goal of the migratory data optimization is to increase performance by eliminating the often unnecessary cache miss incurred when upgrading a readable block to an exclusive state. The danger of this optimization is the possibility of predicting non-migratory blocks as migratory. A misprediction is likely to cause an extra read miss. Consider this sequence of events. Processor A writes a block. Processor B reads that block and receives the data in the exclusive state due to the migratory data optimization. Processor A reads the block and suffers a read miss. In this example, processor A suffers a read miss that would not have occurred without the migratory data optimization. Additionally, processor B does not benefit from the optimization because it has not written the block before moving the the SHARED state when A requests a readable copy. If enough readable-to-exclusive misses are eliminated to overcome the additional read misses caused by misprediction, the migratory data optimization will be beneficial. Unfortunately, Token Coherence lacks a mechanism to directly enumerate the sharers of a block. This is a liability when determining which blocks to treat as migratory. The number of read and write misses suffered by the migratory version of each protocol is shown relative to the non-migratory version in figure 4.2. All benchmarks with the exception of ocean exhibit the expected tradeoff of fewer write misses for more read misses. The results for ocean are anomalous and will be explained separately.

The performance of the DASH protocol with and without the optimization are used as points of comparison for TokenB's behavior. The runtime of each benchmark both with and without the optimization is shown in figure 4.3. All runtimes are normalized to DASH without the optimization. With DASH, enabling the optimization improves runtime by $-0.2\% - +9.0\%$ (ignoring ocean for the moment) with an average of 2.9%. Only one application, barnes, did not benefit from the optimization. Figure 4.2 clearly shows that barnes suffered a significant increase in the number of read misses (56%) while only receiving a moderate decrease in the number of write misses (32%) with the optimization enabled. In this case the negative effect of the optimization did not outweigh the positive effect.

Few applications using TokenB enjoy a significant speedup from the migratory optimization. Speedups range from -1.5% to $+1.1\%$ (again, without ocean) with an average of -0.5%. In fact, only one application, radix, decreases execution time. Radix is unusual in that 92.5% of all persistent requests target one address. This is important because

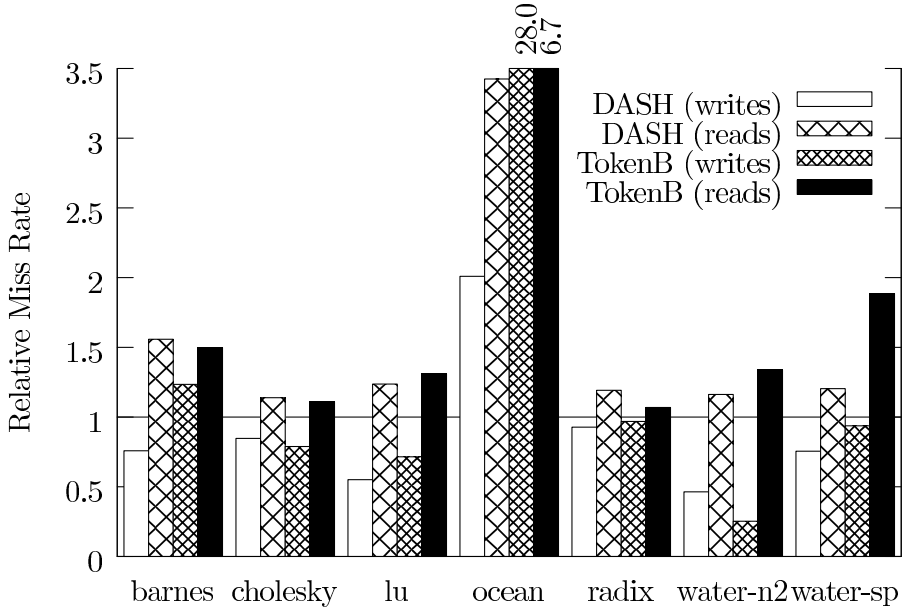


Figure 4.2: Effect of the migratory data optimization on read and write misses.

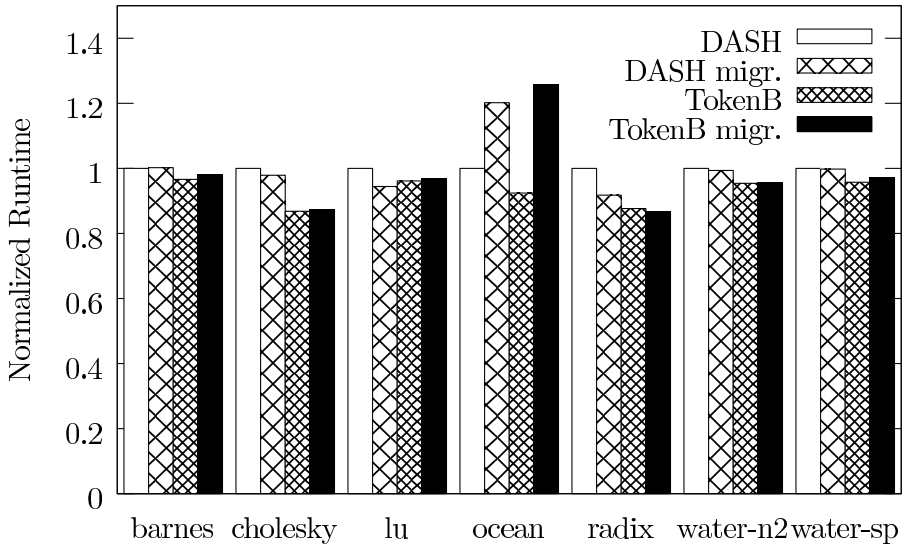


Figure 4.3: The effect of the migratory optimization on DASH and TokenB.

persistent requests are long latency operations. Moreover, persistent requests to the same address cannot be overlapped and must be completely serialized. Persistent requests in other benchmarks are more evenly distributed. The migratory data optimization increases the number of write hits to this block. This eliminates some misses to a highly contended block which would likely result in a persistent request. The result is a shorter queue of persistent requests and therefore a lower average miss latency to the most frequently missed block.

As expected both protocols exhibit a reduction in write misses and an increase in read misses with the migratory optimization active. The percentage of write misses avoided and extra read misses created by the optimization is roughly equal between DASH and TokenB. However, the cause for the difference in runtime becomes apparent when the potential benefit of predicting a migratory block is analyzed. TokenB and other Token Coherence protocols have less to gain from the migratory optimization relative to DASH for the following reason. When the optimization works properly, DASH replaces a dirty to shared operation (116 cycles on average without contention) followed by a ReadX to a shared block (140 cycles average) with a change of owner operation (116 cycles) followed by a cache hit (0 cycles). This saves 116 cycles per invocation of the optimization. On the other hand, TokenB will replace a Read (66 cycles average without contention) followed by a ReadX (66 cycles) with a single Read operation. This saves only 66 cycles.

The performance of ocean is drastically reduced when the migratory data optimization is enabled. The shared data in ocean is divided among processors in the system and each processor only modifies its own section of the shared data. The copying of neighbor data in the first four loops of the laplacalc() function cause a huge number of cache misses with the optimization enabled. The misses due to these loops account for most of the decrease in performance.

The migratory data optimization also has the potential to reduce contention and therefore increase performance. However, the reduction of contention in the network and various memories was not significant in this study.

4.3 Performance Protocols

Given the properties of the studied coherence protocols and the simulated machine the following relationships are likely. (1.) TokenB outperforms DASH because the network

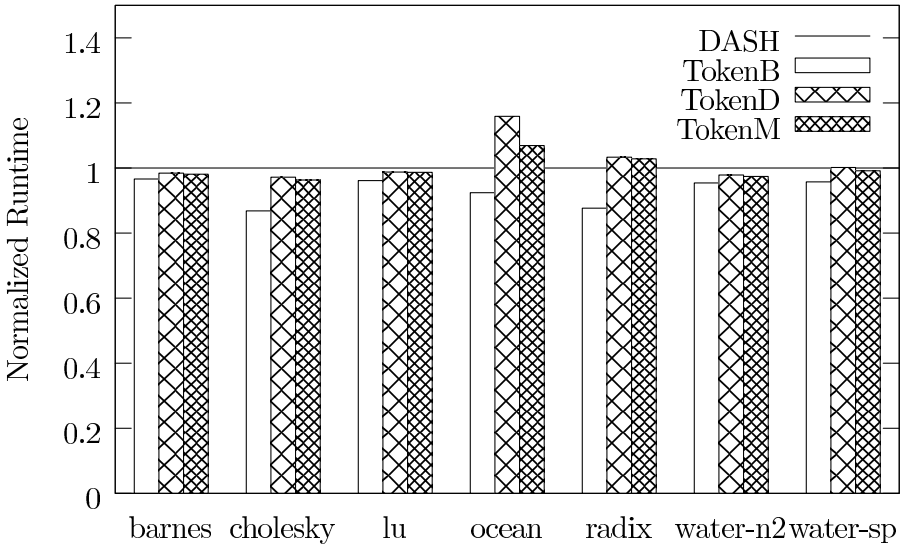


Figure 4.4: Runtime of TokenB, TokenD and TokenM relative to BASH.

has a large amount of bandwidth available to support broadcast. Also, coherence requests in DASH suffer an extra latency penalty for indirection. TokenB avoids the home directory. (2.) TokenB outperforms TokenD for the same reasons. (3.) TokenM's performance will fall between TokenB and TokenD. The latency-bandwidth tradeoff, discussed in Chapter 2, TokenM predicts that because TokenM uses an amount of bandwidth between TokenB and TokenD, TokenM's requests will have an average miss latency between the other two Token Coherence protocols. Therefore, TokenM is likely to not perform as well as TokenB but better than TokenD. However, TokenM has the potential to avoid indirection and simultaneously reduce contention. This combination of features is not possible either TokenB or TokenD. Figure 4.4 shows the runtime of DASH and the three performance protocols without the migratory data optimization. All runtimes are normalized to DASH.

In all cases the above predictions matched the results of the simulator. Despite the high persistent request and retry rates. Token Coherence outperforms DASH in all but four situations: the combinations of TokenD or TokenM and ocean or radix. The added latency of forwarding requests to the home node increases the length of time a request is vulnerable to interference from a racing request[1]. Relative to TokenB, the number of racing requests

[1]Request A is considered to "race" request B if both requests are simultaneously outstanding.

	Persistent Request Rate		
Benchmark	TokenB	TokenD	TokenM
Barnes	6.05	8.04	7.88
Cholesky	6.46	10.8	8.44
LU	9.91	11.2	11.7
Ocean	27.9	34.7	31.0
Radix	16.5	16.4	16.9
Water-n2	11.3	13.7	14.7
Water-sp	18.0	21.1	19.4

Table 4.2: Persistent requests per 100 cache misses for TokenB, TokenD and TokenM.

increases by 68% and 76% for ocean and radix respectively when using TokenD. More races produce conflicts which, in turn, increase miss latency. TokenM can only partially reduce the number of conflicting races.

Table 4.2 shows the persistent request rates for each of the three performance protocols. The persistent request rates of TokenD and TokenM are clearly higher than that of TokenB due to the effect of races just described. Comparing figure 4.4 and table 4.2, we can see that, as expected, persistent request rate is a fairly good estimator of relative performance.

4.4 Persistent Read Requests

Another optimization of Token Coherence that has been used in other work without a quantitative display of its effectiveness is the persistent read request. When a load causes a persistent read request, a minimal number of tokens are moved to the requestor. In contrast the default persistent request mechanism requires all tokens to be sent to the requestor causing unnecessary read misses. Therefore, the benefit of persistent read requests is a reduction of read misses. Figure 4.5 shows the runtime of the Splash2 subset using persistent read requests, relative to the runtime of the default persistent request scheme. The persistent read request mechanism improves performance by 0–5.7%. Lu and radix enjoy the largest reduction in read misses and consequently, the largest speedup. Persistent read requests reduce the number of read misses in lu and radix by 15.2% and 14.8% respectively. The remaining benchmarks average only a 10.9% reduction in read misses.

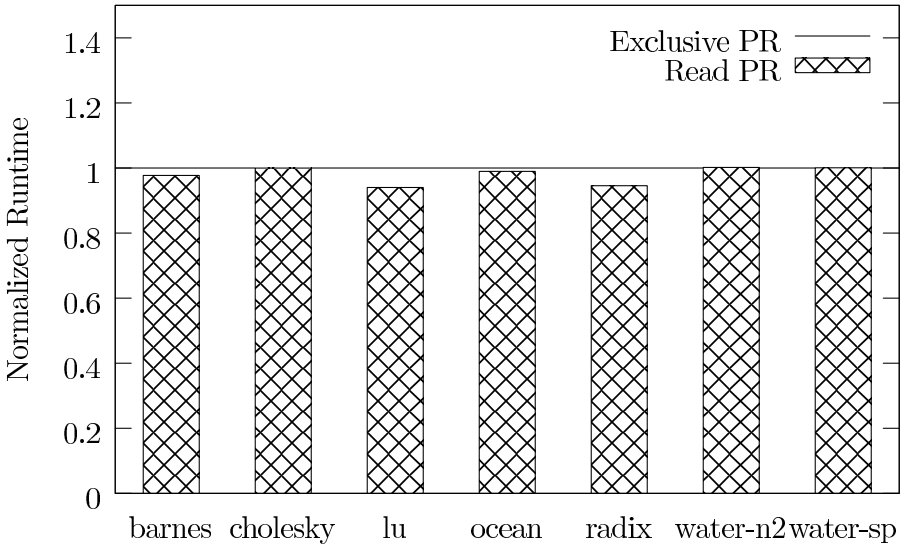


Figure 4.5: Effectiveness of persistent read requests.

4.5 Sensitivity to Network Parameters

The network parameters used in all simulations discussed up to this point have been somewhat ambitious. This section describes the results of three scientific benchmarks simulated with a less aggressive network model. Three benchmarks with interesting properties were chosen to run with the new parameters. Ocean exhibited moderate bandwidth utilization in the previous tests, but proved to be sensitive to miss latency. Radix consumed more bandwidth than any other benchmark run under TokenB and also showed some sensitivity to request latency. Finally, acting as a kind of control group, lu performed well under all three performance protocols. The latency of each network link was extended from 15 to 35 ns and the link bandwidth was halved, resulting in 1.5 Gb/s links. Neither the persistent read request or the migratory data optimization was used runs presented in this section. Figure 4.6 shows the runtime of the three benchmarks with the adjusted network parameters.

With the new parameters, ocean and radix continue to perform poorly with TokenD and TokenM. Again, the number of racing request increases dramatically in those cases. Ocean suffers a 54% in racing requests when using either for TokenD or TokenM. This results in increases 52% and 54% in the number of persistent requests when using To-

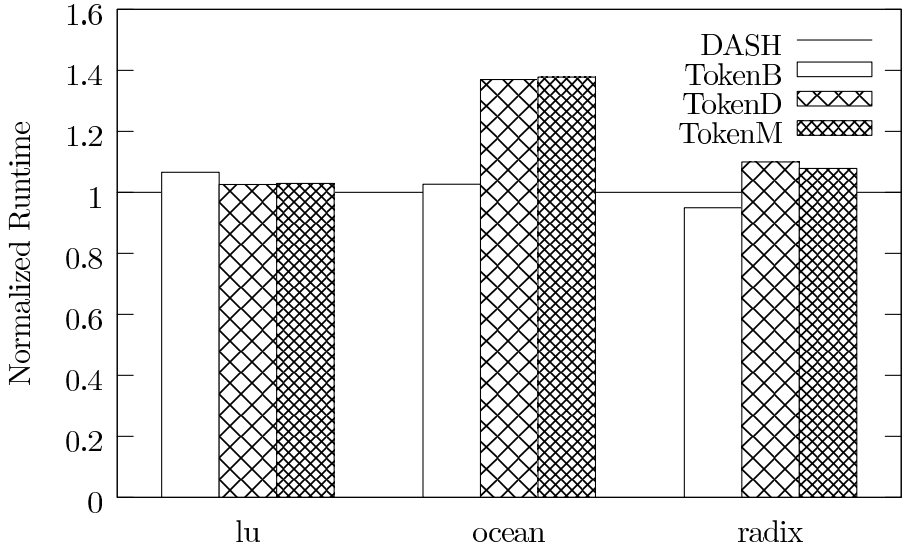


Figure 4.6: The runtime of lu, ocean and radix with an alternate network.

kenD and TokenM respectively. The same effect is less dramatic with radix. The increased request latency of TokenD and TokenM cause 33% and 21% more races respecively. This results in 11% more persistent requests when either TokenD or TokenM is used. In contrast, the performance of lu improves when using TokenD or TokenM. This improvement mainly is due to reduced network contention. Delays caused by network contention were reduced by 59% and 41% for TokenD and TokenM. On average TokenB remained competitive with DASH. The increased network load of TokenB caused a slight slow down compared,u to DASH for lu and ocean, 6.5% and 2.6% respectively.

Figure 4.7 shows the increase in runtime for each lu, ocean and radix compared to the more aggressive network. In lu and radix, the sensitivity of DASH, TokenD and TokenM are comparable. This is expected because of the unicast and multicasting features of the those protocols. The slowdown of TokenB for each benchmark was moderate, despite being a broadcast protocol. For all protocols radix slowed down considerably when using the slower network. This is due to its higher cache miss rate and network bandwidth usage. Ocean's slowdown was quite high for the reasons described earlier in this section.

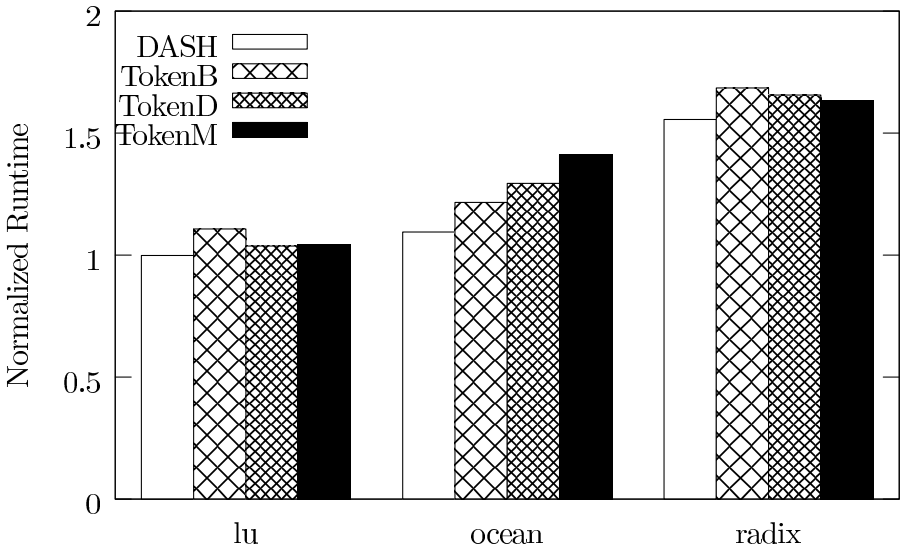


Figure 4.7: Sensitivity to a slower network.

Chapter 5

Conclusion

This work has shown that Token Coherence is not dependent on the peculiarities of commercial workloads and can improve the performance of scientific applications on mesh of highly integrated processing nodes. In fact, Token Coherence performs well despite persistent request rates up to 27.5%, much higher than previously published benchmarks. While the poor distribution of persistent request target addresses across home nodes is a concern for scalability, it is not detrimental to performance at 16 nodes with the parameters used here. However, some variants of Token Coherence that work well with commercial workloads do not produce a benefit for scientific workloads. Specifically, the migratory data optimization is not effective on the benchmarks studied here. The ocean application has a particularly adverse reaction to the optimization increasing runtime by 36%. Also, read persistent requests give a moderate boost to scientific applications, reducing runtime by up to 5.7%.

The three performance protocols evaluated performed as predicted with respect to the baseline DASH protocol. TokenB is the fastest of the four protocols across all benchmarks. TokenM always falls between the performance of TokenB and TokenD. For all but two benchmarks, ocean and radix, TokenD and TokenM are also faster than DASH. The exceptions are due to a high rate of conflicting races. This is a weak point of Token Coherence due to the race recovery mechanism, persistent requests, which have long latency compared to transient requests. To address Token Coherence's sensitivity to network parameters a less aggressive network implementation was also tested. Despite the less generous bandwidth

and increased latency of the system TokenB remained competitive with DASH.

The contributions of this work are as follows.

1. The performance of Token Coherence was measured under a new set of benchmarks quite different from previously published work.

2. The rate of persistent requests were shown to have an impact on performance, but the increased persistent request rates were not detrimental to performance in this study. For example, lu and radix perform significantly better with TokenB than with DASH despite persistent request rates of 9.9% and 16.5%. These are much higher than the 2–5% reported for commercial benchmarks.

3. The effectiveness of two optimizations for Token Coherence used in previous work (without a quantitative measure of performance gain) was measured. The migratory data optimization was shown not to be effective for scientific workloads and the read persistent request optimization provides only minor speedups.

4. The sensitivity of Token Coherence to network parameters was measured. Though more sensitive to a slower network design than DASH, the Token Coherence protocols remained competitive with DASH despite modest network parameters.

5.1 Future Work

The first point of concern with the realism of the setup presented here is the simplified processor model. A more complete system should be modeled allowing multiple outstanding memory transactions from each processor. Additionally, the current processor model lacks the ability to throttle its own network activity when the network becomes congested. This feature is essential to accurately model fully utilized networks.

Finally, the network sensitivity studied conducted here should be expanded. Latency and bandwidth limitations should be studied independently and more data points are needed. Of course, this study ought to provide results for, at least, all seven benchmarks is chosen for the evaluation of the default parameters.

Bibliography

[1] NR Adiga et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[2] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, February 2003.

[3] James Archibald and Jean-Loup Baer. Cache Coherence Crotocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[4] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS Macros for Shared-Memory Multiprocessor Environments, 1997.

[5] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA '98: Proceedings of the 25nd Annual Anternational Aymposium on Computer Architecture*, pages 3–14, 1998.

[6] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *ISCA '99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304. IEEE Computer Society, 1999.

[7] Alan Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, 1998.

[8] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108. ACM Press, 1993.

[9] Zarka Cvetanovic. Performance Analysis of AlphaServer GS1280. *IEEE Micro*, 18, January 2003.

[10] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and Design of AlphaServer GS320. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24. ACM Press, 2000.

[11] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251. ACM Press, 1997.

[12] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA '95: Proceedings of the 22nd Annual Anternational Aymposium on Computer Architecture*, pages 48–59. ACM Press, 1995.

[13] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Dased Dache Doherence Drotocol for the DASH Dultiprocessor. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159. ACM Press, 1990.

[14] Xiaola Lin and Lionel M. Ni. Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 116–125, New York, NY, USA, 1991. ACM Press.

[15] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, February 2002.

[16] Milo M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin - Madison, 2003.

[17] Milo M. K. Martin, Pacia J. Harper, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Using Destination-set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217. ACM Press, 2003.

[18] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token Coherence: Decoupling Performance and Correctness. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193. ACM Press, 2003.

[19] Milo M. K. Martin, Daniel J. Sorin, Anatassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36. ACM Press, 2000.

[20] Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Bandwidth Adaptive Snooping. In *HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, page 251. IEEE Computer Society, 2002.

[21] Prasant Mohapatra and Vara Varavithya. A Hardware Multicast Routing Algorithm for Two-Dimensional Meshes. pages 198–205, 1996.

[22] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The Alpha 21364 Network Architecture. In *HOTI '01: Proceedings of the The Ninth Symposium on High Performance Interconnects*, page 113. IEEE Computer Society, 2001.

[23] Paul F. Reynolds, Jr., Craig Williams, and Raymond R. Wagner, Jr. Isotach Networks. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):337–348, 1997.

[24] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118. ACM Press, 1993.

[25] Virtutech AB. Simics Out of Order Processor Models. October 2003.

[26] Virtutech AB. Simics User Guide for Unix. October 2003.

[27] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM Press, 1995.