PHD: A Hierarchical Cache Coherent Protocol

by

Deborah A. Wallach

S.B., MIT
(1990)

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Science
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
September 1992

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
September 7, 1992

Certified by ______________________________

Dr. William J. Dally
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________________

Professor Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

PHD: A Hierarchical Cache Coherent Proto

by

Deborah A Wallach

Submitted to the
Department of Electrical Engineering and Computer Science
on September 9, 1992, in partial fulfillment of
the requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

As the number of processors in distributed-memory multiprocessors grows, efficiently supporting a shared-memory programming model becomes difficult. We have designed the Protocol for Hierarchical Directories (PHD) to allow shared-memory support for systems containing massive numbers of processors. PHD eliminates bandwidth problems by using a scalable network, decreases hot-spots by not relying on a single point to distribute blocks, and uses a scalable amount of space for its directories. PHD provides a shared-memory model by synthesizing a global shared memory from the local memories of processors. PHD supports sequentially consistent read, write, and test-and-set operations.

This thesis also introduces a method of describing locality for hierarchical protocols an
employs this method in the derivation of an *abstract model* of the protocol behavior. An *embedded model*, based on the work of Johnson [13], describes the protocol behavior when mapped to a k-ary n-cube. The thesis uses these two models to study the average height in the hierarchy that operations reach, the longest path messages travel, the number of messages that operations generate, the inter-transaction issue time, and the protocol overhead for different locality parameters, degrees of multithreading, and machine sizes.

We determine that multithreading is only useful for approximately two to four threads; any additional interleaving does not decrease the overall latency. For small machines and high locality applications, this limitation is due mainly to the length of the running threads. Fo
large machines with medium to low locality, this limitation is due mainly to the protocol overhead being too large.

Our study using the embedded model shows that in situations where the run length between references to shared memory is at least an order of magnitude longer than the time to process a single state transition in the protocol, applications exhibit good performance. If separat
controllers for processing protocol requests are included, the protocol scales to 32k process
machines as long as the application exhibits hierarchical locality: at least 22% of the global
references must be able to be satisfied locally; at most 35% of the global references are allowed to reach the top level of the hierarchy.

Thesis Supervisor: Dr. William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Hierarchical Cache Coherence, Directory-Based Cache Coherence, Locality.

Acknowledgments

This thesis has benefitted greatly from the help of many people.

I thank all of the members of the CVA group, for providing valuable feedback both at group meetings and at other times. I especially thank Nick Carter and Steve Keckler, for cheerfully reading full drafts of my thesis, and John Keen for on-demand proofreading of particularly troublesome passages.

I thank my advisor, Bill Dally, who pushed me in the right direction when I needed nudging.

I thank Fred Chong, for actually *volunteering* to read the earliest draft of my thesis and for providing me with very helpful comments.

I am especially grateful to Kirk Johnson, for proposing the kernel of the idea of the method for describing the locality in applications used by this thesis. Kirk also patiently answere all of my questions about his model, and let me query him numerous times.

I also thank David Chaiken for giving me a copy of his trace data as well as for providing general caching advice at various times throughout the years.

Finally, I am grateful to the late-night crew, who both improved my thesis by providing answers to all sorts of questions and impeded my thesis by distracting me from my work. You know who you are.

The research described in this thesis was supported in part by an Office of Naval Research American Society for Engineering Education Fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The shared-memory model has been a convenient programming paradigm for multiprocessors. As the number of elements in multiprocessors grows, however, efficiently supporting a shared-memory programming model becomes difficult. Bus-based snooping schemes suffice for only small numbers of processors; they are inadequate for large numbers of processors because their bandwidth does not grow with the number of processors [8]. Directory-based caching schemes, on the other hand, allow sharing among large numbers of processors when implemented on network-based computers; the bandwidth of the network must increase with the number of processors. *Hierarchical* directory-based schemes have the potential to scale indefinitely because they have neither the space requirements of full-map directory schemes nor the limited number of copies requirements of limited-directory schemes nor the linear dependence on the number of copies for invalidation of chained schemes. Hierarchical schemes additionally exploit the spatial and temporal locality of processes running on a machine. We have designed the Protocol for Hierarchical Directories (PHD) to provide shared-memory for systems composed of massive numbers of processors.

PHD synthesizes a global shared memory from the private local memories of processors. Processors access global addresses in the same manner as they access local ones. The system operates on *blocks* (or *lines*) consisting of several words of data, capitalizing on spatial locality. PHD maintains sequential consistency [14] in its support of read, write, and test-and-set operations.

In this thesis we also introduce a method of describing locality for hierarchical protocols

We derive an *abstract model* of the behavior of the Protocol for Hierarchical Directories using this method and use it to study the average height reached in the hierarchy per operation, the longest path of messages traveled per operation, and the number of messages generated per operation for different machine configurations. We validate this model using a trace-driven simulator.

The abstract model is used to generate inputs to an *embedded model*. The embedded model describes how the protocol behaves when mapped to a k-ary n-cube using the our proposed mapping scheme.

1.1 Directory-Based Protocols

Many of the ideas used in the Protocol for Hierarchical Directories evolved from directory-based protocol research as well as early hierarchical protocols. Most of the early research assumed certain capabilities, such as a broadcast ability, which do not scale well. Several other hierarchical protocols have been proposed since the start of this work.

1.1.1 Flat Directory-Based Protocols

All directory-based protocols keep a record associated with each block of main memory. There have been a wide variety of directory schemes proposed and studied. Tang [26] proposed a write-back scheme in which the main memory and every cache must keep a directory. In order to find a block, all of the individual directories need to be checked. Censier and Feautrier [7] first proposed the concept of a "dirty bit" which indicates whether or not the value stored in main memory is the newest one. They also added a bit vector to the main memory directory indicating which caches have copies of the block. These additions eliminated the need to search every local directory after every data modification.

Agarwal [4] discussed these schemes and their lack of scalability due either to the need to broadcast or to the presence of a bottleneck. He mentioned the idea of distributing the directory across the memories, in order to prevent any bottlenecks. Chaiken [8] showed that directories are scalable and that some shared-data caching schemes, for many applications, perform better than schemes which cache only private data. The shared-data schemes that he looked at include limited directory, full map, and singly and doubly-linked chains.

Limited directory schemes employ a limited number of pointers to keep track of which processors have copies of particular blocks; when a new processor wants a copy and the limited number of pointers have all been allocated, the scheme must resort to broadcast or to invalidation. In a full-map directory, all processors can have copies of any block. Singly linked chains distribute the directory entry, threading it through the processors which have copies. Doubly-linked chains use a double linkage, to allow the chain to be followed either way.

1.1.2 Hierarchical Schemes

In the above-mentioned directory schemes, the *home* location of a particular block is statically fixed: any processor without a copy of the block in its cache which wishes to access that block must look in a single fixed location. Hierarchical directory schemes were designed both to reduce this static requirement by providing adaptive data migration and to solve the limited bandwidth problem of single bus schemes.

A hierarchical scheme, in general, has a tree structure. At the lowest level of the tree are processors with caches; at the other levels are directories recording which blocks are cached by nodes located physically below them in the tree. Any number of copies of a block are allowed to exist at a time. A read request is typically propagated up the tree until a copy is located; a write typically involves locating and invalidating all of the extra copies by tr traversal and then performing the write.

In [30] Wilson proposes the first hierarchical multiprocessor architecture. He suggests modifications to several bus-based schemes in order to form a protocol for his proposed hierarchy which uses shared buses of caches to form the tree. He does not, however, consider how his ideas would work on very large scale systems. Archibald, in [5], proposes another solution intended for a small hierarchy of buses, remarking that his protocol is feasible fo a two-level hierarchy, but not necessarily a three-level, or four-level one.

Haridi and Hagersten [12] later proposed a hierarchical scheme which was designed for a much larger systems: the Data Diffusion Machine (DDM). Their architecture also assumes a tree composed of buses, which forces all requests to be routed through the hierarchy. The intermediate level directories store information as to whether copies of each block cached below is cached anywhere above or whether it is *exclusive* to that subtree, thus allowing them to reduce traffic on the higher-level buses during writes. Their architecture also eliminates the need for a *home* location for a block. Their hierarchical scheme typifies a COMA, or Cache-Only Memory Architecture, as defined in Stenström's [25] paper.

In a later paper, Yang, Thangadurai and Bhuyan [32] proposed a similar hierarchical bus scheme which also keeps track of the exclusivity status for each block. Unlike Haridi's scheme, however, they assume that the main memory is situated at the top of the hierarchy, providing a static place for blocks to be stored in when they are thrown out of the caches.

Scott and Goodman describe a hierarchical scheme for processors connected using a k-ary n-cube network in [23] and [24]. Their mapping scheme provides rings, which replace buses as the broadcast method for their protocol. They also introduces the concept of *pruning caches*, which eliminate the need of all of the earlier protocols for complete multi-level inclusion, *i.e.* keeping a higher level directory entry for every lower level one. Prunin caches allow a tradeoff between directory size and network bandwidth to be dynamically made, and could be added to PHD.

Maa, Pradhan, and Thiebaut [18] [19] are currently working on a hierarchical directory scheme for non-bus-based architectures, but have not yet fully worked out the details of the protocol.

The Kendall Square Research Company has built a system with a ring-based hierarchical directory scheme [6]. In their scheme multi-level inclusion is required. They have not released much information about their protocol.

Parthasarathy [21] studied an earlier version of PHD, DHP, described in [29]. Although his refined version of DHP traverses the hierarchy fewer times than does PHD, it will deadlock under certain conditions. Parthasarathy's work does not consider this problem His protocol also does not guarantee that a read operation will make enough progress to complete even in the absence of deadlock since write operations can skip ahead of reads indefinitely.

1.1.3 PHD

The Protocol for Hierarchical Directories is a tree-based hierarchical directory protocol. Any number of processors can have read-only copies of a block in their caches. To find a block, a processor sends a location message which travels upwards until a node which knows of a copy is found. This node sends a message which travels downwards until it reaches a node with a copy. The node with a copy sends the block directly to the requesting node, which then sends a confirmation message upwards to indicate that it has finished its read. In this manner, reads can be satisfied in the lowest common subtree containing the requester and a copy of the block.

Write operations involve finding all of the copies of a block in the system and deleting them. Only the nodes in the smallest subtree containing all copies of the block and the write requester are involved in a write. The *owner* of the block transfers ownership to the node requesting the write. Acknowledgments of deletion from all of the nodes which previously had copies are combined, and an acknowledge message is sent downwards to the node that requested the write. The test-and-set operation is actually a test-and-test-and-set operation; it is implemented as an optimized combination of the read and write operations.

1.2 Analysis and Locality Modeling

Many previous models [15] [28] [31] of hierarchical cache consistency protocols have modeled bus architectures, and as such, considered bus traffic effects to be most important. Leutenegger and Vernon, in [15], assume uniform cache miss rates across the machine. Yang [31] assumes a single-level clustering model for reference rates, where each smallest group of processors is equally likely to access some blocks and all other processors are equally less likely to access those blocks. Vernon, Jōg, and Sohi [28] do not directly consider data locality; instead, they choose fixed miss ratios for different level caches.

Scott, in [24], actually calculates the traffic for a ring-based hierarchical system. He assumes best-case, worst-case, and random data-access patterns in his study.

Johnson[13] studies locality and its effects on multiprocessor performance. He derives a combined model of applications, communication mechanisms, and interconnection networks, and uses the result to show that "exploiting communication locality provides gains which are at most linear in the factor by which average communication distance is reduced," as long as the outstanding number of communications per processor is bounded. We use his model as the basis for the embedded model studies of Chapter 6.

Stenström, Joe, and Gupta [25] compare the performance of a COMA architecture with that of a NUMA (non-uniform memory architecture). They find that the COMA architecture performs worse than the NUMA one for many situations, such as those where coherence misses dominate over capacity misses. Many of their assumptions, however, do not apply to the work described in this thesis. They assume a 16 processor configuration, where the effects of locality are going to be less important than on a massively parallel machine. They also assume that the COMA architecture will be running Haridi and Hagersten's DDM protocol. PHD, on the other hand, as will be explained in Section 1.4, not only uses a shorter path in order to satisfy read requests, but also eliminates the replacement problem of the DDM protocol. The paper concludes with their proposal of COMA-F, a flat COMA architecture. COMA-F like PHD, has a master (owner) node.

1.3 J-Machine

The cache coherence protocol was designed as part of the J-Machine [9] project at MIT. The J-Machine is a massively parallel, fine-grained message-passing concurrent computer. Although the J-Machine was designed to efficiently support a message-passing language, it provides inexpensive synchronization primitives to support other programming models as well. The cache coherence protocol was developed in the context of considering shared-memory programming environments for the J-Machine.

1.4 Contributions

The Protocol for Hierarchical Directories differs in several ways from previously proposed hierarchical schemes. It is designed for message-passing multicomputer systems which use small cache block sizes. It is both scalable and strongly coherent.

1.4.1 Scalability

PHD is scalable in cost and latency, as defined by Scott in [23]. He requires that cost grow slower than $O(N^2)$ with machine size, and latency grow no faster than $O(N^{\frac{1}{2}})$.

Cost The cost of the hardware includes the cost of the network and the cost of the directory which stores tag bits. A k-ary n-cube, as long as the dimensionality properly increases with size, grows at less than $O(N^2)$ [23]. The directory overhead for PHD is $O(N \log N)$ by Scott's definitions. Therefore, PHD is scalable in cost.

Latency As shown in Chapter 5, the unloaded-network predicted latency per read or write operation scales at less than $O(N^{\frac{1}{2}})$. The latency due to protocol overhead for the proposed embedding of PHD into a k-ary n-cube depends on the total number of messages sent and thus the degree of sharing.

Bottleneck at the Top of the Hierarchy Unlike in a bus-based hierarchical architecture, requests which span across the machine are not constrained to cross through the same point. PHD distributes the levels of the hierarchy across each node of a machine. There is no bottleneck at "the" top directory, because there are different top directories for different blocks. This mapping is described in Section 2.2.

1.4.2 Messages and Longest Path Traversed

Because PHD is not restricted by the machine architecture to a follow the hierarchy at all times, both the longest path traveled and the number of messages generated per read are shorter and fewer than in an enforced hierarchy.

Longest Path per Operation As shown in Figure 1.1, a read in PHD is satisfied directly after two traversals of the hierarchy and a single direct message to deliver the data. Strict hierarchies require four hierarchy traversals before a read result can be used.

Messages per Operation The number of messages per read operation is also smaller in PHD than in a standard hierarchy. As Figure 1.2 illustrates, only three traversals of the

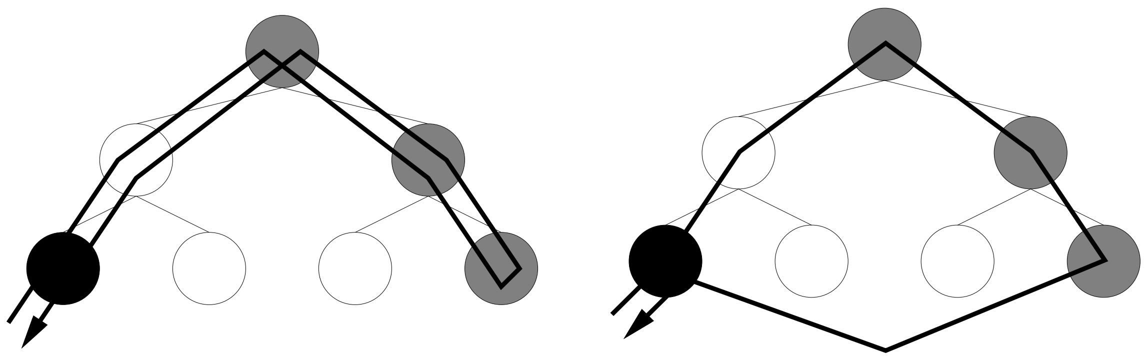


Figure 1.1: The left side of the figure shows the path a read request must fulfill before it receives the data for the other protocols. The right side shows how the path is shorter for PHD.

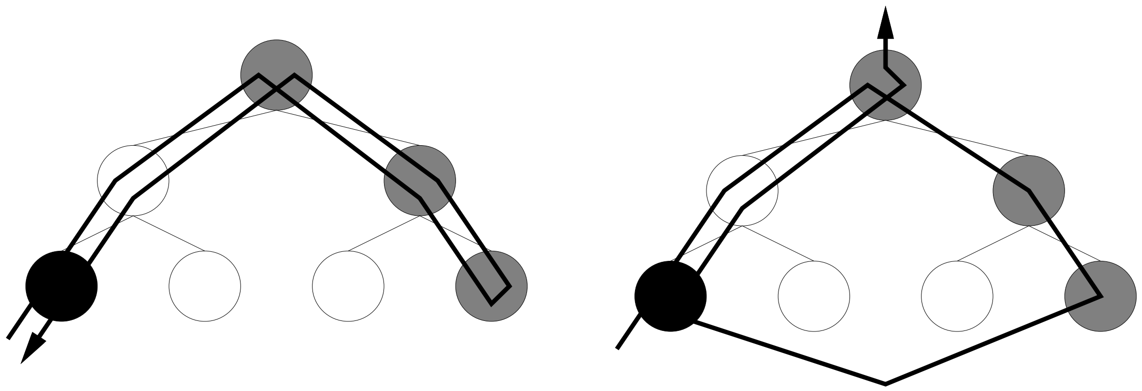


Figure 1.2: The left side of the figure shows the path a read request must fulfill in order to complete for the other protocols. This path is identical to the number of messages which must be sent. The right side shows how the path is shorter for PHD.

hierarchy worth of messages plus one direct data-delivery message need to be sent for PHD as opposed to four traversals of the hierarchy worth of "messages" for the strict hierarchy.

1.4.3 Ownership

The concept of ownership [10] as used in this protocol was derived from both Li [16] [17] and Totty [27]. An owner of a block is responsible for it. Any other node can only have a copy of the block, which can be asynchronously thrown away in order to make room for other blocks. That node can then inform the rest of the system at its leisure without affecting the correctness of the protocol. This ability to throw away unneeded copies of blocks without the global transactions required by Haridi and Hagersten results in less time needed in order to invalidate blocks when caches are full.

1.4.4 Mapping Scheme

This thesis proposes a mapping scheme designed to map hierarchical cache coherence protocols onto non-toroidal k-ary n-cubes. This scheme allows easy calculation of parent and child nodes, and is designed to reduce communication to the area of the network containing participating nodes.

1.4.5 Locality Model

This thesis introduces a method of describing locality for hierarchical cache coherent protocols and incorporates this method in a model. The thesis also shows how the method can be used to accurately predict the longest path traveled per operation and the number of messages sent per operation.

1.4.6 Embedded Model

This thesis also introduces a model for describing the behavior of PHD as embedded into a k-ary n-cube. This model is based on the work of Johnson [13], and models applications, processors, and networks. The model is used to show that the embedding will scale well for applications with moderate locality in situations where the number of cycles needed to process the protocol transactions is small.

1.5 Thesis Overview

The focus of this thesis is the description and the modeling of a hierarchical, directory-base cache coherence protocol. Chapter 2 describes the Protocol for Hierarchical Directories in moderate detail and proposes an embedding of the protocol into a k-ary n-cube. Chapter 3 discusses some of the issues involved in designing hierarchical protocols. Chapter 4 outline the simulator used to test and explore PHD; this chapter also explains the simulator verifier.

Two models were used to study the protocol. The abstract model, which considers the protocol running on an abstract hierarchy, is described in Chapter 5. Chapter 6 extends this model to show how the protocol behaves when embedded as proposed in Chapter 2. Chapter 7 concludes the thesis, outlining areas of future research.

Chapter 2

Protocol Overview

This chapter provides a description of the behavior of the Protocol for Hierarchical Directories. A table listing the exact behavior of the protocol is located in Appendix C. This chapter also briefly outlines a method of mapping a hierarchy to a k-ary n-cube. The next chapter discusses the issues involved in the design of a hierarchical protocol.

2.1 Protocol Description

This section explains the operations used by PHD to ensure consistency while coordinating the global read, write, and test-and-set operations. Section 2.1.1 briefly describes the operation of the protocol. Sections 2.1.2 and 2.1.3 outline the definitions and the notations used in the description of the protocol. Sections 2.1.4, 2.1.5, and 2.1.6 explain the protoco in more detail, describing the read, write, and test-and-set operations, respectively.

2.1.1 Overview

PHD supports three essential global primitives: read, write, and test-and-set. Any number of nodes can have read-only copies of a block in their caches. To find a block, a node asks its parent for a copy. The parent must know which of its child subtrees have copies. If none do, it forwards the message upwards. If one does, the read message is forwarded to it. Read operations can therefore be satisfied locally.

Write operations involve finding all of the copies of a block in the system and deleting them. Only the nodes in the smallest subtree which contains all copies of the block and the write requester are involved in the write process. Acknowledgments of deletion from all of the nodes which previously had copies are combined, and an acknowledge message is sent down to the node requesting the write. The *owner* of the block transfers ownership and a valid copy of the block to the write requester. The test-and-set operation is actually a test-and-test-and-set operation; it is implemented as an optimized combination of the read and write operations.

2.1.2 Definitions

There are two types of directory entries in the hierarchy. The first type, a *leaf* level entry, represents an actual block of cached data and would be found in the memory of a node at the bottom of the hierarchy. The second type, a *parent* entry, is a directory that stores information about which child subtrees have copies of a particular block. The parent entries correspond to memory on some node of the hierarchy which is not at the leaf level.

Every cache entry on a leaf node may be *purgeable* or *unpurgeable*. *Purgeable* entries may be deleted at any time. One copy of every block must never be deleted; the node designated as the owner is responsible for keeping this master copy until ownership is passed on. The only purgeable entries are ones which are in the readable state are yet not owned. A full list of the possible states a of leaf entry is shown in Table 2.1.

A parent entry consists of a vector containing two bits of state for every child subtree, three additional bits of state, and a pointer to the subtree that the current write request, i any, was sent from. The entry will indicate which of four possible states each child subtree is in: *invalid*, *confirmed*, *valid*, or *waiting*. The *invalid* state means that there is no copy of that block in that subtree. The *confirmed* state means that either at least one node in that subtree has a copy of that block or somewhere below a message is propagating upwards indicating that the block has been deleted. The *valid* state means that an operation is occurring in the subtree that will eventually make the subtree confirmed. The *waiting* state means that the subtree has at least one node which is waiting for the result of a read, and that the parent entry needs to send the block down to that node upon receiving the data. The waiting state is employed by the protocol to support read combining. A subtree vector

State	Description
readable_yowner	Entry is readable. Node is owner.
readable_nowner	Entry is readable. Node is not owner.
waiting_for_read	Node is waiting for a read to complete. Node is not owner.
writable	Entry is writable. Node is owner.
waiting_for_write_nowner_npl_nread	Node is waiting for a write to complete. Node is not owner. Invalidation has not yet reached this node. Node may not respond to *read* messages.
waiting_for_write_nowner_npl_yread	Node is waiting for a write to complete. Node is not owner. Invalidation has not yet reached this node. Node has valid value which can be distributed.
waiting_for_write_yowner_npl	Node is waiting for a write to complete. Node is owner. Invalidation has not yet reached this node. Node has valid value which can be distributed.
waiting_for_write_nowner_ypl_nread	Node is waiting for a write to complete. Node is not owner. Invalidation has reached this node. Node may not respond to *read* messages.
waiting_for_write_nowner_ypl_yread	Node is waiting for a write to complete. Node is not owner. Invalidation has reached this node. Node has valid value which can be distributed.
waiting_for_write_ok_yowner_npl	Node is waiting for a write; only needs final ack. Node is owner. Invalidation has not yet reached this node. Node has valid value which can be distributed.
waiting_for_write_ok_yowner_ypl	Node is waiting for a write; only needs final ack. Node is owner. Invalidation has reached this node. Node has valid value which can be distributed.
waiting_for_write_value_nowner_ypl_nread	Node is waiting for a write; only needs ownership. Node is not owner. Invalidation has reached this node. Node may not respond to *read* messages.
waiting_for_write_value_nowner_ypl_yread	Node is waiting for a write; only needs ownership. Node is not owner. Invalidation has reached this node. Node has valid value which can be distributed.
waiting_for_tas	(Full set corresponding to waiting_for_write set).

Table 2.1: The possible states of a leaf cache entry.

Combination	Description
v0_w0_cX	All subtrees are either *confirmed* or *invalid*.
vX_w0_c0	All subtrees are either *valid* or *invalid*.
vX_wX_c0	All subtrees are *valid*, *waiting*, or *invalid*.
vX_w0_cX	All subtrees are *valid*, *confirmed* or *invalid*.
vX_wX_cX	All subtrees are *valid*, *waiting*, *confirmed* or *invalid*.

Table 2.2: The possible combinations of states in the subtree vector of a cache parent entry.

can only have certain combinations of these states, shown in Table 2.2.

All parent entries are marked as either *shared* or *exclusive*. An exclusive entry indicates that all copies are within the current subtree. All entries at the top level node of the hierarchy, by definition, are exclusive. A directory entry on a node which is marked as shared, on the other hand, indicates that there may be a copy outside of the subtree rooted at that node.

A parent entry may be *locked* or *unlocked*. If an entry is locked, then all messages which wish to access it must wait until it is unlocked. Messages which unlock an entry are of course not required to wait. During a write, when a parent entry is locked, there are two more possible state modifiers a node might have: *on_request_path* and *writer_acknowledged*. If the node containing the parent entry is located on a direct path between the writing node and the top of the write, it is *on_request_path*. If the parent entry is on the request path of the write, the final state, *writer_acknowledged*, indicates whether or not the writing subtree has acknowledged the write invalidation. Table 2.3 shows these states.

There are eighteen different messages used by the protocol. They are listed in Table 2.4, and will be explained as they are used.

2.1.3 Notation

Throughout this chapter, diagrams of trees will be shown. These trees are virtual trees, and do not actually exist on the typical architecture PHD will be mapped to. The mapping is described in Section 2.2. Except where noted, the figures only consider a single cache block.

State	Description
S_U_NOP_NGA	Entry is also contained in another subsystem(*shared*). Entry is *unlocked*.
E_U_NOP_NGA	Entry is only contained in this subsystem(*exclusive*). Entry is *unlocked*.
S_L_NOP_NGA	Entry is also contained in another subsystem(*shared*). Entry is *locked*. Entry is not located on path from writer to top node for the write.
S_L_YOP_NGA	Entry is also contained in another subsystem(*shared*). Entry is *locked*. Entry is located on path from writer to top node for the write. Entry has not yet received acknowledge from the writing subtree.
E_L_YOP_NGA	Entry is only contained in this subsystem(*exclusive*). Entry is *locked*. Entry is located on path from writer to top node for the write. Entry has not yet received acknowledge from the writing subtree.
S_L_YOP_YGA	Entry is also contained in another subsystem(*shared*). Entry is *locked*. Entry is located on path from writer to top node for the write. Entry has received acknowledge from the writing subtree.
E_L_YOP_YGA	Entry is only contained in this subsystem(*exclusive*). Entry is *locked*. Entry is located on path from writer to top node for the write. Entry has received acknowledge from the writing subtree.

Table 2.3: The possible states of a cache parent entry.

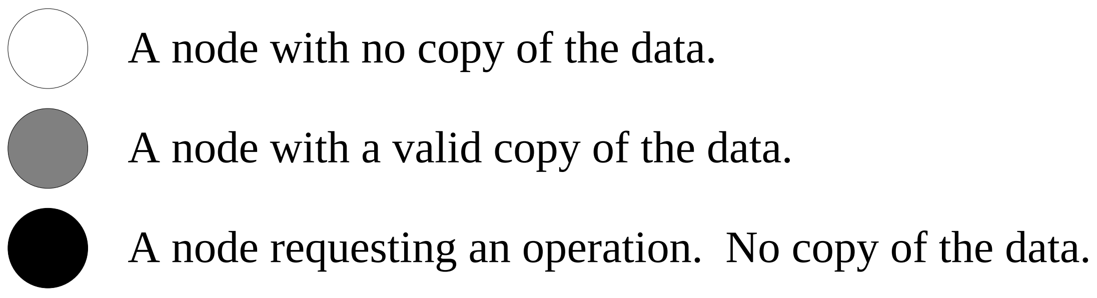

A node with no copy of the data.
A node with a valid copy of the data.
A node requesting an operation. No copy of the data.

Figure 2.1: This figure explains the symbols used throughout the chapter. Note that a node with a "valid" copy of the block is an imprecise description, basically implying that the node, if leaf, has a copy of the block in a readable or writable state. If a grey node is not a leaf node, it is assumed to have at least one subtree in the confirmed state.

Message	Description
find_lowest_common_for_read	Look upwards for nearest node with value.
redirected_find_lowest_common_for_read	Look again; failed in current subtree.
read	Walk downwards to a node with value.
find_lowest_common_for_write	Look for lca of all nodes with value.
lock	Lock all nodes below with value.
ack	All copies below are invalid.
ack1	All copies below except writer's are invalid.
throwing_away	Subtree below invalid; once was confirmed.
change_to_exclusive	Node is least common ancestor of all copies.
find_lowest_common_for_tas	Look upwards for nearest node with value.
redirected_find_lowest_common_for_tas	Look again; failed in current subtree.
confirm_value	Subtree below has gotten a copy of value.
read_data	Level $= 0$: Send data directly to reader. Level > 0: Send data to waiting subtrees.
unconfirm_value	Subtree below now not confirmed, not invalid.
read_tas	Walk downwards to a node with value.
write_ok	No other copies left in tree.
s_write_own	Ownership transfer message for writes.
tas_failed	Test-and-set failed in initial read stage.

Table 2.4: The messages sent by the protocol.

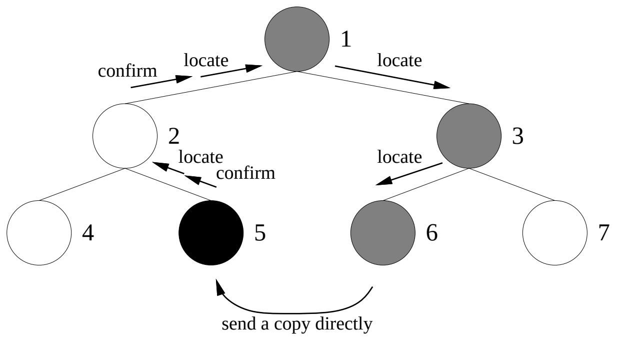


Figure 2.2: This diagram shows the three phases which occur during a read operation. Node 5 wants to read X, so it sends messages to locate X. This first phase, locating a block, finishes when Node 6 is informed that Node 5 wants a copy. At that point, phase two starts, in which a copy is sent directly to Node 5. Finally, in phase three, confirmation that the value arrived is sent to Node 2 and then from Node 2 to Node 1.

At the level of detail of the figures in this chapter, nodes may be in one of three states as shown in Figure 2.1: invalid, valid, and requesting an operation. These states apply intuitively to both leaf and parent entries. Note that the "valid" state for a parent node is most similar to having a confirmed subtree.

2.1.4 Reading

A read to a locally cached block occurs immediately. On a read miss, however, a three-phase operation must be performed, as sketched in Figure 2.2. The first phase locates the nearest block while simultaneously updating the states in the hierarchy. The second phase sends a copy of the block directly to the requesting node. The third sends a confirmation that the node has actually received the copy. There are two possible complications to a read. First, a write may be going on at the same time. Second, the copy chosen to be replicated may be deleted locally before the read request reaches it. Both of these problems are handled by the protocol.

A node which wishes to locate a block for a read sends a *find_lowest_common_for_read* message to its parent. If the parent has no record of the block, it sends the same message up, until a node is found that has it. If the block exists, the message traveling upwards will eventually arrive at a node which knows where the block is. If the block does not exist, the protocol will allocate it automatically or signal an error, whichever it has been configured

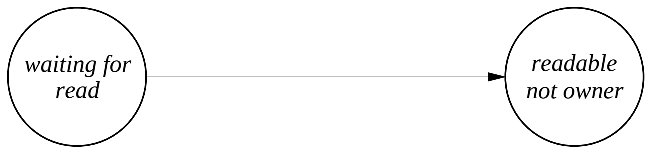


Figure 2.3: The states that a leaf node can enter during a read.

to do.

If the entry at that node is unlocked, and at least one subtree is confirmed but none are valid, the node must update its vector of who has the block and pick one of the confirmed subtrees to send the *read* message down to. If any of the subtrees are valid, the requesting subtree is marked as waiting, and the read process suspends here. When the valid subtrees are changed to confirmed, indicating that they have finished the read, the values are sent down to all waiting subtrees. This mechanisms supports read combining.

When a non-leaf node receives a *read* message, it changes the state of its entry to shared, since the request must have come from outside of the subtree it heads, and forwards the *read* message down towards the confirmed subtrees and leaves. When a leaf entry in a readable or writable state receives a *read*, it sends a purgeable copy (using the *read_data* message) directly to the requesting node. When the requesting node receives the value, it sends a *confirm_value* message upwards to its parent, to confirm that it has received a copy of the block.

Complications to a read can occur when a node deletes a block which some other node is trying to read. This deletion may cause a *read* message to reach a node which no longer knows about the block. In this case, a *redirected_find_lowest_common_for_read* message is sent upwards to find a different record of the block.

In all of the cases described above, if a node is ever reached whose entry for the block being read is locked, the read is temporarily halted. This halting permits the serializatio of reads and writes.

Figure 2.3 shows the states that can occur in a leaf node during a read. When a node wishes to read a block which it has not locally cached, it enters a *waiting_for_read* state and sends a *find_lowest_common_for_read* message to its parent. In a normal read case,

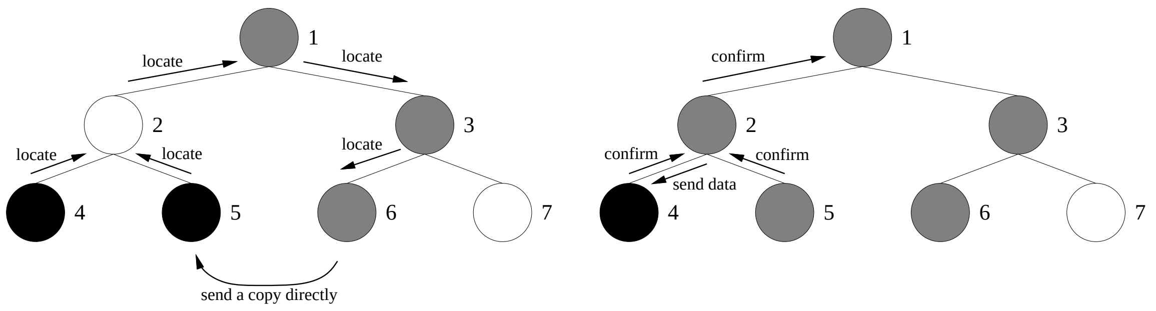


Figure 2.4: This diagram illustrates how read combining occurs. As shown in the left tree, Node 5 requests a read just as in Figure 2.2. At any time between the locate message reaching Node 2 and the confirm message reaching it, Node 4 also decides to read the same value, and sends a locate message to its parent, Node 2. Node 2 records that Node 4 is waiting for that block. As shown in the right tree, when the confirmation message from Node 5 reaches Node 2, Node 2 sends the data from the read to Node 4, which responds with a confirmation. Note that only one confirmation is sent from Node 2 to Node 1; confirmation is sent by Node 2 only after receiving confirmation from Node 5.

the node will be informed by a *read_data* message of the value, and will then enter the *readable_nowner* state. If, on the other hand, a write starts before the read completes, the node may receive a *lock* message before the new value. The node blocks the write from completing by not acknowledging the *lock* until it receives the new value and completes its read.

If several nodes simultaneously try to read a block which is not already widely distributed, the messages will be combined. For example, if three leaf nodes with the same parent try to read block X, they will all send *find_lowest_common_for_read* messages to their parent. When the first message reaches the parent, it will update its entry to record that the node which sent the first message, Node 1, has a valid (but not confirmed) copy, and forward up the request. The second message to arrive will result in Node 2's state being changed to waiting. This time, of course, the parent node does not forward the request upwards. The third message to arrive will result in a change of Node 3's state to waiting. When the value of X is sent to Node 1, it will send that data to the parent as part of the *confirm_value* message. The parent will then send the data to all of its subtrees in the waiting state, in this case Nodes 2 and 3. A similar example of read combining is illustrated in Figure 2.4.

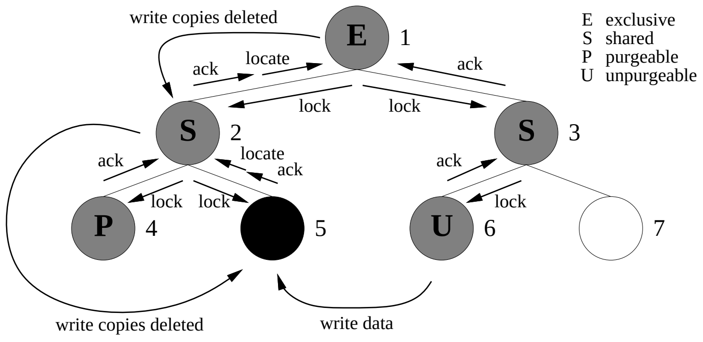


Figure 2.5: This is a sketch of the write process. Node 5 wants to write X, so it sends messages to locate X. When the locate messages reach the lowest common ancestor of Node 5 and all nodes that know about X (in this case Node 1), lock messages are sent down to every node which has X. Each leaf node receiving a lock message deletes its copy and sends an acknowledgment upwards. The owner (6) additionally sends a copy of X to 5. When all of the acknowledgments have been collected, Node 5 is sent a message. Node 5 may now update X.

2.1.5 Writing

A global write involves finding all of the copies of a block in the system, locking them, deleting them, transferring ownership and the current value to the new owner, and then performing the actual write. This process is shown in Figure 2.5.

Several consecutive write requests from a single node to a particular location can be fulfilled quickly and easily. As soon as a node has written to a block once, it has sole ownership and control over that block, and can thus perform consecutive reads or writes locally until another node requests a copy.

A node wishing to write to a block which it does not have a *writable* copy of must first create an entry in the state *waiting_for_write_nowner_npl_nread*, or modify an existing entry to be in the *waiting_for_write_nowner_npl_yread* or *waiting_for_write_yowner_npl* state, as appropriate. The location phase then begins. The node sends a *find_lowest_common_for_write* message to the node above it in the hierarchy. If that node has no record of the block, it sends the same message up.

The locate phase continues until the lowest common ancestor (lca) of the block is reached. The calculation of the lowest common ancestor for a block considers all leaf nodes containing the block and the node requesting the write (see Figure 2.6). Note that

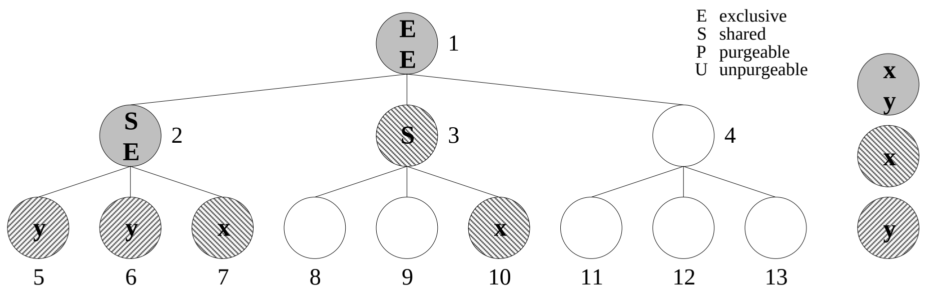


Figure 2.6: The least common ancestor for a block depends not only upon the block, but also upon where the writer is located. The lca for block y, cached in Nodes 5 and 6, from the point of view of Nodes 5, 6, or 2, is Node 2. From the point of view of all of the other nodes, the lca for y is Node 1. The lca for block x, from any node's point of view, is Node 1. The definition of the lca for a block from the perspective of a node N is that the lca is the first node whose entry is tagged exclusive in a path starting from N going up to the node at the highest level of the hierarchy. Nodes 1 and 2 label block y as exclusive. Nodes 2 and 3 label block x as shared. Node 1 labels block x as exclusive.

the lca node is the highest node in the hierarchy that is involved in the write. An lca node receiving a *find_lowest_common_for_write* message locks its entry, signifying the beginning of the lock phase and ensuring the serialization of writes. It then sends down *lock* messages to every node which has a copy of the block.

Most nodes receiving the *lock* message will have copies of the block being locked.[1] Non-leaf nodes with records of the block lock their entries, and forward down the *lock* messages to all those nodes below them which have the block.

All of the leaf nodes with copies of the block will receive *lock* messages. Those with purgeable copies just erase the copies and send an *ack* message up immediately. The old owner of the block will have an unpurgeable entry. This owner node first sends a copy of the block directly to the node requesting the write, then deletes its copy and sends up an *ack* message. The main purpose of the copy message is to transfer the ownership of the block and to give the writer the old value to distribute if necessary to the reads serialized befor
the write. This prevents a deadlock situation, which will be described more fully later.

The node that is in the state *waiting_for_write* will also have an unpurgeable version

[1]If a node has deleted a block and the information that this has happened is still propagating upwards, some nodes may receive a *lock* message but not have a record of the block. In this case they immediately send up an acknowledgment of deletion.

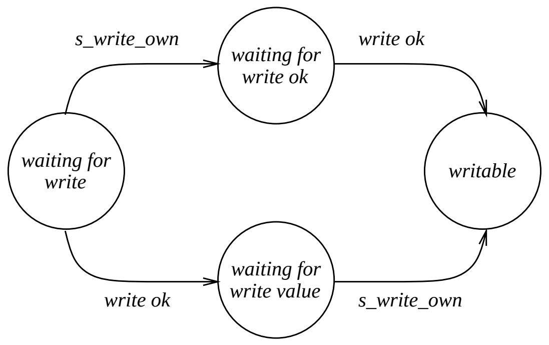


Figure 2.7: The finite state machine describing a leaf node entry during a write. These states are all approximate; the exact transitions are described in Table C.5.

of the block. This is the node that requested the write. If two writes were requested at approximately the same time, the one who the *lock* message records as the writer is the one that won the race. The other write will be waiting on a locked entry somewhere. The winning node sends up a special acknowledgment, *ack1*, indicating that it is on the path of the write.

The streams of *ack* and *ack1* messages signal the combining phase of a write. This phase is used to ensure that every copy of the block is deleted before any modifications are performed. Each parent node collects *ack* and *ack1* messages until it receives responses from all of its involved children. If the parent node is on the direct path between the writing node and the lca node, it sends up an *ack1* as soon as there is only one subtree below it with a copy, and it has already received an *ack1* from a subtree. The single remaining subtree contains the node which requested the write. If the parent node is not on the write path, it waits until it receives *ack* messages from all subtrees directly below it which had copies, and then deletes its record of the block. The lca node for the block will be on the path for deletion. When it receives the last acknowledgment, it sends a *write_ok* message down to the node requesting the write and unlocks its cache entry. The *write_ok* message travels through all nodes that were on the write path, unlocking them as it descends.

The node requesting the write will receive two final messages, in an indeterminate order. One is the *s_write_own* message, which contains the value of the data and performs the ownership transfer. The other is the *write_ok* message, which indicates that all other copies on the system have been deleted. Only after receiving both messages does the node change the state of the entry to *writable* and perform the write. Figure 2.7 shows the finite state machine representing this sequence.

In the protocol, any read in progress when a write reaches a certain point will complete before the write does. In particular, when a *lock* message reaches a leaf node in the *waiting_for_read* state the lock will be delayed at the node until a value is actually sent there. After the read, the block is purged from the cache, and an acknowledge is sent. In order to avoid deadlock, the protocol always allows at least one node to distribute the old value on demand. Before the ownership transfer, the owner will have the value for distribution; after the ownership transfer the writer will have and distribute the value. Reads attempting to complete during the later stages of a write often end up being sent to the writer.

2.1.6 A Synchronization Primitive

Although the read and write primitives which operate on shared memory ensure consistency, they do not provide a simple method for synchronization between nodes. We have therefore included the test-and-set (TAS) instruction. This primitive is included for completeness, and could be implemented better by a variety of methods [11] [20].

The TAS is a combination of a read and a write. First a read is performed, up to the point where a copy of the value is located. If the copy is non-zero, the TAS fails, and a *tas_failed* message is sent to the requesting node (see Figure 2.8). If the copy's value is zero, the write phase begins. The "write" continues just until the requesting node would be about to perform the write. At this point, the value is again checked. If it is non-zero, no value is written. If it is zero, the TAS completes successfully. This second check must be performed in order to ensure the atomicity of the test-and-set. A diagram of a successful TAS is shown in Figure 2.9.

Although the test-and-set primitive was designed with barrier synchronization in mind, it is still not as good as a mechanism specially designed for barrier synchronization. Synchronization using the provided test-and-set primitive does a preliminary read before attempting to gain ownership of the test-and-set variable in order to reduce useless thrashing. To perform a barrier synchronization, however, every node will still have to gain ownership of the

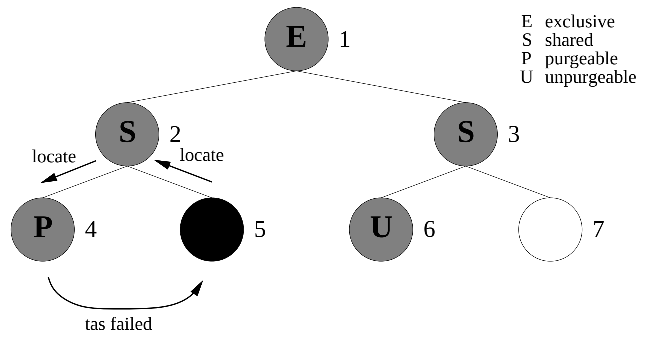


Figure 2.8: The diagram shows the steps of a test-and-set which fails in phase one. Node 5 tries to perform a TAS on X. Node 5 does not find X locally, and sends a locate message up to Node 2. Node 2 knows where a copy is, so sends a message down to Node 4. Node 4 examines X, and finds out that X is non-zero, implying that the test-and-set has failed. Node 4 therefore sends a message to Node 5 telling it that the tas has failed.

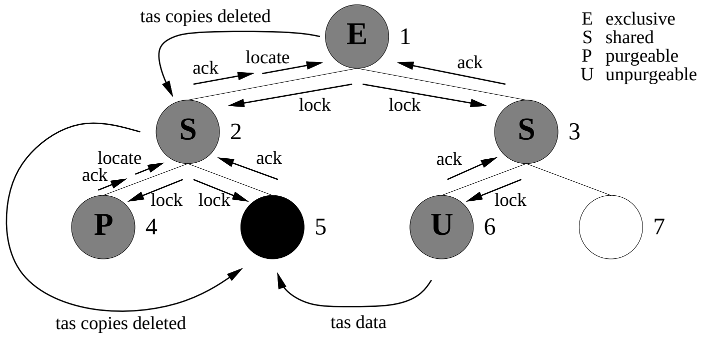


Figure 2.9: The diagram shows the steps of a test-and-set which completes successfully. The first part is the same as in Figure 2.8 and is not repeated here. After Node 4 verifies that X is 0, it begins the same steps as would happen in a write. The lca node for X (Node 1) is found. It sends lock messages to all nodes which have copies of X. Those nodes delete their copies, and send acknowledgments upwards. After both the value and the final acknowledgment are sent to Node 5, it checks to make sure X is still 0. If so, it sets X.

block at some point. On a machine such as the J-Machine, the separate message facility can be used by an application to build a more efficient barrier synchronization.

2.2 Physical Layout

The hierarchy is mapped to a physical machine in such a way as to realize hierarchical locality as physical locality. The mapping is also designed to split the address space so as to increase bandwidth and prevent bottlenecks at higher levels of the tree. The mapping is designed to work for all k-ary n-cubes, although the protocol may not perform well on configurations such as high-dimensional cubes.

Each processor stores part of the global address space. The locations of every block are stored in a hierarchical directory, forming the virtual tree described in Section 2.1. virtual tree is composed of virtual nodes, each of which may be mapped onto several physical processors. This mapping allows us to form a different physical tree traversal patterns: one for each set of addresses.

2.2.1 Hierarchical Directory

A directory records which nodes have copies of blocks. In a multiple level system, every parent node at level 1 knows which of its child nodes have copies of a block. Every parent node above level 1 stores which of its child nodes are the roots of subtrees containing copies of a block at their leaves. To locate a block that is not stored locally, a node sends an inquiry which will travel upwards until a copy is found.

In order to increase bandwidth, the directories of the virtual nodes at every level are split onto many physical processors. This splitting is shown in Figure 2.10. Each leaf node is mapped directly onto a unique physical processor. The parent (non-leaf) nodes are distributed equally onto all processors of the machine, while maintaining locality. The top node of the tree is distributed onto all nodes of the machine. The mapping is also designed to promote locality: every physical processor stores part of a node from every level. This implies that some requests can traverse the entire tree while staying local to a processor.

Figure 2.11 shows a hierarchical directory embedded into a two-dimensional mesh network. The highest level of a virtual tree consists of a single node. Its four children are th

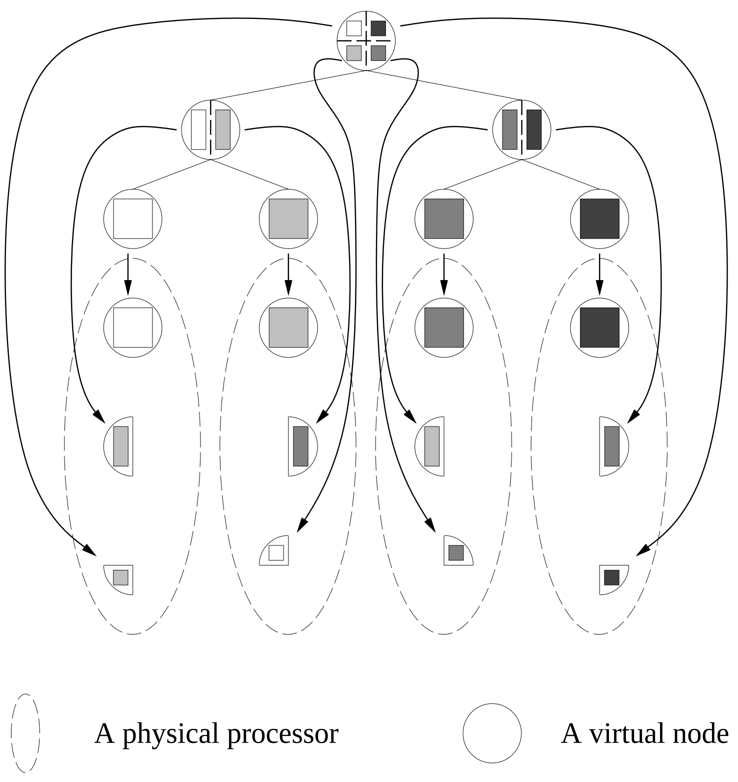


Figure 2.10: The virtual address tree is split to increase bandwidth. In this case, a 3 level radix 2 tree is mapped onto a 4-ary 1-cube. Each virtual leaf node is stored on a unique physical processor. The first-level parent nodes are each split onto two physical processors (forming sub-*lines*). The second-level parent nodes (in this case the root node) are split onto four physical processors (forming a sub-line of double the size of the first-level ones). I a k-ary 1-cube, the parent of a leaf node will be located in the same two-processor sub-line as leaf node itself. The grandparent of a leaf node will be located in the same four-processor sub-line as the leaf node itself. For every additional level in the radix two tree, the number of processors needs to be doubled.

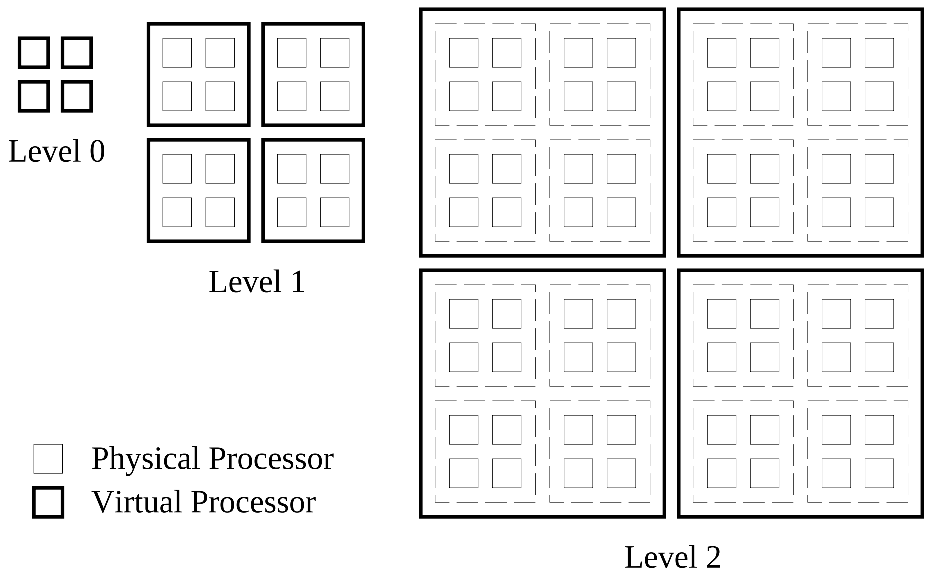


Figure 2.11: A conceptual view of a two, three, and four level tree. Each group at a level becomes a single node at the next highest level.

---

four level 2 nodes which compose that single node. The four children of a level 2 node are the four level 1 nodes, and of a level 1 node are four level 0 nodes. Level 0 nodes correspond to leaves of the tree, and are physical processors. Each virtual parent node can contain information about any block, yet each of the physical processors composing a parent node can only hold some predetermined subset of the blocks, based on the block addresses. This mapping results in physical locality, because any messages traveling in the hierarchy will always stay within sub-cubes.

The hierarchical directory can also be viewed as consisting of multiple trees. As an example, consider the mapping of a virtual 3 level, radix 4 tree to a physical 4-ary 2 cube shown in Figure 2.12. The collection of nodes that can store a particular address forms a complete tree. In this example, sixteen different trees are formed, each rooted at a different processor. Because the trees for different addresses are different, there is no bottleneck at the “top node” of the hierarchy.

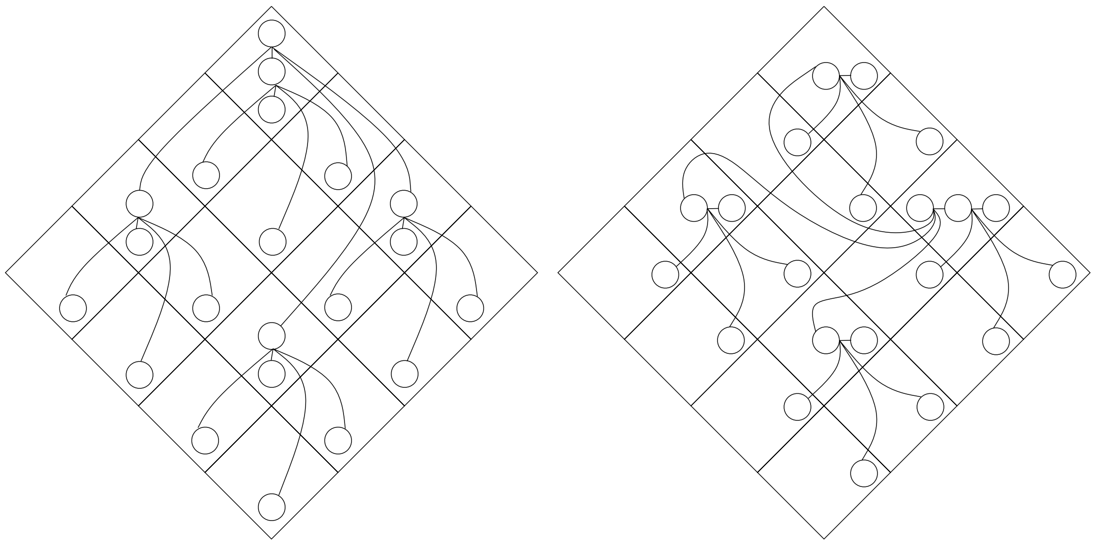


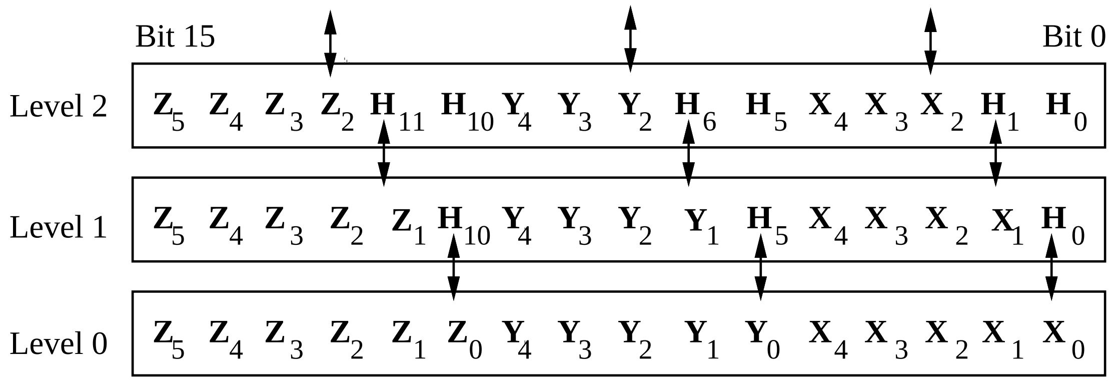
Figure 2.12: Trees embedded into a 2-dimensional grid. Only two out of sixteen are shown.

2.2.2 Mapping Function

The mapping function is used to calculate the node number of the parent (or child) of a node, given an address, a level in the hierarchy, and the current node number. This particular mapping function only works for machines whose radices are powers of two.

A global address consists of two parts. The *map* part must encode the information necessary for the mapping function to operate, such as a global processor ID. The *key* part is used to distinguish among addresses with identical map parts, such as local addresses on a single processor. There are no restrictions as to how the map and key parts may be combined to form a global address.

Any node can store any block at the leaf level. To calculate the parent for that block, replace some part of the current node number with the map part. For example, on the J-Machine, which has a three-dimensional mesh network, take the low bits of the node number's three coordinates and replace these three bits with the corresponding three bits from the global address. This strategy implies that the highest level nodes will store only blocks whose map part of their addresses equals their node number. Figure 2.13 illustrates



Bit 15
Bit 0
Level 2
Z_5 Z_4 Z_3 Z_2 H_{11} H_{10} Y_4 Y_3 Y_2 H_6 H_5 X_4 X_3 X_2 H_1 H_0
Level 1
Z_5 Z_4 Z_3 Z_2 Z_1 H_{10} Y_4 Y_3 Y_2 Y_1 H_5 X_4 X_3 X_2 X_1 H_0
Level 0
Z_5 Z_4 Z_3 Z_2 Z_1 Z_0 Y_4 Y_3 Y_2 Y_1 Y_0 X_4 X_3 X_2 X_1 X_0

Figure 2.13: This figure demonstrates the mapping function used for a 3-dimensional mesh. The three node numbers indicate which nodes can store address H. H could be stored on any leaf (level 0) node. To calculate the level 1 node that H could be stored at replace the low three coordinate bits, one from each dimension, with their corresponding value from H. To calculate the level 2 node, replace the next highest three coordinate bits, *etc.*

the J-Machine mapping function.

This mapping function will keep messages confined to physically small areas whenever possible. A message being sent from the leaf level to the first level will by definition have a destination somewhere within the eight (more generally, 2^n) node cube which includes the sender. Assuming bidirectional links, the farthest such a message would need to travel is three (n) hops. More generally, the farthest a message will have to travel to communicate between levels i and $i+1$ is $n \cdot 2^i$ hops. On average, assuming random destinations, the distance is only $\frac{n}{2} 2^i$ hops. There will be more discussion of this embedding in Chapter 6.

2.3 Summary

This chapter described the operations of PHD. PHD supports cache coherent read, write, and test-and-set operations. Read requests are satisfied in the smallest subtree containing both the requester and a copy of the requested block; only three sets of messages are sent up or down that subtree. Write requests are confined to the subtree containing the lowest common ancestor of the requester and all copies of the requested block; four sets of messages traverse the hierarchy, two of which fan out to all nodes with copies. The test-and-set request is implemented as an optimized combination of read and write requests, and implements a test-and-test-and-set operation.

This chapter also described a mapping of PHD to arbitrary k-ary n-cubes. The mapping translates hierarchical locality into physical locality. The mapping also statically sprea higher-level tree nodes onto many physical processors, in order to increase bandwidth and prevent bottlenecks at the top of the tree.

Chapter 3

Protocol Issues

Many decisions must be made in the design of a complex system. These decisions often involve tradeoffs between space, time, and complexity. This chapter discusses some of the tradeoffs that were made in the design of the Protocol for Hierarchical Directories as well as the consequences of these decisions.

Section 3.1 examines those tradeoffs intended to increase the parallelism in comparison to other hierarchical protocols by increasing the asynchrony. Section 3.2 considers small, easily changeable design decisions that further optimize the performance of the protocol.

3.1 Parallelism

PHD was designed to reduce the serialization of protocol actions by introducing parallelism in the satisfaction of requests. Parallelizing a problem, however, often makes the problem more complex. Many of the choices made in the protocol design therefore significantly increased the complexity of the protocol. Whether there is a commensurate decrease in latency is an open issue to be studied.

3.1.1 Extra Traversals of the Hierarchy

Extra traversals of the hierarchy provide information to a protocol. Avoiding extra traversals of the hierarchy increases both the state necessary to support a protocol and the complexity of a protocol. There are two specific cases of this tradeoff in PHD: one in the read

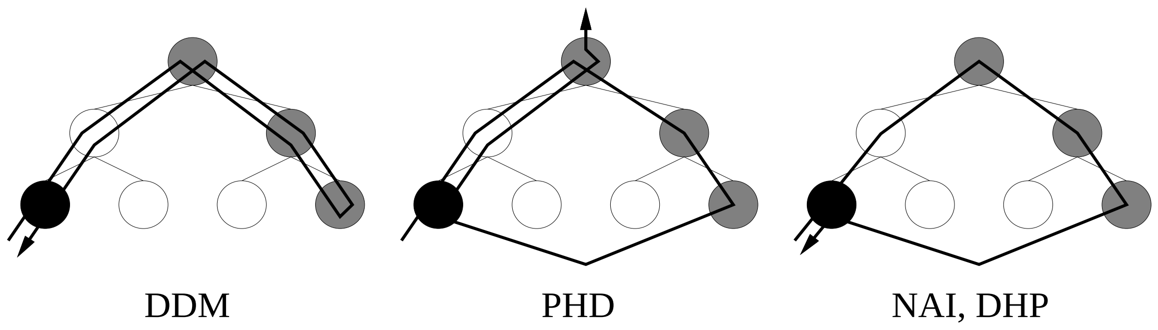


Figure 3.1: This figure compares the number of traversals of the hierarchy needed for a read request for the four different protocols: DDM, PHD, NAI, and DHP. The black node is performing a read request. The grey nodes have copies of the block being requested.

request mechanism, and one in the asynchronous invalidate mechanism. We briefly compare four different solutions to this tradeoff, three of which are part of existing protocols.

DDM The DDM protocol [12] requires more traversals of the hierarchy than do any of the other protocols. This requirement is reasonable given the assumptions of that project: they propose to implement their protocol on a bus-based system, where the hierarchy is fixed in hardware and cannot be circumvented. During a read request, four traversals of the hierarchy occur (see Figure 3.1): first up, to find a node which knows where a copy is, then down, to a node with a copy, then back up and down through the network, updating the interior directories as the read occurs.

There is only partially asynchronous invalidation in the DDM protocol. In order to discard a block, a node must initiate a transaction which carries the data. This transaction will continue to propagate upwards until at least one other copy of the block is found. This system prevents the protocol from deleting the last copy of a block. The transaction must carry the value with it, unlike in the other protocols, in order to be sure that the value is preserved.

This protocol differentiates four read states in the hierarchy for a subtree: invalid, reading, answering, and valid data. These states are updated as the read travels twice up and down the hierarchy, and provide full information to the protocol as to the exact stage of a read. The valid data state only implies that the data has been sent into the subtree, not that the data is still there.

PHD The PHD protocol requires one fewer traversal of the hierarchy for a read request than does the DDM protocol. PHD, as illustrated in Figure 3.1, also routes the read request up and down through the hierarchy, but then sends a copy of the value directly through the network to the node requesting the read, and then updates the hierarchy by a confirmation sent only upwards.

PHD can also completely asynchronously discard blocks. This feature allows most nodes to quickly discard cache entries whenever the caches are full. Unlike the DDM protocol, the value is not carried in the discard message. The use of a special *owner* node guarantees that all nodes will not simultaneously discard their copies. The owner, which is defined as the last node to write to a block, cannot asynchronously discard its copy. If a particular node is the only node to write to many blocks, its cache will eventually fill up. PHD can be extended to solve this problem by adding a message which requests some other node write the value (freeing it from the full node's cache). This solution introduces complicated load-balancing issues not addressed by this thesis.

The combination of these two features introduces complexity to the protocol. Although invalidation is now simpler, because the value is not carried in the deletion message, the problem of the owner capacity overflow has been introduced. The longest path for a read request is now shorter than it was before, but the read-combining path is slightly longer. Reads which have been combined do not receive the value of the data until after the confirmation step of the read request. A read request that has been read combined must, in the worst case, wait through two traversals of the hierarchy (one up and down), one message being sent across, a confirmation being sent up through the hierarchy, and the data finally being sent down to the combined nodes.

PHD also differentiates four read states in the hierarchy for a tree: invalid, reading, waiting for a read combination, and valid data. The valid data state means that the subtree received the data but may have already deleted it.

DHP The DHP protocol [21] both requires the fewest number of traversals of the hierarchy and implements asynchronous invalidation. Together this combination results in a protocol vulnerable to deadlock, because not enough information in the hierarchy is available to tell whether or not a subtree is waiting to receive a block, has already received the block, or has received and already deleted (but not yet propagated this information upwards) the block. This lack of information is used as the read combining mechanism; if a read request reaches a node which is in the process of reading data, it waits there until the data arrives. This mechanism by itself is perfectly reasonable. Unfortunately, when combined with asynchronous invalidation, the mechanism results in deadlock, where two nodes can each end up waiting for the same block, and each also be promising to tell the other when they receive the block. In this case neither request can ever be filled.

The read combining mechanism of this protocol theoretically reduces the path length of the longest combined read. The only problem which can occur is read chains, where a list of nodes is waiting for a block. The values will propagate one at a time; upon receiving the value it has been waiting for, a node forwards that value to its own list of waiting nodes. Those nodes in turn might themselves have lists of other nodes waiting for the same value.

The DHP can only differentiate two states for subtrees in the hierarchy: invalid and valid, where valid implies that the subtree will receive the data, has received the data, or had received (and already deleted) the data. It cannot use any other states because every node is only visited once.

NAI A fourth protocol, not yet proposed, is identical to the DHP except that it eliminates the asynchronous invalidation ability. We call this protocol NAI (No Asynchronous Invalidation). Read requests still take only two traversals of the hierarchy. There are still only two states for subtrees in the hierarchy but the meanings have changed: now the hierarchy keeps track of whether a subtree is invalid or has or will get a particular block. This protocol eliminates the deadlock situation of the DHP by guaranteeing that a "valid" subtree either has a copy of the block or has an outstanding read request which is being satisfied outside of that subtree.

The disadvantage of this protocol is that it requires that nodes reserve enough room in their caches for the blocks that they delete between the time that the deletion is initiated and the time that they receive an acknowledgment indicating that it is safe to perform the deletion.

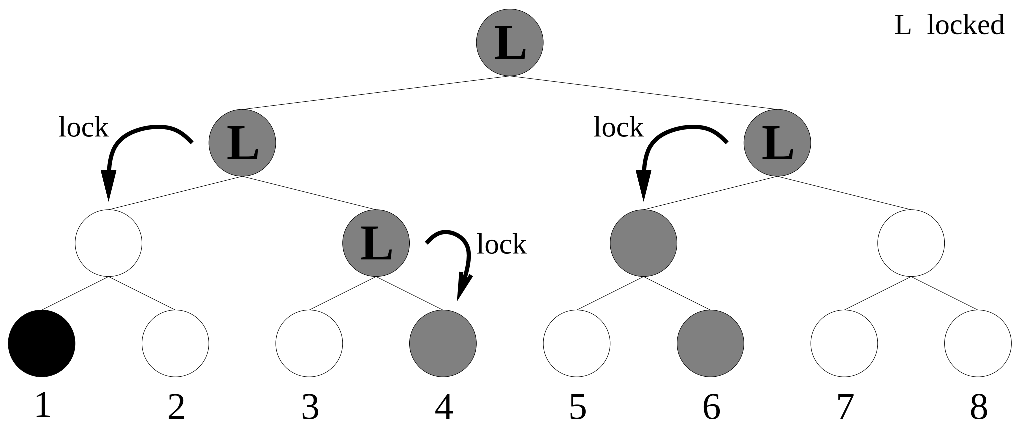


Figure 3.2: This figure shows how the distributed write commit works. Node 1 is in the process of requesting a write. The grey nodes have copies of the object. Consider what happens if Nodes 2-8 all requested read operations at this point, and the lock messages all froze in the network, so that the reads would have time to complete. Nodes 4 and 6 would complete their read requests, because they have locally cached copies. The requests from Nodes 3, 7, and 8 would stall on the write, because they would reach a locked node before reaching a valid node. Node 5, on the other hand, would be able to complete its read, because a valid node above it is still unlocked. Node 2 would be able to complete its read, because its request is coming from the writing subtree.

3.1.2 Distributed Write Commit Point

The commit point for a write is distributed in PHD. A write waits until all reads *in progress* finish. After a particular time, newly starting reads will be stalled until the completion of a write; the calculation of this time is distributed. As soon as a lock message reaches a node, no read originating from any invalid subtree below it will be satisfied until after the write. The one exception to this rule is that reads coming from nodes in currently writing subtrees will be allowed to complete as well. An example of the various cases of reads and writes interacting to form the commit point is shown in Figure 3.2. This scheme supports sequential consistency [14], but also adds complexity to the protocol.

The DHP protocol, on the other hand, does not make any guarantees about reads making progress. It is possible that a read in the DHP can be indefinitely delayed by a series of write requests coming from other nodes; the read will spend all of its time searching the machine for a valid copy.

3.2 Design Decisions

Several design decisions which were made in the construction of the Protocol for Hierarchical Directories could be easily varied. These decisions involve the read combining mechanism, the write invalidation mechanism, and the invalidation mechanism.

3.2.1 Read Combining

The design of the read combining mechanism is another example of a time-space tradeoff. The read combining of PHD delays read requests from later requesting nodes until the requests from earlier requesting nodes have been answered. This delay prevents the later requests from sending another set of location and data transfer messages. Another way to implement read combining is to use address-specific delaying queues which are examined whenever a block's directory state on a node changes. This strategy saves some bits in the state of each node, because a slot to store the waiting status of every subtree for every block is no longer needed.

Using these queues provides two methods for deciding what to do when a lock message reaches a directory node which has children waiting for the value. The first is to send the lock message to all waiting child subtrees, as in the current version of PHD. Unlike PHD, however, this scheme requires that a lock message always check the delaying queue before continuing, in order to find out which subtrees need copies of the value. The second scheme is to not lock waiting subtrees; the reads coming from those subtrees are considered to happen after the writes.

Another interesting question to consider about read combining is whether it is worthwhile at all. Without read combining the protocol becomes substantially simpler. It is not clear how often nodes request the same value nearly simultaneously, except for special synchronization variables; these could be handled separately.

3.2.2 Read Combining Whenever Possible

The read combining mechanism of PHD combines two read requests whenever they occur nearly simultaneously. When a read reaches a node with a subtree that is already reading that block, and that node has no subtrees which definitely have copies of the block, read

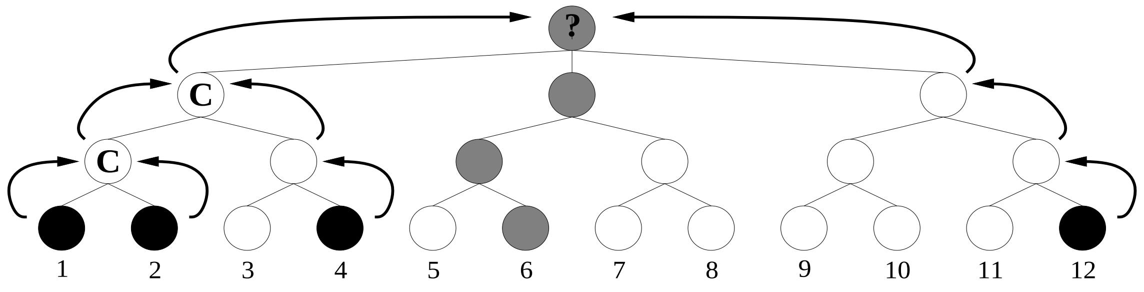


Figure 3.3: This figure shows the two possibilities for read combining. In both methods, Node 1 request a read operation. Since it has no copy of it, the request is sent up to the first node that has it, in this case the root node. Now Nodes 2, 4 and 12 issue read requests to the same block. Node 2's request waits at its parent, as explained in Chapter 2. Similarly, Node 4's request waits at its grandparent. The issue is what happens to the request from Node 12. The request could be sent down the path to Node 6, like Node 1's request was, or the request could be combined.

combination occurs. The question is what to do in the situation, shown in Figure 3.3, where a read request propagates up to a node that has both a subtree in the middle of a read and a subtree with a definite copy of the block. If a new read request is sent down to the subtree with a copy, one-fifth of the protocol table (shown in Table C.11) will be no longer reachable and can be eliminated, because the combined vector state vX_wX_cX (subtrees may be invalid, valid, waiting, or confirmed) can no longer occur. The other possibility for implementation is that the read request be combined, and thus forced to wait until the first read completes, when it will be sent its value. Both versions of the protocol have been simulated. The scheme which combines read requests performed better on the studied synthetic traces and is currently implemented in the system

3.2.3 Write Invalidate versus Write Update

Another interesting issue is whether to invalidate a block or to update it with the new value when a remote write occurs. PHD could be modified to use an update scheme, instead of an invalidate one. For performance, a mechanism to periodically remove unused copies of blocks would be required. Without this capability, writes would involve all nodes who had ever read the block and whose caches had not subsequently chosen to discard the block.

Write update could be extremely valuable in some situations, such as where a few nodes were constantly sharing data.

3.2.4 Non-leaf Invalidation

The protocol does not currently address the issue of a full cache in a non-leaf node. When non-leaf nodes fill up, we cannot simply discard the non-leaf values. Scott and Goodman have addressed this problem in [24]. Their solution is *pruning caches*, which could be adapted to work in PHD. In their pruning cache scheme, non-leaf directory entries may be discarded when a cache is full. Pruning caches store information about where a block does *not* reside rather than where it does reside. In their protocol, if a lock message ever reaches an invalid entry located in a node whose parent has a valid entry, lock messages must be broadcast to all children. Scott and Goodman have determined that "pruning caches with a modest hit rate significantly reduce the invalidation traffic." They also found, in their simulation studies, that when a cache filled up and they needed to discard an entry, "it is better to suffer increased invalidation traffic when the line is written than to prematurely invalidate the line."

3.3 Summary

This chapter examined some of the decisions made in the design of the Protocol for Hierarchical Directories. Some of these decisions are easily changeable implementation issues. Others, such as the number of traversals that should be made of the hierarchy, expose major differences between PHD and other protocols.

Although the minor decisions could be easily isolated and tested to determine which performs better, the major ones cannot be tested in isolation, as they are not necessarily separable from each other. In order to determine the benefits of these major decisions, we must compare the performance of PHD and the other hierarchical cache coherence protocols for a variety of benchmarks.

Chapter 4

Simulator

We wrote a simulator to model the operation of the protocol running on a computer, such as the J-Machine, with a k-ary n-cube network topology. The simulator currently models machines of 64 nodes with two or three dimensions. The simulator is trace-driven, taking as input a statically scheduled list of memory references and simulating them by following the protocol. It outputs a log file detailing the steps it took. We also wrote a verification program which takes the output of the simulator and verifies that it follows a legal ordering of events.

4.1 Overview

The simulator serves two purposes: first, it tests the protocol and second, it provides a platform for studying several characteristics of protocol behavior. In particular, it provi
a method to examine the number of messages sent per operation, the longest path traveled per operation, and the average height in the tree reached per operation for different types of operations. The results of this study are in Chapter 5. The simulator was not designed to support an analysis of how the protocol behaves when burdened by network constraints and different costs for different activities. An analysis of these issues is located in Chapter 6.

The simulator operates at the message level; one unit of simulated time is the time a message takes to travel one hop between two adjacent nodes. The time a message takes to travel from node A to node B is therefore equivalent to the distance in hops between nodes A and B. Operations which can be satisfied locally, such as a local read, write, or test-and-set, occur instantaneously on a single node. Message processing for a node, on the other hand, takes a constant amount of time, 10 hops, during which the node is *busy* and can process no new events.

Time and event sequencing is represented by an event-driven queue. The queue represents a range of time. Each slot in the queue corresponds to one particular time, and contains a list of events to occur at that particular time. There is one global queue for the simulation plus one local queue per processor.

Events are removed from the queues and processed according to their type: messages or operations. The simulator supports local allocation, read, write, and test-and-set operations. It additionally supports all of the types of messages specified by the protocol. The global event queue also supports printing events, cache-emptying events, warm start events, and memory-dumping events.

In addition to listing every operation as it completes, the simulator can be configured to print any of the following log information: events processed (as they occur in the event queue), messages processed, and messages sent. The simulator can be configured to print out many different types of statistics about the protocol and its operation.

4.2 Data Layout

The "global" memory of the system is scattered throughout the nodes. Each node has a section of its memory devoted to storing the data blocks that it has copies of, or knows about. It also has a section which contains mappings from an {address, level} pair to a pointer to the data block stored in the data section.

There are two types of entries which might be pointed to from the data-mapping table. The first type is a leaf entry. It represents an actual block of data, and corresponds to a block of memory which would be found in a node located at the bottom of the hierarchy. The second type is a parent entry. A parent entry stores information about which subtrees have copies of particular blocks. These entries correspond to memory which would be found on a node of the hierarchy not located at the leaf level.

A leaf cache entry, shown in Table 4.1, takes up $N+2$ words, where N is the line size.

31																															0
State					Word																										
Address																													Level		
Data (N words)																															

Table 4.1: The parts of a leaf cache entry.

31																															0
State			VWC Vector															Writer													
Address																													Level		

Table 4.2: The parts of a parent cache entry.

The first word contains the state of the entry, as described in Chapter 2. It also contains the information, during writes, of which word in the cache line is being written. The second word encodes the global address of the object and the level. The final N words are the value of the block starting at the address. Because only the address representing the start of the block is important, this implementation hides the level in those redundant bottom address bits.

A parent entry is any entry which does not correspond to a leaf of the hierarchy. It is always composed of exactly two words. The first word contains the bits specified by the protocol, as well as a vector of up to sixteen bits indicating which subtrees have copies of that object. The vector contains two bits per subtree. The writer field, which is used only during writes by nodes located on the path between the write requester and the lowest common ancestor of all copies, stores the index of what subtree is performing the write, so that lost read requests can be routed to the writer, as described in Chapter 2. A 32-bit entry can actually store up to 12 subtrees, although only the capability for eight is in the current simulator version. The second word encodes the global address of the object and its level, exactly as in the leaf entry.

The simulator manages the simulated heap by using a memory allocation manager. The manager prevents memory from being fragmented by rearranging memory whenever a block is freed, and throwing away purgeable blocks when necessary. The simulator also provides a method for removing blocks from the cache on demand from the input, in order to test the protocol.

4.3 The Simulator

The main parts of the simulator are the node model, the network model, and the event-driven queues.

4.3.1 Node Model

Only the state of a node essential to the operation of the simulator is modeled. Every node has a `nodeid`, a local event queue, a set of delaying queues, and memory. Associated with the memory is a table which stores mappings between {address, level} pairs and pointers into the memory. In an implementation on an actual machine, the table would be part of the memory.

Because the processing of a message in a node occurs in a single time-step, part of the state of a node stores whether or not a node is busy and, if so, for how long. A special event, *node_done*, is added to the local event queue at the time when a node should finish processing the current message. A node will perform no actions in the meantime.

Each node also stores a special association list recording the value to be written for any ongoing write. This information would normally be stored directly in the instruction stream

Ideally, each node can timeshare among several different processes. When a process makes a global memory reference which is not locally satisfiable, it suspends while the reference is filled. In the meantime, other processes can run.

The simulator models this ability by allowing n global requests per processor to be occurring simultaneously, where n is an execution-time parameter. Operations which have already been read in from the input trace are placed in a reference queue which is part of the node model, but is disjoint from the event queues. Whenever an operation on a node completes, the queue for that node is checked. If the next operation on that queue is due to happen in the future, it is scheduled. If the next operation was supposed to happen already, it is started. If there is no waiting operation, the parser is invoked to read more input.

4.3.2 Network Model

The network of the J-Machine is a three-dimensional mesh. The simulator models this network, or optionally a two-dimensional mesh, in a completely unloaded condition, *i.e.* under a zero congestion situation. Message delivery takes time proportional to the distance between the nodes along a Manhattan route: first the X direction is followed, then the Y, then the Z. A message is "sent" at the end of the period of time corresponding to any processing that a node is doing.

We have chosen parameters such that each hop in the network takes one-tenth of the time to process a message. This models a system with balance between computation and communication for fine-grained processing. The longest message sent is $4 + N$ words (N is the line size); the shortest 2 words. All messages are approximated as being the same length for purposes of arrival time. If the destination of a message is the node that generated the message, the transmission is suppressed and the computation continues immediately.

4.3.3 Event-Driven Queues

Time is implemented as a circular list of queues. At the start of a simulation, the simulator reads a block of the input and schedules the specified events. It places each event in the queue entry representing the appropriate time, creating entries as needed. It then begins processing the queue. The simulator processes input on a node by node basis when any node runs out of operations to perform, as more fully described in Section 4.3.1. When there are no more events in any of the queues, the simulator halts.

Global Queue The global event-driven queue keeps track of the events that are to be activated during each time slice. There are several different types of events. Operations are specified by the input file, and include READ, WRITE, TAS, and ALLOC. As mentioned earlier, there are also various types of debugging events, not necessarily specific to particul nodes, which can be specified.

Local Queue Local event-driven queues are used, one per node, to keep track of node specific events, such as messages, and global events that become local. Messages are generated in response to other messages or operations. These queues correspond most closely to the message queues that would be found on some machines.

Delaying Queue Each node additionally has delaying queues. Events which cannot be satisfied until another event occurs are placed on an address-specific delaying queue, and woken up only when an event referring to that address occurs.

4.4 Verification

The simulator was tested by running a hand-written set of tests, designed to exercise all of the features of the protocol, as well as many sets of machine-generated synthetic address streams. The simulator also contains self-consistency code, ensuring that an error is signalled if state and message combinations which are illegal occur. A verifier was written in order to help verify the simulator.

4.4.1 Verifier

We wrote a verification program which takes the output of the simulator and ensures that the output sequence of reads, writes, and test-and-sets is a legal ordering of the requested events. The verification of the verifier was done for a large set of hand-crafted test cases. The rules that the verifier obeys are as follows:

- Any read that finishes before a write operation starts will see the old value.
- After a write operation finishes, the value changes, and any read that starts gets the new value.
- Any read that starts before a write operation finishes and finishes after the write operation starts may see the old or new value.
- A test-and-set may only complete successfully if the value of the data is zero at the point when the set would occur.

4.4.2 Internal Checking

The simulator is peppered with assertions which check for illegal states and combinations. The simulator also has an option allowing one to choose maximum bounds on a random interval for ejecting values from the cache. When this interval is set to one, values are ejected from the cache one time unit after they are placed there, allowing for a thorough testing of the protocol.

4.5 Summary

This chapter describes the simulator used to experiment with PHD. The simulator implements the full protocol plus certain extensions, such as local allocation and optional automatic allocation on uninitialized data. The simulator is trace-driven, and can gather many types of statistics for studying the protocol.

The simulator has been used to test the protocol; additional features for debugging include printing events and cache-emptying events. A special verification program was also designed to ensure that the protocol keeps the memory consistent.

The simulator runs 3,999,800 cycles in just under two hours. This represents 4096 allocation requests, 384,417 read requests, and 255,583 write requests, all resulting in total of 1,496,929 messages being sent. Each node was allocated 0x3000 words of memory for this simulation. This time measurement took place on an unloaded Sparc II with 32 megabytes of DRAM, accessing only locally stored files, and was typical of how the simulator was actually run.

Chapter 5

Abstract Analysis

This chapter presents an abstract model of the Protocol for Hierarchical Directories and then uses the model to show the effects of locality and machine size on several characteristics of the protocol. The model is shown to be valid using results generated by the simulator described in Chapter 4. The model is used to study the average height per operation, the longest path of messages traveled per operation, and the number of messages generated per operation for machine configurations too large to simulate. The results from this chapter become the inputs of an embedded model, described in Chapter 6, which addresses the protocol behavior when it is mapped onto a specific architecture.

Section 5.1 describes the model and a new method of representing the amount of locality in an application. Sections 5.2 through 5.5 discuss the applications and methods used to validate the model. Section 5.6 presents the results of the study, showing the importance of locality as machines increase in size. An alphabetical listing of all of the variables define in this thesis can be found in Table A.1.

5.1 Modeling Hierarchical Behavior

Before measuring the application-dependent behavior of the protocol, the protocol characteristics to be measured, the application characteristics needed to measure these aspects of the protocol, and a model of the protocol behavior must be defined.

5.1.1 Overview

Since we are primarily interested in understanding how a hierarchical protocol scales as machine size and locality change, we have studied three application characteristics: the average height in the tree a read or write operation reaches, the length in message hops of the "longest" path traversed in order to satisfy a read or write operation, and the number of messages generated per read or write operation.

Because this chapter does not study the protocol as mapped onto a particular architecture, issues such as whether or not the calculated message-generation rate can be sustained due to bandwidth considerations are not considered. Similarly, we are also assuming infinite caches, since finite cache effects complicate the model, clouding the important multiprocessor issues under consideration. Finite cache modeling can always be factored in later [1].

5.1.2 Locality Characteristics

In order to study the behavior of a cache coherence protocol which is highly dependent on locality, we must have some method of expressing the locality present in applications. We propose a representation of locality tailored to studying hierarchical cache coherence protocols[1].

Shared data operations are always caused by node requests. Instead of choosing a node to make a request and following that request up the hierarchy, as in the actual protocol, the abstract model chooses which class of nodes a request occurs in. The actual node that makes the request is unimportant, all that matters is what class that node is in with respect to what other nodes have copies of the block. All nodes in a class have equal-height lowest valid[2] ancestors. Choose the node to make a request as follows (see Figure 5.1 for an illustration): start at the root node of a directory tree, and choose from one of two groups: the invalid and the valid subtrees of the root. If the invalid class is chosen, the process stops. If the valid class is chosen we again choose from two groups: the invalid and the valid subtrees of the valid children of the root. This process continues until an invalid clas

[1]Kirk Johnson greatly assisted in the development of this locality model.

[2]Because all requests are modeled as occurring instantaneously, we consider only two states: valid and invalid. Valid implies that there is a copy in the subtree; invalid implies that there is not.

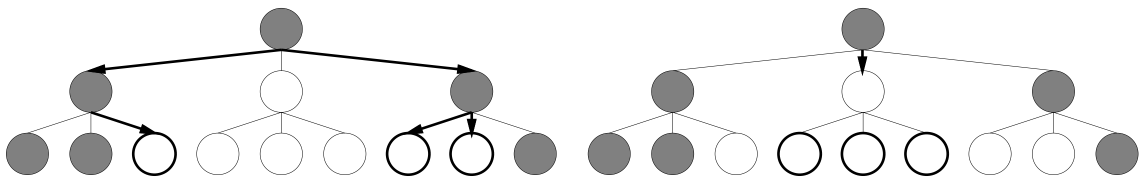


Figure 5.1: This diagram illustrates the selection of node classes performed by the model. The grey nodes are valid. The selection process starts at the root node, where either the group of subtrees who are valid or are invalid are chosen. In the left side of the figure, the valid group is chosen. Because the valid class was chosen, at the next level another selection must be made. At this selection, the invalid group is chosen. This means that the node to make the next request will be in the class of nodes who are not valid, but whose parents are valid. In the right side of the figure, the selection process again begins at the root, where the invalid group is chosen. This ends the selection process; the next request will be made by a node who is invalid and whose parent is invalid but whose parent's parent is valid.

is chosen, or until the leaf is reached. If an invalid class is chosen, all nodes below that class are in the group of nodes that will make the next request. If a valid path down to the leaves is chosen, all leaf nodes which are valid are in the group of nodes that will make the next request.

We calculate the probability of choosing the valid class as follows. We define p_l, the locality parameter of level l, as the a priori probability that the choice will be the valid group when looking down from level l. These locality parameters can be different at each level of the tree. If the request did not a priori come from an already valid subtree, we distribute the probability of where it came from uniformly over all of the children. The locality in an application is thus expressed by this set of locality parameters. For example, in an application where blocks were accessed uniformly by all processors, for all l, p_l would be 0.

This set of locality parameters lets us describe an application's data usage. For greater accuracy, instead of considering an average block, we could consider several classes of blocks with their own sets of p_l.

5.1.3 Model

The performance model calculates the average height in the tree, the longest path traversed, and the number of messages sent per read and per write operation, using the p_l locality

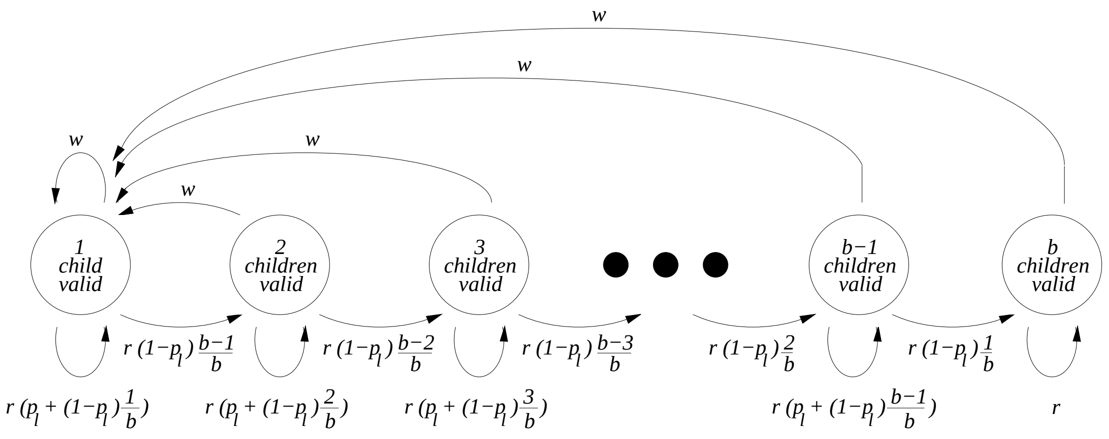


Figure 5.2: The Markov model for counting the number of valid children of a valid parent.

parameters, w, the write ratio, b, the branching factor, and L, the number of levels in the tree.

Define r to be the fraction of reads to shared data and w to be the fraction of writes to shared data, where $r+w=1$.

We first determine v_l^c, the probability that c children of a valid node at level l are valid, by constructing a Markov model, as shown in Figure 5.2. We use the solution to this model to calculate the expected value, c_l, of the number of valid children at l: $c_l = E[\sum_{c=1}^{b} c v_l^c]$. Using c_l, we calculate the value, t_l, the probability of taking a valid path while performing a node selection, looking down from level l. Let $t_L = 1$; this means that the root node will be chosen with probability one and simplifies the equations.

$$t_l = p_l + (1 - p_l)\frac{c_l}{b} \tag{5.1}$$

Height Calculating the expected height a read will reach given the model is straightforward. A read by a node in the class of nodes which are valid will be of 0 height. A read by a node in the class of nodes whose parents are valid but who is not itself valid will be of height 1. The expected height of a read request, $E[h_r]$, is given in Equation 5.2.

$$E[h_r] = \sum_{h=1}^{L-1} h(1 - t_h) \prod_{l=h+1}^{L} t_l \tag{5.2}$$

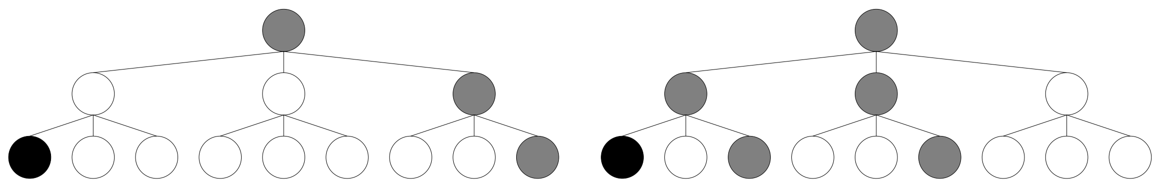


Figure 5.3: In both of these examples, the grey nodes have copies of the value, and the black node is attempting to perform a write. The write operation will have to reach the top level of the tree in order to complete.

In order to calculate the expected height a write request will reach, we must consider not only whether or not the requesting node has a copy, but also whether or not other nodes of any classes have copies. A write must progress upwards in until such a height as which only a single node at each level is valid, and those nodes are all ancestors of the writing node. For example, as shown in Figure 5.3, where the black node is requesting the write operation and grey nodes have copies of the value, a write operation would need to reach the top of the tree in both cases. In the first case, the full height of the tree is needed just t reach any other copies. In the second case, although a shorter height is sufficient to locate a node with a copy of the block, the write operation must reach the top of the tree in order to invalidate the other copies. Keeping these rules in mind, we calculate the expected write height, $E[h_w]$.

$$E[h_w] = \sum_{h=1}^{L-1} h(1 - v_h^{1} t_h) \prod_{l=h+1}^{L} t_l v_l^{1} \tag{5.3}$$

Longest Path The longest path traversed during a read request is the path of the request up the tree, down to the node that has it, and then directly to the requesting node, as shown in Figure 5.4. The expected longest length is $E[l_r]$ thus:

$$E[l_r] = \sum_{h=1}^{L-1} (2h+1)(1 - t_h) \prod_{l=h+1}^{L} t_l \tag{5.4}$$

The longest path for a write is up to the highest node, down to all of the copies, back up to the top, and then down from the top to the requesting node. $E[l_w]$ is given in

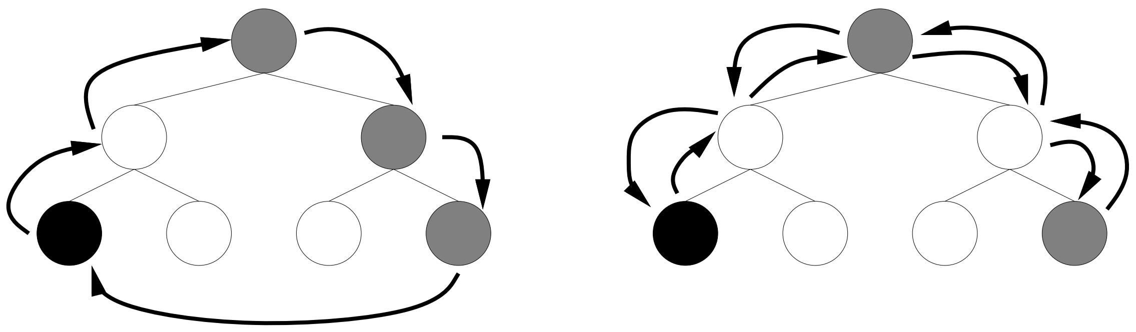


Figure 5.4: The left example shows a read request, the right a write request. In both of these examples, the grey nodes have copies of the value, and the black node is making the request. The longest path traversed is shown for both cases. Note that for the write case, there are other equally long paths not shown.

Equation 5.5.

$$E[l_w] = \sum_{h=1}^{L-1} 4h(1 - v_h^{1} t_h) \prod_{l=h+1}^{L} t_l v_l^{1} \tag{5.5}$$

Number of Messages The calculation of $E[m_r]$, the expected number of messages per read operation, is very similar to that for the expected longest read path. The only difference is that the set of messages sent from the read originator to the top node of the read to confirm that the read has occurred must be added in.

$$E[m_r] = \sum_{h=1}^{L-1} (3h+1)(1 - t_h) \prod_{l=h+1}^{L} t_l \tag{5.6}$$

The expected number of messages per write operation, on the other hand, requires more knowledge than the longest write path calculation. This is because the number of messages depends on how many nodes have read the block since the last write and therefore need to be invalidated. Remember that ${}_l$ is the expected number of valid children of a valid node at a level l. For each non-local write, one set of messages is sent from the requester to the highest node in the tree involved in the write, as shown in Figure 5.5, a full fan-in and fan-out of acknowledgments and invalidates is sent to all nodes with copies, a final acknowledgment is sent down to the writer, and write ownership is transferred to the writer.

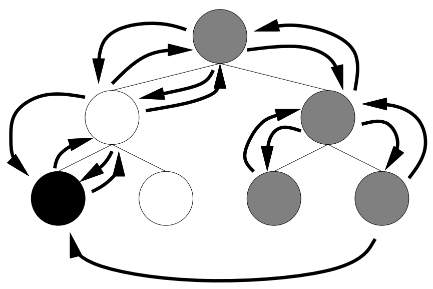


Figure 5.5: This figure illustrates the number of messages sent for a typical write operation. The grey nodes have copies of the value, and the black node is performing the write.

The expected number of messages per write, $E[m_w]$, is thus:

$$E[m_w] = \sum_{h=1}^{L-1} \left(\left(2h+1+2 \sum_{l=1}^{h} \prod_{e=l-1}^{h-1} c_e \right) \left(1 - v_h^1 t_h\right) \prod_{l=h+1}^{L} t_l v_l^1 \right) \tag{5.7}$$

5.2 Applications for Model Verification

Three applications have been employed in the validation of the model. One is a uniform reference pattern, in which every processor is equally likely to reference all of data. Th second mimics a basic relaxation pattern, such as a Jacobi relaxation. The third is a synthetic pattern exhibiting *clustering* behavior: nodes further away from a fixed "home location" of data access it less frequently than do closer nodes.

5.2.1 Uniform

The uniform reference pattern fits the model exactly. Uniformity implies that every node is equally likely to reference any block. Because of this property, there is no locality, so t entire set of locality parameters, t_l, p, should always be zero.

5.2.2 Relaxation

In the particular relaxation we simulated, during every iteration every point of an n dimensional mesh updates its value by a function of the value of its $2n$ neighbors.

Consider a 2-dimensional relaxation implemented on a 2-d grid of processors. The

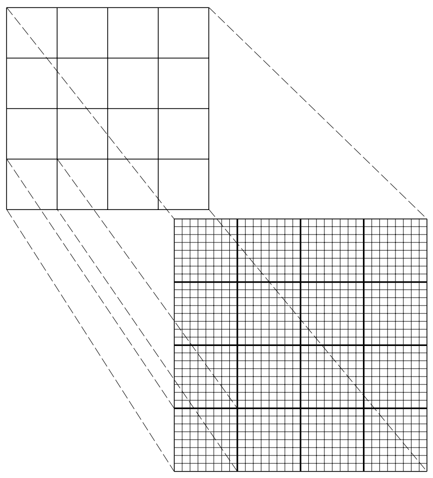


Figure 5.6: A 32 by 32 grid of relaxation data is mapped onto a 4 by 4 grid of processors.

obvious way to embed the problem is to map a contiguous 2-d portion of the relaxation array onto a single processor, such that the nearest neighbors of all of the points in a single processor are either on that processor or a neighbor of that processor. The embedding, shown in Figure 5.6, is reasonable for a hierarchy as well. For the model, we assume that the relaxation grid is mapped in the above fashion.

Note that an exact calculation of the read and write height can be performed for this application. Define R_l and W_l as the number of read and write operations which reach level l, respectively. These formulas are shown and derived in Equations B.1- B.3 in Appendix B. The read and write heights for the application can then be exactly expressed as:

$$h_r = \frac{\sum_{i=0}^{L-1} iR_i}{\sum_{i=0}^{L-1} R_i} \tag{5.8}$$

$$h_w = \frac{\sum_{i=0}^{L-1} iW_i}{\sum_{i=0}^{L-1} W_i} \tag{5.9}$$

5.2.3 Cluster

The cluster algorithm assumes that there are *clusters* or groups of processors working on data. Clusters are said to *own* blocks. The processors within a given cluster are more likely to reference blocks owned by the cluster than blocks owned by other clusters. This model

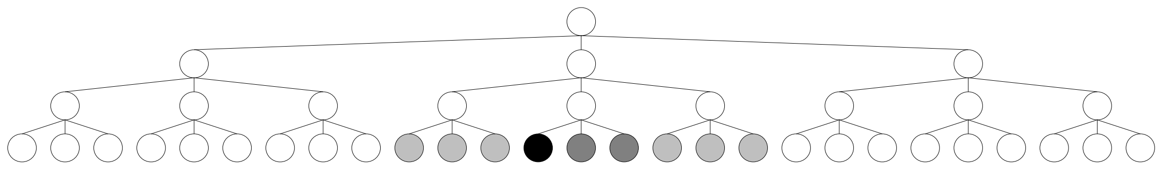


Figure 5.7: In this figure, the black node owns a block. The lighter the color of a node, the less likely it is to access the block.

is similar to the one proposed by Qing Yang, in [31].

Define e as the fraction of all operations by node P which occur to its own blocks. e is the defining parameter of a cluster application. As shown in Figure 5.7, P accesses blocks owned by processors in the group of b processors containing but not including P with uniform probability, E_1 which is calculated from e. P accesses blocks owned by processors in the group of b^2 processors containing but not including the aforementioned b processors with the smaller uniform probability, E_2. This access probability is calculated as follows:

$$E_l = \begin{cases} e & l = 0 \\ (1-e)\,\frac{2^{L-1-l}}{2^{L-1}-1} & l \in [1, L-1] \end{cases} \tag{5.10}$$

This formula implies that the frequency of requests to processors in the next largest cluster but not in the current one decreases by a factor of two as the clusters increase.

5.3 Simulation of Applications

The synthetic address traces of three applications were simulated using the simulator described in Chapter 4 in order to determine values for the set of locality parameters, with which to check the model. All applications were simulated both for a 2-dimensional, radix 4, four level tree (L=4, b=4) and for a 3-dimensional, radix 8, three level tree (L=3, b=8). In both cases this resulted in a 64 processor simulation. As much memory was allocated to the processors as was necessary to run without incurring cache overflow misses, in order to simulate infinite cache size.

Uniform	
n	200
τ	64
T	400

Relaxation	
n	2
T	100
w	1/5, 1/7

Cluster	
n	10000
τ	64
T	400

Table 5.1: These are the values of the parameters used in the simulations.

Uniform The uniform address trace consists of references, by every processor, every T simulator steps, to a randomly chosen one of τ addresses. All addresses are equally likely to be chosen by each processor. The parameter varied in this trace is the percentage of writes, w

Relaxation The relaxation address trace consists of cycles of reads to neighbors followed by writes. In the trace every node simultaneously makes the read requests followed by the write requests for the first block, then the read requests followed by a write request for the second block, etc. In other words, the grid is being updated such that some blocks are updated by values from later iterations on earlier iterations, similar to a Gauss-Seidel relaxation. There are T simulator cycles between each reference. The amount of the grid assigned to each node was varied across the simulations. n, the number of iterations, is low because the simulator performed a warm start for this application. For the 2-dimensional ($b = 4$) relaxation, the percentage of writes was 20; for the 3-dimensional ($b = 8$), the percentage was 14.

Cluster The cluster address trace consists of references, by every processor, every T simulator steps, to a randomly chosen one of N addresses, where N is the number of processors. Accesses by a particular processor to self-owned addresses occur with probability e. The probability of references to other clusters is calculated according to the formula described earlier in Equation 5.10. The parameters varied in this trace are the percentage of writes, w and the base probability e.

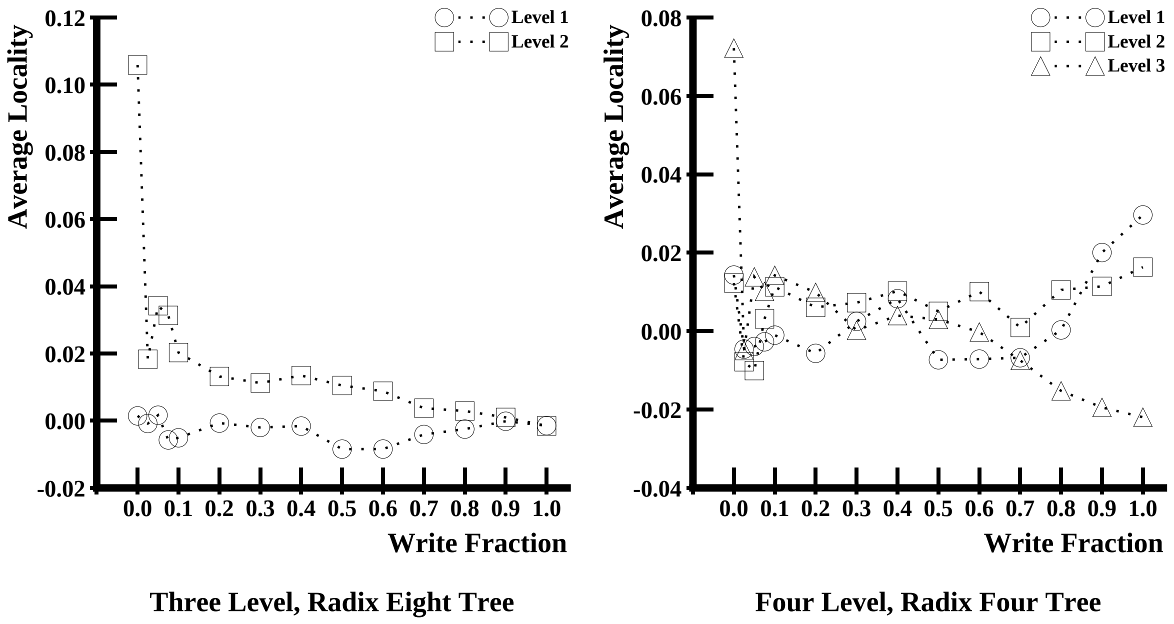


Figure 5.8: The locality parameters, p_l, as measured from the uniform application simulation.

5.4 Locality Parameters Measured from Simul

The simulation allowed us to measure p_l, the set of locality parameters for the applications.

Uniform As predicted, the values for p_l are nearly zero for the uniform application, as shown in Figure 5.8. A high point occurs when the percentage of writes is zero; this behavior is caused by the fact that there are no writes at all. If the simulation was run for a very long time, eventually nearly every node would have a copy, and then there would seem to be correlation in the choice of a subtree; a valid subtree node would be more likely to be selected than an invalid one because there are so many.

Relaxation As the amount of data per node increases, the application exhibits more and more locality at every level, as expected. Note that there is tremendous locality for the references which reach the higher levels, as there are extremely few of them

Cluster For all values of e, the fraction of references by an owner to its own blocks, and l, the number of levels, as the write fraction increases the value of p_l decreases. This is because a write issued from a more remote node causes p_l to decrease twice: once when the

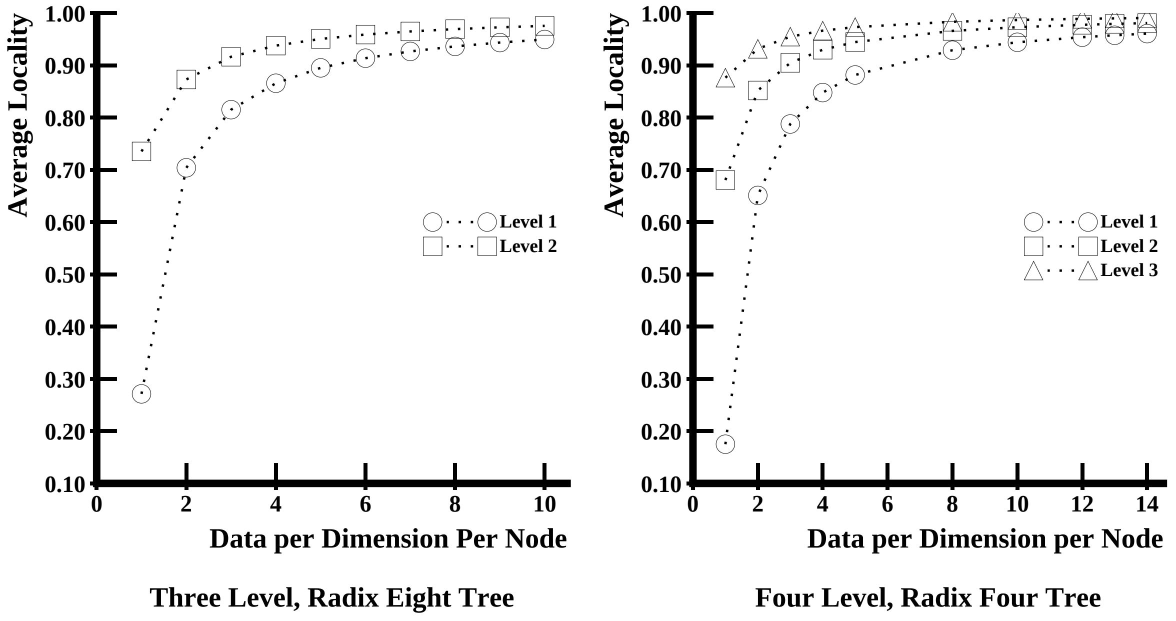


Figure 5.9: The locality parameters l_i as measured from the relaxation application simulation.

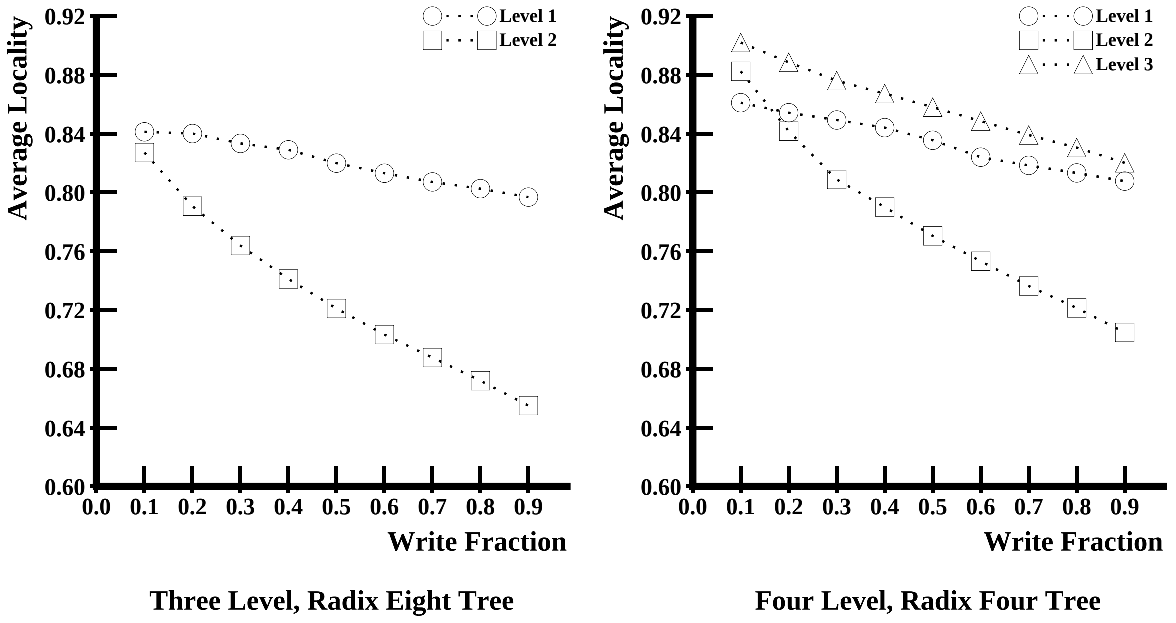


Figure 5.10: The locality parameters l_i as measured from the cluster application simulation. The base reference fraction $e = 0.75$.

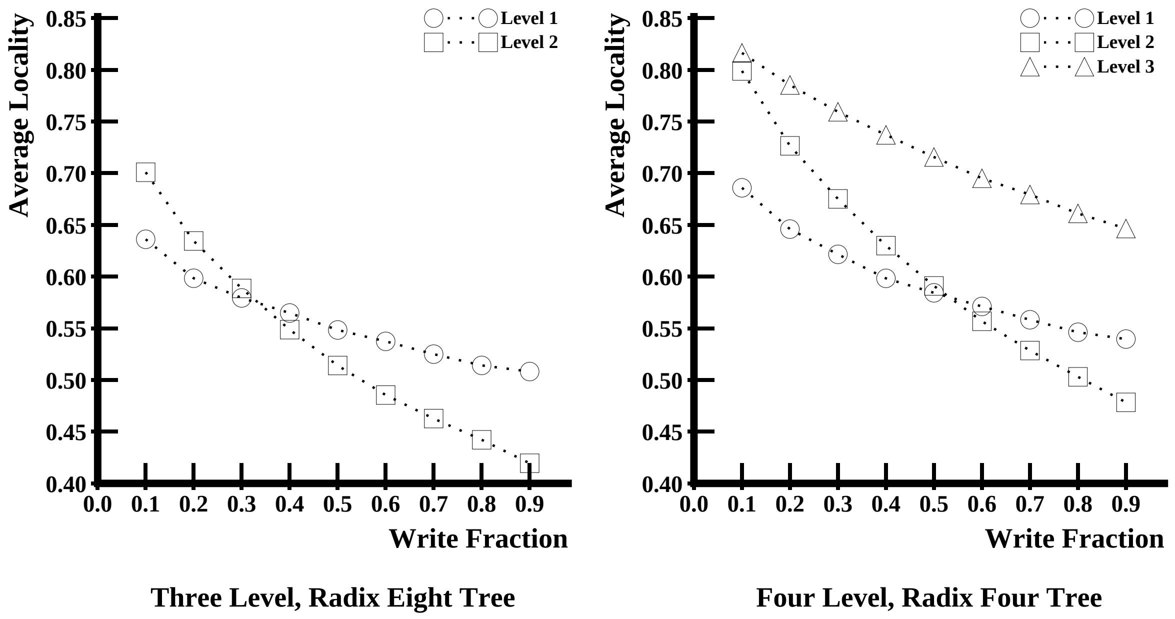


Figure 5.11: The locality parameters, p_i, as measured from the cluster application simulation. The base reference fraction $e = 0.5$.

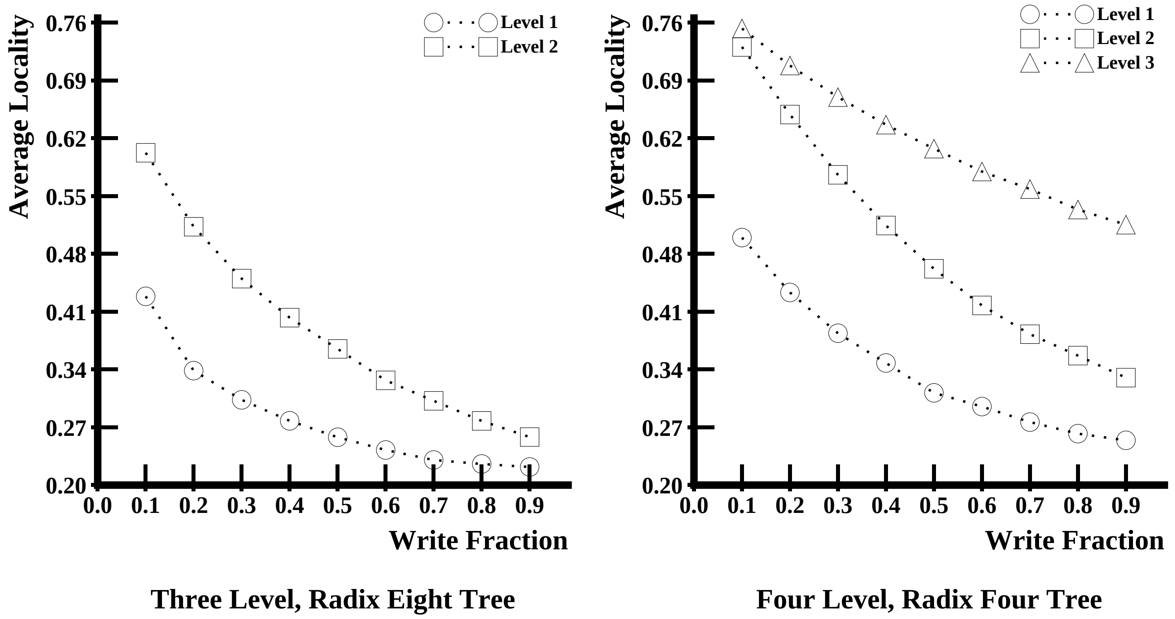


Figure 5.12: The locality parameters, p_i, as measured from the cluster application simulation. The base reference fraction $e = 0.25$.

write is issued, to pull the value into the cache, and once when the more common read or write occurs to the owner node.

Note that as e increases, p_1 stays artificially high. This behavior is caused by the extremely high value of e. When e is 0.75, three-quarters of all requests to blocks owned by node P are made by node P. This means that the effect mentioned above, where p_1 decreases with write fraction, does not often occur (since a write to the owner node followed by another operation to the owner node (with no intervening writes by other nodes)) does not lower the locality parameter.

5.5 Comparison of Model and Simulation

The predicted and simulated average read and write heights are very similar, and confirm that the set of locality parameters is a valid way of expressing the behavior of an application. The predicted number of messages per read and write operation also compares favorably to the simulation.

Uniform We examined the average read and write height of the uniform application as the write fraction is varied, as shown in Figure 5.13. Note that as expected, using the value of the set of locality parameters measured from the simulation produced identical results as just using the value zero, showing that the deviations from zero in the p_i curve are insignificant.

Figure 5.14 compares the predicted number of messages per operation with the simulated number. Although the predicted number of write messages is higher than the simulated for low values of the write fraction, the predicted and simulated number nearly match for the rest of the write fraction range, and the shape of the curve is very similar.

Relaxation For the relaxation application, we examined the average height characteris as a function of the amount of data allocated to each node. The resulting 3-dimensional (radix 8) and 2-dimensional (radix 4) graphs can be seen in Figure 5.15. Note that the numbers shown on the x-axis of the graphs represent the amount of data allocated per dimension per node. In other words, to calculate the actual data per node, cube the number

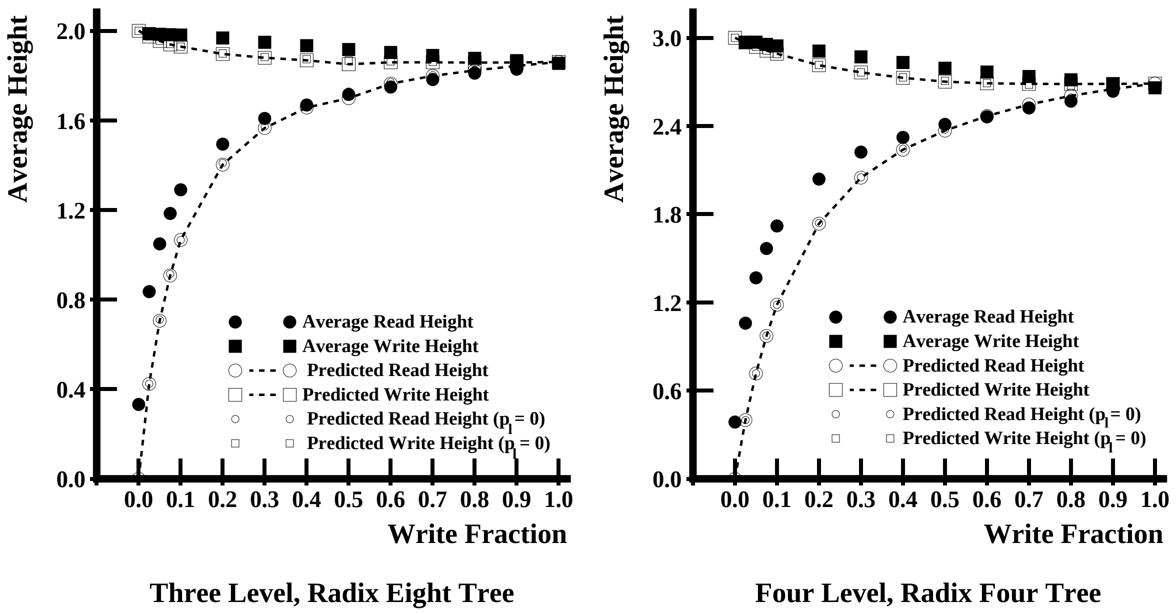


Figure 5.13: This figure shows the average read and write height model predictions as well as the simulated ones for the uniform application. Note that two sets of values for p_l were used: one where p_l was set to zero, and one where p_l was measured from the simulation.

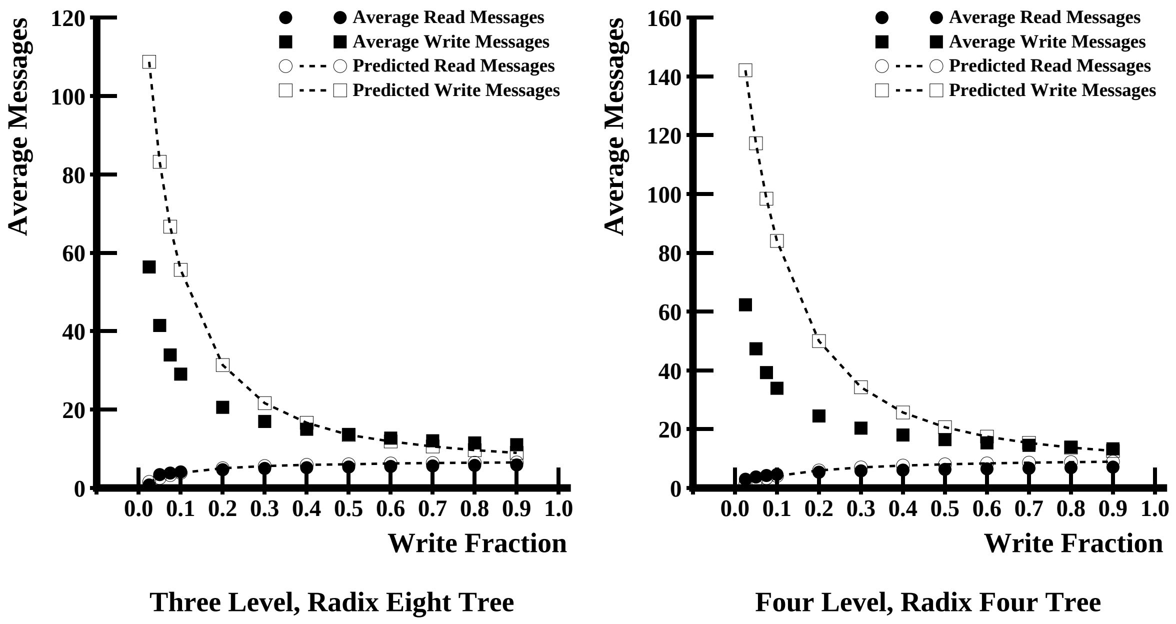


Figure 5.14: This figure compares the predicted number of messages per read and write operation with the simulated values for the uniform application. p_l was measured from the simulation for the predicted curve.

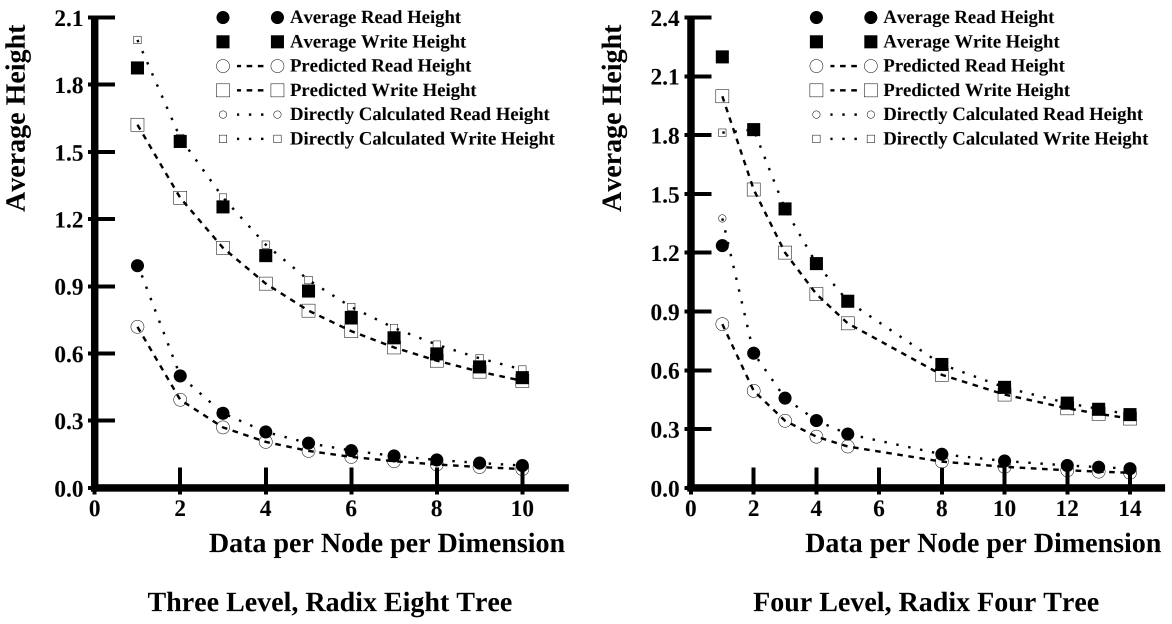


Figure 5.15: This figure shows the average read and write height model predictions as well as the simulated ones for the relaxation application. Two different predictions are shown: one uses the locality parameter model, and one uses an analytical calculation of the relaxation data motion to directly calculate the heights.

shown in the figure for the 3-dimensional case, and square the number for the 2-dimensional case. The directly calculated average heights mentioned in Section 5.2.2 are also included on the graphs. The exact calculation does not work properly when there is only one data point allocated per processor.

Cluster Figure 5.16 shows graphs of the write fraction versus the average read and write characteristics for three values of the base reference parameter: 0.75, 0.5, and 0.25. The model is more accurate for higher base reference values.

Figure 5.17 compares the number of messages per read and write operation predicted from the model with the number of messages recorded as sent by the simulation. A base reference rate of 0.75 and 0.25 is shown in the figure. Note that in these graphs, the predicted number of messages per read and write seems over-predicted for higher values of the base reference rate. In fact, the number of simulated messages is low. This effect is caused by the method of gathering message statistics in the simulation: messages which are sent from a physical processor to itself, even if the virtual nodes being represented change,

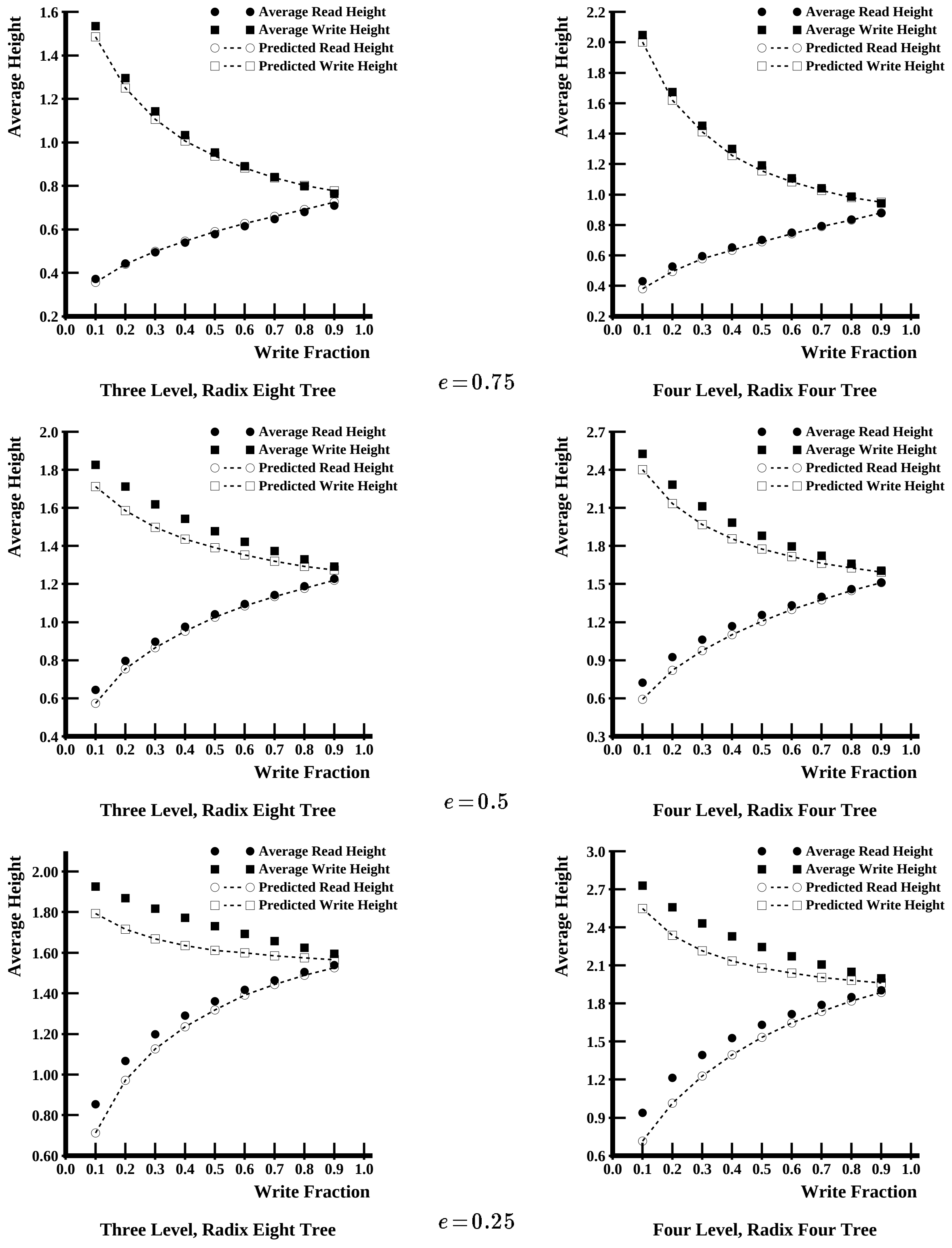
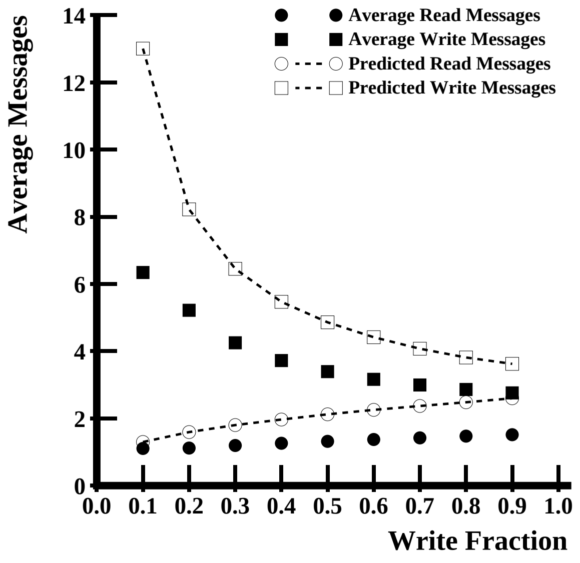
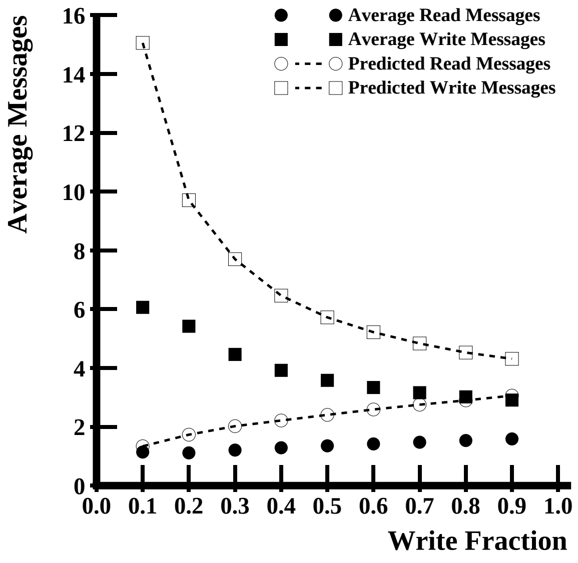


Figure 5.16: This figure shows the average read and write height model predictions as well as the simulated ones for the cluster application.

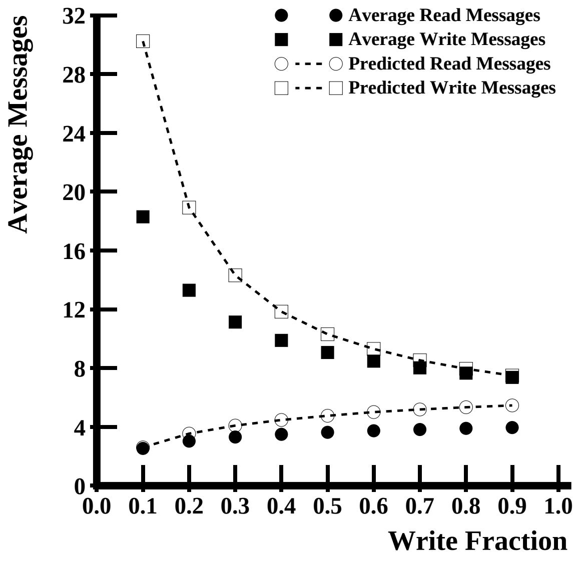


Three Level, Radix Eight Tree

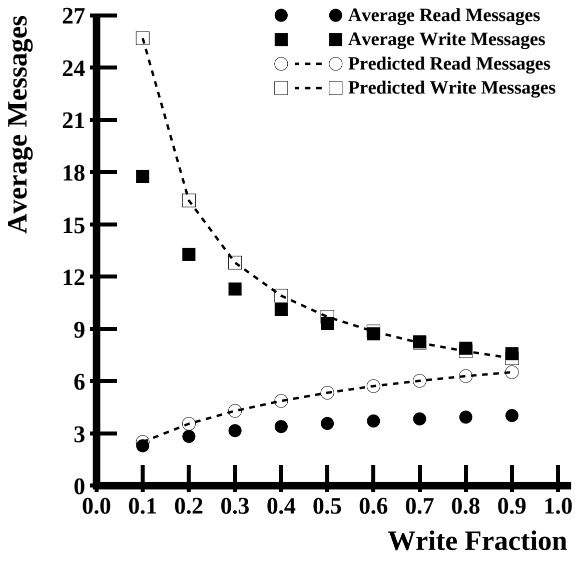


Four Level, Radix Four Tree

$e = 0.75$



Three Level, Radix Eight Tree



Four Level, Radix Four Tree

$e = 0.25$

Figure 5.17: This figure compares the simulated and the predicted number of messages per read and write operation for the cluster application.

Model Parameters	
w	0.3
b	8
L	3, 4, 5, 6
N	64, 512, 4096, 32768

Model Parameters	
w	0.3
b	4
L	4, 5, 6, 7, 8, 9
N	64, 256, 1024, 4096, 16384, 65536

Table 5.2: These are the values of the input parameters for the model.

are not counted. Because of the way the mapping of virtual nodes to physical processors is performed (see Section 2.2 for details), many more messages are sent from a processor to itself when the base reference rate is high.

5.6 Protocol Characterization for Large Machine Sizes

In this section, the verified model is used to predict the behavior of the protocol on machine sizes too large to simulate.

5.6.1 Parameters

In order to simplify the study, several of the model input parameters have been constrained, as shown in Table 5.2. The fraction of writes is fixed at 0.3: a reasonable choice for parallel applications [28] as well as one at which the messages per operation calculation is accurate. Trees with two different radices, eight and four, are modeled. The range of machine sizes is chosen to show the trend of the curves.

The set of locality parameters is fixed to a single value for all levels, rather than a set of values for each level. This fixing still provides interesting results because $\forall l : p_l = 0$ is a uniform reference input stream, and $\forall l : p_l = 1$ is a completely local input stream. Most applications will lie between these two extremes. Furthermore, the values of p_l for different l seen in the uniform and relaxation application were very close, and the values in most of the cluster applications were similar.

5.6.2 Average Height

Figure 5.18 shows the average height per request as a function of the machine size and the locality. The "Height/Operation" characteristic is determined by weighting the "Height/Read" and the "Height/Write" values by the write fraction. Results for both radix eight and radix four trees are shown. Note that the "Machine Size" axis is plotted on a logarithmic scale; alternately, the scale can be viewed as linear, relabeling that axis a "Number of Levels" with the values 3–6 for radix eight, and 4–9 for radix four.

There are two trends to observe. First note the importance of locality, especially with larger machine sizes. The second effect is that of machine size. The average heights all grow sub-linearly with machine size, and nearly linear with the number of levels. Note, however, that with a large machine size, and low locality, nearly the entire tree is being traversed during an average request. This behavior is clearly unacceptable.

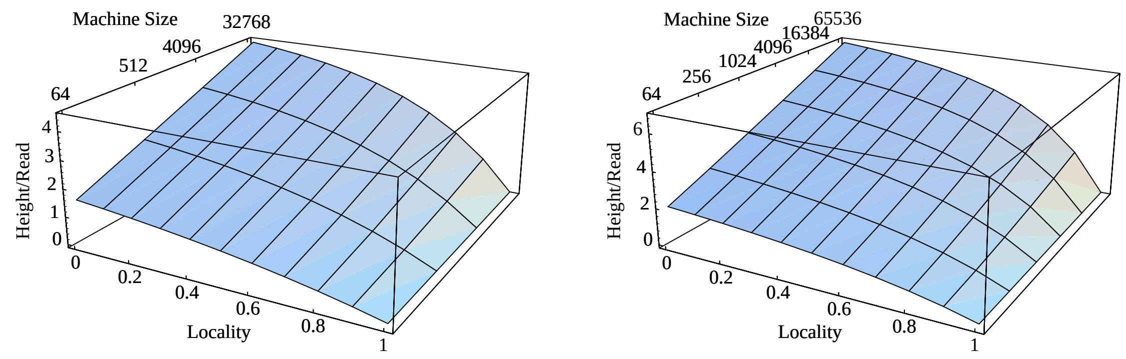
5.6.3 Longest Path

Figure 5.19 shows the effect on the longest path per request as a function of the machine size and the locality. The form of these results is very similar to that of the average height. The main point to note about these graphs is the sheer number of nodes each request will, on average, have to pass through. Even if the network bandwidth were large enough to support this many requests, the nodes need to examine each message passing through, and would likely have long queues of pending messages to examine.

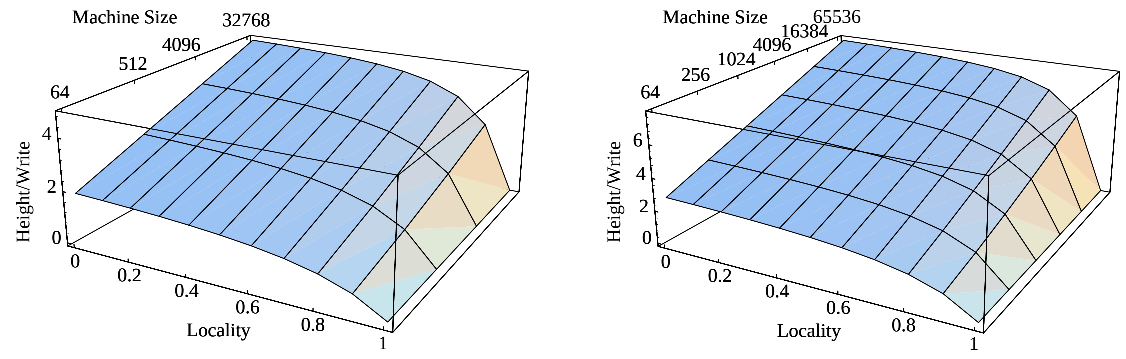
5.6.4 Number of Messages

The number of messages sent per request as a function of the machine size and the locality is shown in Figure 5.20. The shape of the curve of number of messages sent per read is similar to those discussed earlier. The curves for the number of messages sent per *write* and the number of messages sent per *operation* (the weighted combination of reads and writes), on the other hand, are different.

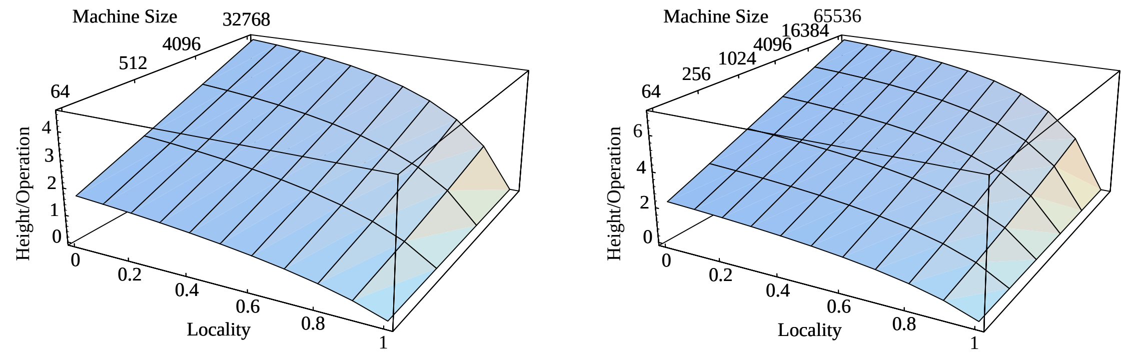
Because the number of messages sent per write depends not only on the *distribution* of the nodes with copies of the block, but also on the *number* of nodes with copies of the block, the effects of machine size and locality are much more pronounced. Instead of varying



Average Height/Read

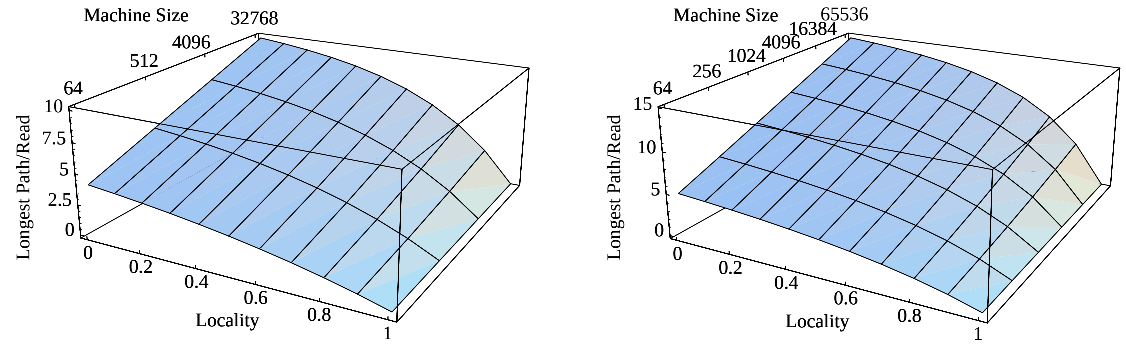


Average Height/Write

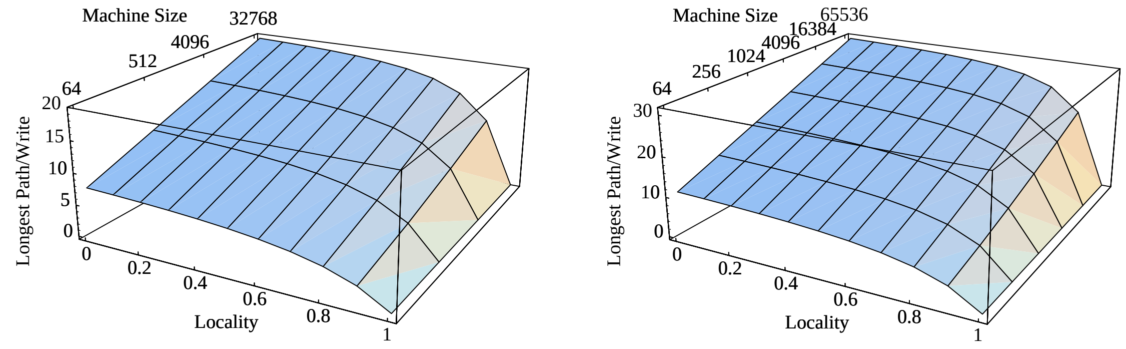


Average Height/Operation

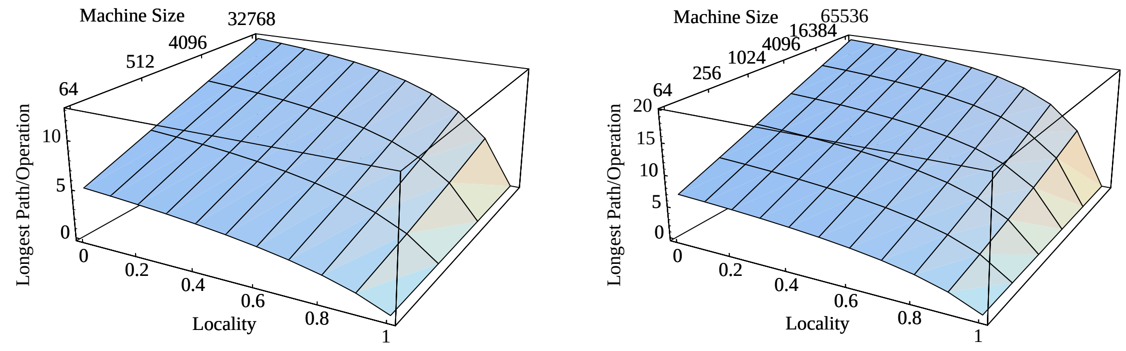
Figure 5.18: This figure shows the predictions for average height per operation as a function of machine size and of locality. The left graphs are for radix eight trees; the right graphs are for radix four.



Longest Path/Read

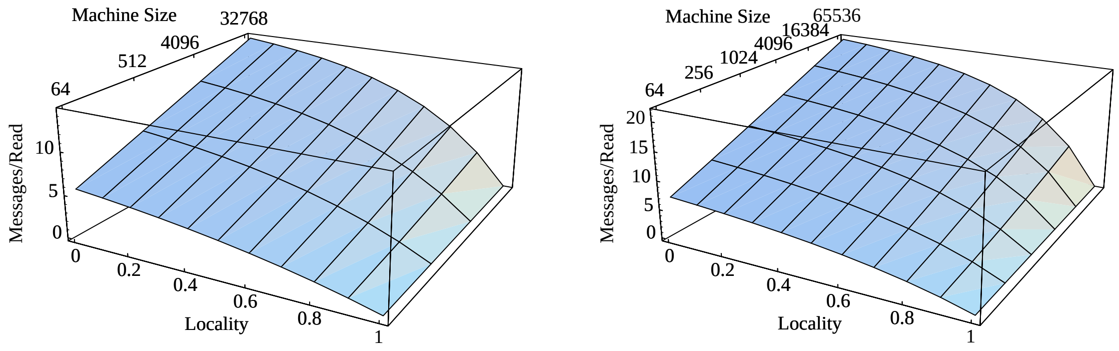


Longest Path/Write

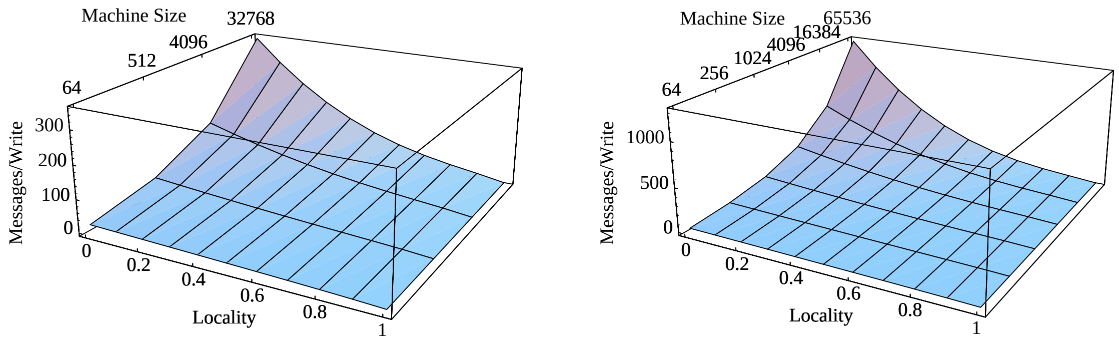


Longest Path/Operation

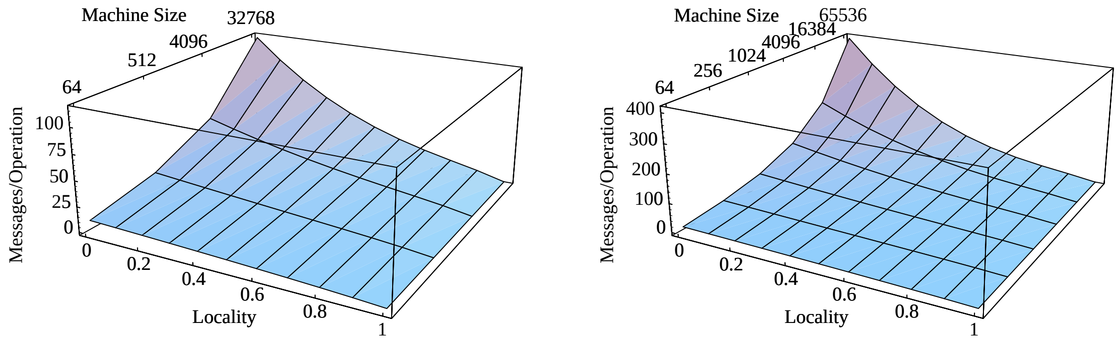
Figure 5.19: This figure shows the predictions for the length of the longest path traveled per request as a function of machine size and of locality. The left graphs are for radix eight trees; the right graphs are for radix four.



Messages/Read



Messages/Write



Messages/Operation

Figure 5.20: This figure shows the predictions for the number of messages sent per request as a function of machine size and of locality. The left graphs are for radix eight trees; the right graphs are for radix four.

logarithmically, where the main gains are for locality in the 0.75 to 1 range, the number of messages per write versus locality curve is barely affected by small changes for high locality; as the locality decreases, the number of messages sent per write increases polynomially. The degree of the polynomial varies with machine size, implying that while applications with poor locality may perform reasonably on small machines, they will swamp large machines.

The number of messages sent per write and per operation as a function of machine size is actually sub-linear. As a function of the number of levels, however, the number of messages sent is definitely quadratic.

5.7 Issues

Although the results described in this chapter are of interest in examining the performance of the protocol, there are many extensions that could be done to provide more insight into the abstract protocol behavior. Many applications should be analyzed to determine the precise meaning of the set of locality parameters. A better estimate of the locality parameter could be used. Finally, the data in an application could be divided into sets, and the locality parameters separately calculated for each one.

Currently, the locality parameter set can only be derived for an application by measuring the parameters from simulation. We have performed some initial work towards deriving the set of locality parameters from a spatial locality model of an application, such as that available for the cluster application. The derivation works best, however, for applications which exhibit a very high degree of clustering. More work needs to be done in this area.

Using a flat set of locality parameters is not necessarily realistic. For large applicatio running on massively parallel machines, we might expect less sharing to occur near the top of the hierarchy, and more at the bottom Studies need to be done of applications to provide insight as to what the hierarchical locality of applications actually is like.

For applications which have a large variance in the types of data referencing, several sets of locality parameters can be used to avoid averaging effects. This would allow one to separate widely shared data such as synchronization variables from less used ones. This separation is useful because an application may stall due to the high sharing of synchronization variables. This method might also provide new insight into the interactions between shared data and program execution-time behavior.

5.8 Summary

In this chapter we have proposed a method of expressing locality in applications mapped onto hierarchical architectures. We have used this model to predict the average height per request, the average longest path per request, and the average number of messages sent per request. We used three applications in order to validate the model: a uniform reference stream, a relaxation algorithm, and a clustering data-reference stream.

After validating the model, we employed it in the prediction of the abstract performance of very large machines as a function of the locality, studying how the model outputs varied with machine size and locality. The most important result is that locality is extremely important in an application. As machine sizes grow, the locality becomes increasingly important for reducing latency.

We will use the abstract model as input to an embedded model in Chapter 6. The embedded model describes how the protocol runs when mapped onto particular machines. This will allow us to study how the protocol behaves under conditions where requests are not allowed to send an unlimited number of messages without penalty.

Chapter 6

Embedded Analysis

This chapter extends the abstract analysis of Chapter 5 to show how embedding PHD into a machine affects the behavior of the protocol. We use the mapping described in Section 2.2 to embed the protocol into a k-ary n-cube. The embedded model describes this mapping, as well as the configuration of the architectures being studied.

In our study we find that multithreading is only useful for approximately two to four threads; interleaving more than that does not decrease the overall latency. For small machines and high locality applications, this limitation is due mainly to the length of the running threads. For large machines with medium to low locality, this limitation is due mainly to the large protocol overhead.

We also consider the addition of controllers to the processing nodes. We will see that the gains from the addition of these controllers are not large enough to justify hardware which is more expensive than processors. In no case does the addition of the controllers save more time than double the number of processors.

We first provide a brief description of the embedded model in Section 6.1 and derive the necessary inputs. We then characterize the behavior of the mapped protocol in Section 6.2 for several different architectures. Finally, we discuss what further issues need to be studie in Section 6.3. An alphabetical listing of all of the variables defined in this thesis can b found in Table A.1.

6.1 An Embedded Model

The embedded model is basically the model derived by Johnson in [13], slightly modified to suit our purposes. The abstract model is used to generate the inputs to the embedded model.

The embedded model is used to study two main architectural configurations: a machine in which protocol activities are handled by the same processor which is attempting to do work, as outlined in Section 2.2, and a machine in which protocol activities are handled by a separate controller.

6.1.1 Model Overview

Johnson developed a framework for modeling how communication affects performance. His framework consists of three parts: a network model, an application model, and a transaction model. These three are combined into a single model in order to provide feedback between each subsystem nodes will be unable to inject messages into the network faster than the transaction latencies will allow. The model is fully described in [13]; only the parts of th model which have been changed for this analysis will be discussed in detail.

The embedded model directly uses the application and the network models. An application consists of threads running on processors. The threads run until they make off-node requests (communication transactions). In the absence of multithreading, the threads suspend until their transactions finish. If there is multithreading, and there are still runnabl threads, a context switch occurs, and a new thread is started.

In Johnson's model the application model invokes communication transactions; in our embedded model it invokes off-node requests to shared memory. The off-node requests are modeled identically to the transaction model, except that the meaning of one of the parameters is different; t_f, the fixed delay of Johnson's model, represents the time necessary to process protocol requests by a non-leaf node. As such, it becomes a function of how many messages are sent.

The network model is used to determine average message latency, given an input message size, injection rate, and communication distance. The network is assumed to be a k-ary n-dimensional mesh, with separate unidirectional channels in both directions.

R	Average thread run length between successive requests to shared memory.
T_s	Context switch time.
M_l	Average time to satisfy a locally satisfiable request to shared memory.
M_r	Average time to process a protocol message invoked on a processor.
N_i	Average network interface overhead
C	Number of words in a cache line.
f	Number of flits per word.
p_m	Degree of hardware multithreading.

Table 6.1: The additional basic input parameters needed for the embedded model.

c	Average number of messages in critical path of a non-local shared-memory request.
g	Average number of messages per non-local shared-memory request.
B	Average message size (in flits).
d	Average distance a message travels (in hops).
k_d	Average distance a message travels in each dimension.
T_r	Average thread run length between successive non-locally satisfiable requests.
T_f	Non-network overhead to satisfying a non-local shared-memory request.

Table 6.2: The derived input parameters needed for the embedded model.

6.1.2 Model Inputs

The embedded model takes as input many parameters. Some of these parameters have been discussed in the abstract analysis chapter, and vary depending on the application. Other parameters, shown in Table 6.1, need to be specified only when the protocol is mapped to an architecture. The embedded model uses a third set of parameters, derived from the first two sets, as its actual inputs. This third set is listed in Table 6.2, and will be derived i this section.

Locally Satisfiable Shared-Memory Requests We first calculate the number of requests to shared memory that are locally satisfiable, to use in later equations. We determine the expected fraction of locally satisfiable reads by calculating the probability that a reques will come from a node in the class of nodes which are valid.

$$E[Z_r] = \prod_{l=1}^{L} t_l \tag{6.1}$$

We find the expected fraction of locally satisfiable writes by calculating the probability that a request will come from a node in the class of nodes which are valid, and all of whose ancestors are the only valid nodes in their set of siblings.

$$E[Z_w] = \prod_{l=1}^{L} t_l v_l^1 \tag{6.2}$$

The expected fraction of locally satisfiable shared-memory requests is just the weighted fraction of Z_r and Z_w.

$$E[Z] = rZ_r + wZ_w \tag{6.3}$$

Number of Messages in Critical Path The calculation of the expected number of messages in the critical path of a non-locally satisfiable shared-memory request, c, is similar to the calculation of the longest path for an operation (d_r and d_w in Equations 5.4 and 5.5). There are, however, two differences. First, we condition the calculation on non-local operations by dividing by the fraction of non-local requests. Second, we only want to consider those messages which actually need to be sent off-node. In the embedding, a parent node and its b children for a particular block correspond to b nodes. This means that one of the children is situated on the same physical processor as its parent. Equation 6.4 gives the expected number of messages for a read request.

$$E[c_r] = \frac{1}{1-Z_r} \sum_{h=1}^{L-1} \left(\frac{b-1}{b} 2h + 1 \right) (1 - t_h) \prod_{l=h+1}^{L} t_l \tag{6.4}$$

The critical path for a write request contains a fan-in and fan-out to all nodes with copies of the block. We make the reasonable assumption that at least one of these paths will contain no message sends between nodes mapped to the same processor, so the expected number of messages in the critical path for a write operation is just the expected number in the longest path for a write, conditioned on the non-local factor, as shown in Equation 6.5.

$$E[c_w] = \frac{1}{1 - Z_w} E[l_w] \qquad (6.5)$$

The expected number of messages in the critical path of a general request is calculated by weighting c_r and c_w by the write fraction.

$$E[c] = rc_r + wc_w \qquad (6.6)$$

Number of Messages The calculation of the expected number of messages sent for non-locally satisfiable shared-memory requests, g, is similar to that of the number of messages sent in the abstract model (m_r and m_w in Equations 5.6 and 5.7). There are two differences between these calculations, the same as described for Equations 6.4 and 6.5 above. The expected number of messages for a read operation is given in Equation 6.7.

$$E[g_r] = \frac{1}{1 - Z_r} \sum_{h=1}^{L-1} \left(\frac{b-1}{b} 3h+1 \right) (1 - t_h) \prod_{l=h+1}^{L} t_l \qquad (6.7)$$

A write operation generates many messages, some of which are expected to stay local to a physical processor. Since we are not counting the critical path length, just the total messages sent here, we do weight by the branching factor, as shown in Equation 6.8.

$$E[g_w] = \frac{1}{1 - Z_w} \sum_{h=1}^{L-1} \left(\left(\frac{b-1}{b} 2h+1 + \frac{b-1}{b} 2 \sum_{l=1}^{h} \prod_{e=l-1}^{h-1} c_e \right) \left(1 - v_h^1 t_h\right) \prod_{l=h+1}^{L} t_l v_l^1 \right) \qquad (6.8)$$

The expected number of messages sent by a general request is calculated by weighting g_r and g_w by the write fraction.

$$E[g] = rg_r + wg_w \qquad (6.9)$$

Flits per Message The expected number of flits sent per message is dependent on the exact machine to which the protocol is mapped. Define f as the number of flits per word, and C as the cache line size in words. We assume 32 bit words for the purpose of this calculation. A read sends the 4 word *find_lowest_common_for_read* message, the 3 word *readm* message, the $3+C$ word *read_data* message, and the $4+C$ word *confirm_value* message. These messages are described in Table 2.4.

$$E[B_r] = f\frac{1}{1-Z_r}\sum_{h=1}^{L-1}\frac{(11+C)h+(3+C)}{3h+1}(1-t_h)\prod_{l=h+1}^{L}t_l \qquad (6.10)$$

A write operation sends the 4 word *find_lowest_common_for_write* message, the 3 word *lock* message, the 4 word *ack* and *ack1* messages, the $2+C$ word *s_write_own* message and the 3 word *write_ok* message.

$$E[B_w] = f\frac{1}{1-Z_w}\sum_{h=1}^{L-1}\left(\frac{(7)h+(2+C)+(7)\sum_{l=1}^{h}\prod_{e=l-1}^{h-1}c_e}{2h+1+2\sum_{l=1}^{h}\prod_{e=l-1}^{h-1}c_e}\left(1-v_h^1 t_h\right)\prod_{l=h+1}^{L}t_l v_l^1\right) \qquad (6.11)$$

The expected number of flits per general message sent, B, is weighted by the number of messages sent by each type of operation as well as the frequency of each operation.

$$E[B] = \frac{r g_r B_r + w g_w B_w}{r g_r + w g_w} \qquad (6.12)$$

Distance per Message The expected distance in hops that the average message travels, d, depends on the radix k and the number of dimensions n of the machine to which the protocol is mapped. Note that the embedding is such that $\log_2 b$, where b is the branching factor of the tree, is equal to n.

We first determine d_l, the expected number of hops needed for a message sent between a node at level l and its parent, where the parent and the node are located on separate processors. This is just the number sent for a 2-ary n-cube [2] [22], scaled by the dilation caused by higher levels.

$$E[d_l] = \frac{b\log_2 b}{2(b-1)}\cdot 2^{l} = \frac{n2^{n-1}}{2^n-1}\cdot 2^{l} \qquad (6.13)$$

We also need to determine d_l', the number of hops that a hierarchy-circumventing message, such as *read_data*, will take. A message of this form is sent from a node directly to another node. The two nodes are guaranteed not to share a common sub-cube smaller than that of their lowest common ancestor. The values of d_l' were counted, and are enumerated

Radix	Level							
	1	2	3	4	5	6	7	8
4	4/3	3	37/6	149/12	199/8	2389/48	9557/96	12743/64
8	12/7	57/14	237/28	957/56	3837/112			

Table 6.3: This table shows t_l the empirically determined number of hops a message sent directly from a node in a subtree to one outside of its subtree takes, if the lowest common ancestor of those two nodes is at level l. This calculation assumes a uniform distribution.

in Table 6.3.

The expected distance per read message is given in Equation 6.14. Note that the number of hops sent between each level is summed and then multiplied by the number of tree traversals. We do not worry about excluding the messages which are not actually sent; the difference is negligible.

$$E[d_r] = \frac{1}{1-Z_r} \sum_{h=1}^{L-1} \frac{(3\sum_{l=1}^{h} d_{l-1}) + d'_h}{3h+1} (1-t_h) \prod_{l=h+1}^{L} t_l \tag{6.14}$$

The expected distance per write message is similarly calculated in Equation 6.15.

$$E[d_w] = \frac{1}{1-Z_w} \sum_{h=1}^{L-1} \frac{2(\sum_{l=1}^{h} d_{l-1}) + d'_h + 2\sum_{l=1}^{h} d_{l-1} \prod_{e=l-1}^{h-1} c_e}{2h+1+2\sum_{l=1}^{h}\prod_{e=l-1}^{h-1} c_e} \left(1 - v_h^1 t_h\right) \prod_{l=h+1}^{L} t_l v_l^1 \tag{6.15}$$

The expected distance per general message type sent, d, is weighted by the number of messages send by each type of operation as well as the frequency of each operation.

$$E[d] = \frac{r g_r d_r + w g_w d_w}{r g_r + w g_w} \tag{6.16}$$

Distance per Dimension The average distance a message travels in each direction, k_d, assuming independence, is just d/n. In Johnson's model, when k_d is less than one, the average per-hop latency for the head of a message is fixed to one.

Run length Between Requests The average thread run length between successive requests to shared memory, R, is one model input. Another, M_z, is the average time needed to satisfy a locally satisfiable shared-memory request, *i.e.* one that does not cause any off-node traffic. The average thread run length between successive requests to non-locally satisfiable shared memory, T_r, is a function of these two parameters. For the purpose of this model, T_r is considered to be the useful work done by the processor.

$$E[T_r] = R + \frac{Z}{1-Z}(R + M_z) \tag{6.17}$$

Non-network Overhead The mapping of non-leaf nodes onto the same processors as leaf nodes guarantees that all processors will need to spend some time processing protocol transitions, instead of performing actual work. We model this non-network overhead, T_f, as a function of g, the average number of messages sent per operation, M_r, the average time in processor cycles needed to handle a protocol request, b, the branching factor, and N_i, the time taken up by the network interface.

If every node is sending g messages on average per request, every node will have to process g messages. The cost of processing a message is M_r if it stays on the current processor, and $M_r + N_i$ if it comes in from another processor.

$$E[T_f] = gM_r + \frac{b-1}{b} gN_i \tag{6.18}$$

6.1.3 Model Constraints

The model developed by Johnson is only valid under certain conditions, where the available parallelism is smaller than the communication transaction latency, so that despite multi-threading, the processors will be idle part of the time. In the embedded model, the extra time where threads would normally be waiting can be used to support protocol transactions.

In an architecture with no controller, where every processor both runs threads and supports protocol transactions, the analysis is only valid as long as the thread run length, the context switch time, and the protocol transaction overhead time do not exceed the transaction latency time, as shown in Equation 6.19.

$$T_t \geq p_m(T_s + T_f) + (p_m - 1)T_r \tag{6.19}$$

When this condition is false, the average inter-transaction issue time, t_t, is limited as follows:

$$t_t = T_r + T_s + T_f \tag{6.20}$$

When the protocol requests are handled by a controller, there are two constraints that must be met. First, the transaction latency time must be greater than the thread run length and the context switch time:

$$T_t \geq p_m(T_s) + (p_m - 1)T_r \tag{6.21}$$

When this condition is false, the average inter-transaction issue time, t_t, is limited to $T_r + T_s$. The transaction latency time plus the thread run length must also be less than the time required by the controller to process the protocol requests:

$$T_t \geq p_m T_f - T_r \tag{6.22}$$

When this condition is false, the protocol is limited by the speed of the controller to a time of no less than:

$$t_t = T_f \tag{6.23}$$

When both of those conditions are false, the protocol is limited by the largest of the above transaction issue times.

6.2 Protocol Characterization

There are several fundamental questions we must address. The first is the smallest average inter-transaction issue time that can be sustained. This time depends on the degree

Shared Model Parameters	
T_s	20 cycles
M_l	10 cycles
N_i	15 cycles
C	8 words
f	2 flits
p_m	1, 2, 4
p	0-0.9
w	0.3
b	8
L	3, 4, 5, 6
N	64, 512, 4096, 32768

Table 6.4: These are the values of the input parameters which are shared for the different architectures.

of multithreading: with more threads running, more latency can be hidden. Multithreading is only useful up to a certain point, however. Another important question is what sort of overhead is seen, and what are its sources? Are the limits set by protocol overhead or by network latency? This section first describes the parameters chosen for the study, then shows the results of the study.

6.2.1 Parameters

We chose to study three machine configurations, representing a variety of architectures. Across all three configurations particular parameters, listed in Table 6.4, were held constant, since we were most interested in varying the other parameters.

Figure 6.1 shows the number of flits per message, and the distance traveled per message, as a function of machine size and locality. These parameters are used as input to the embedded model. The number of flits per message is higher for high locality; this effect is caused by the domination of longer messages, such as *read_data*. The more nodes which need to be invalidated, the lower the number of flits will be. The effect of the distance being so comparatively large for high-locality large machines is caused because the distance grows exponentially as one ascends the hierarchy, yet all levels of the hierarchy are assigned an

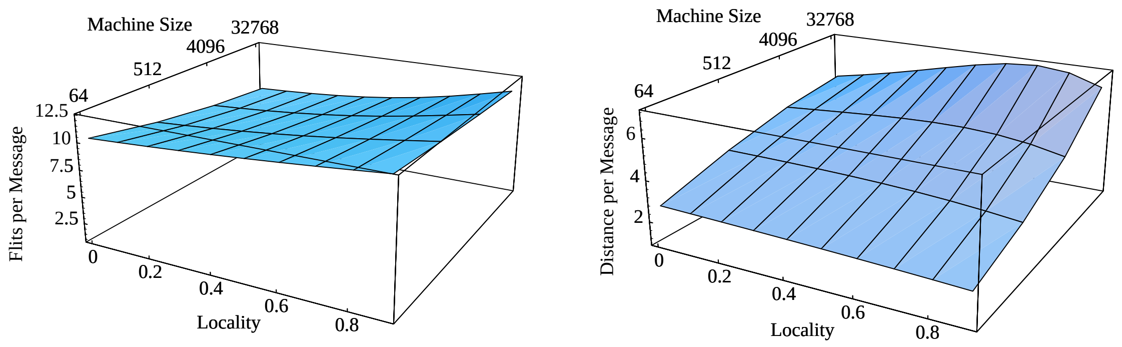


Figure 6.1: This figure shows the predictions for the average number of flits needed per message, and the average distance that a message travels, as functions of machine size and of locality.

Model Parameters	
Optimistic-Optimistic	
R	500
M_r	20

Model Parameters	
Optimistic-Pessimistic	
R	500
M_r	100

Model Parameters	
Pessimistic-Optimistic	
R	50
M_r	20

Table 6.5: These are the values of the input parameters which are varied across the different architectures.

---

equal locality parameter. This implies that further studies with a set of graduated locality parameters rather than flat ones might be interesting. The number of critical messages in an operation, c, and the total number of messages sent per operation, g, are similar to l and m graphed in Figures 5.19 and 5.20 and are therefore not shown here.

Table 6.5 lists the values of the parameters which were varied across the architectures. These values correspond to three situations:

1. Optimistic-Optimistic: The run length between shared-memory references is long (such as on the J-Machine, where floating point operations are implemented in software), and the protocol overhead is low.

2. Optimistic-Pessimistic: The run length between references is high, and the protocol overhead is high (for example if the protocol was implemented entirely in software).

3. Pessimistic-Optimistic: The run length between messages is very short, and the pro-

tocol overhead is low.

We additionally consider the case where the non-leaf protocol overhead is handled by a separate controller for all three architectures.

Figure 6.2 shows the average run length between requests to off-node shared memory, T_r and the non-network-related protocol processing overhead, T_f. This T_r run length is an input to the embedded model, and was calculated from average run length between requests to shared memory (R_l), in Equation 6.17. Note that T_r varies from being almost exactly R_l for machines with no locality, to approximately $4R_l$ for small machines with a locality of 0.9, and $2R_l$ for large machines with a 0.9 locality. The overhead, shown in Equation 6.18, is another input to the model, and is mainly affected by the number of messages sent to satisfy an operation.

6.2.2 Architectures Without A Separate Cache Controller

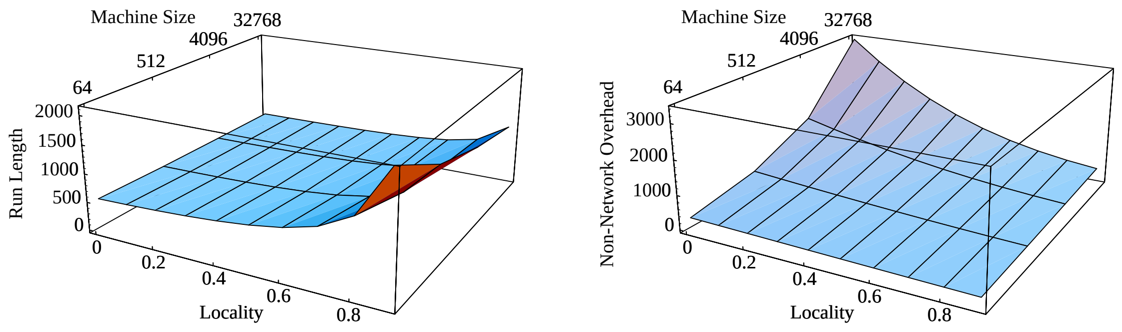
In all three architectures studied, multithreading is only useful up to two threads. In other words, interleaving more than two threads does not increase the transaction issue rate. For small machines and high locality applications, this limitation is due mainly to the length of the running threads. For large machines with medium to low locality, this limitation is due mainly to the protocol overhead being too large.

Inter-Transaction Issue Time

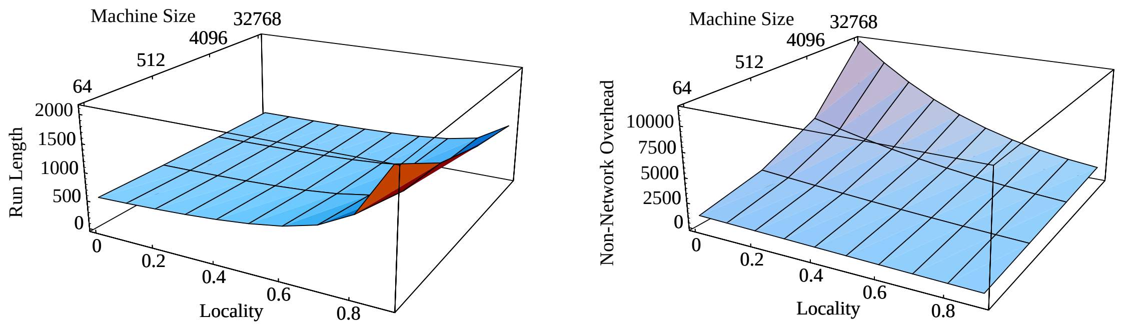
Figure 6.3 shows the average inter-transaction issue time for one thread and two threads. Note that increasing the number of threads from one to two provides little speedup. As expected, the lower protocol processing times create much better transaction issue times. Since the run length varies for different machine sizes and localities, we must look at what percentage of time is taken up in the protocol overhead.

Protocol Overhead

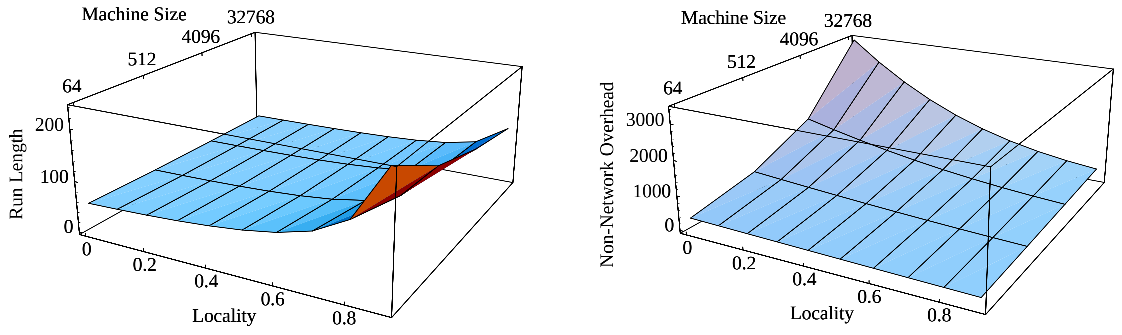
We examine the protocol overhead in order to see how much of the transaction latency is caused by overhead and how much represents work being done. Overhead (O) is defined as the fraction of the average transaction issue time not spent running:



Optimistic-Optimistic ($R = 500$; $M_r = 20$)

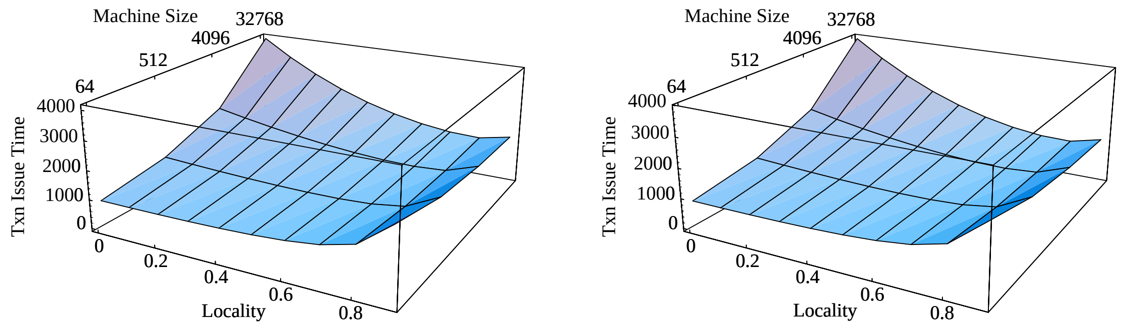


Optimistic-Pessimistic ($R = 500$; $M_r = 100$)

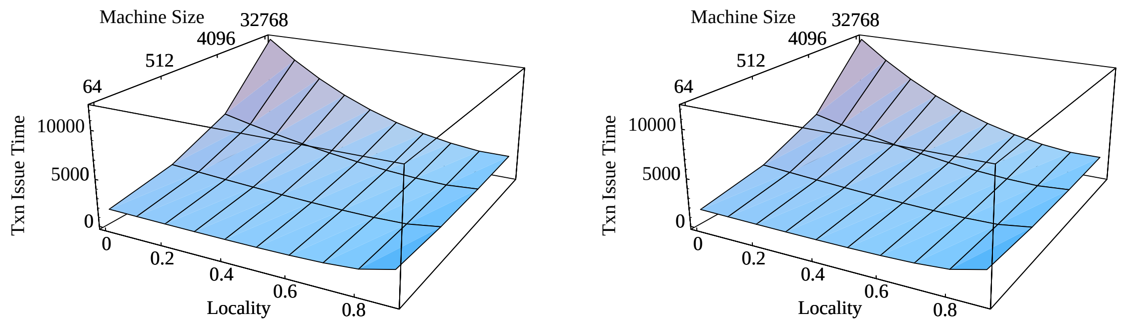


Pessimistic-Optimistic ($R = 50$; $M_r = 20$)

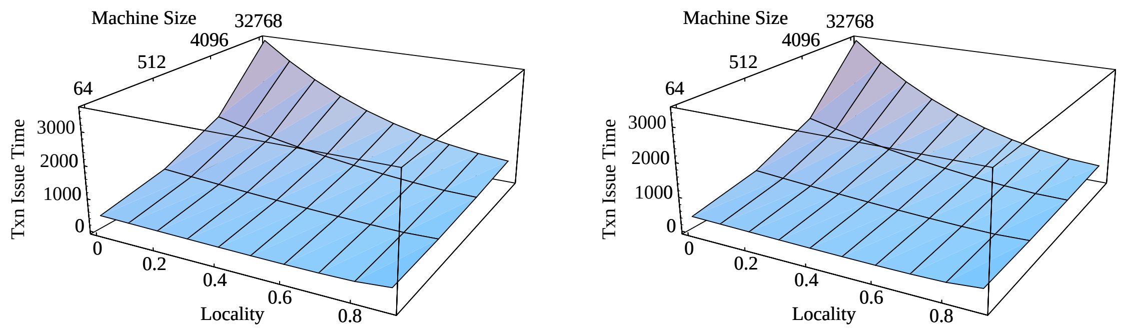
Figure 6.2: The left half of this figure shows the predictions for the average run length between off-node references to shared memory as a function of machine size and of locality. The right side shows the predicted protocol processing overhead time (dependent on the number of messages sent) per off-node shared-memory request.



Optimistic-Optimistic ($R = 500$; $M_r = 20$)



Optimistic-Pessimistic ($R = 500$; $M_r = 100$)



Pessimistic-Optimistic ($R = 50$; $M_r = 20$)

Figure 6.3: This figure shows the predictions for the average inter-transaction issue time as a function of machine size and of locality. The graphs on the left side are for no multithreading ($p_m = 1$); the graphs on the right are for a multithreading of 2 ($p_m = 2$).

$$E[O] = 1 - \frac{T_r}{t_t} \qquad (6.24)$$

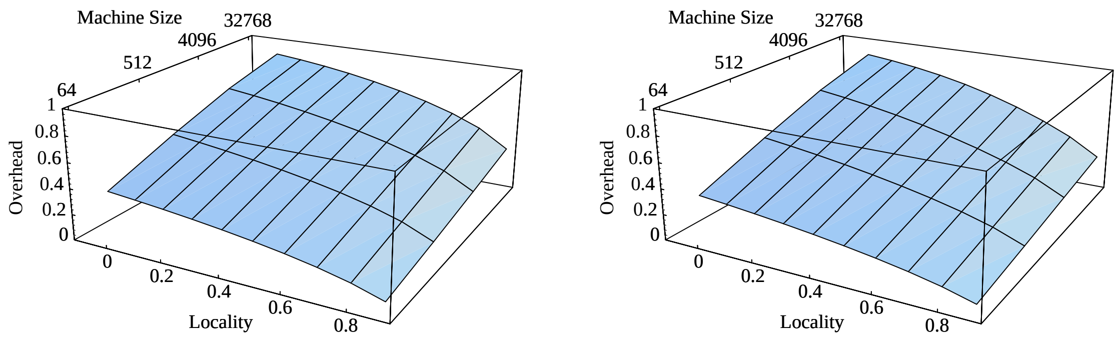
Figure 6.4 shows these results.

The Optimistic-Optimistic case does best, as expected. The Optimistic-Pessimistic case is tolerable for small machines with high locality. The Pessimistic-Optimistic case, on the other hand, shows extremely high overhead for nearly all conditions. If the typical run length is only 50 cycles, as assumed for this case, the protocol processing time needs to be reduced before this system can be effectively used. Note that there is very little speedup from going to two threads; in general, only small machines with poor locality benefit.

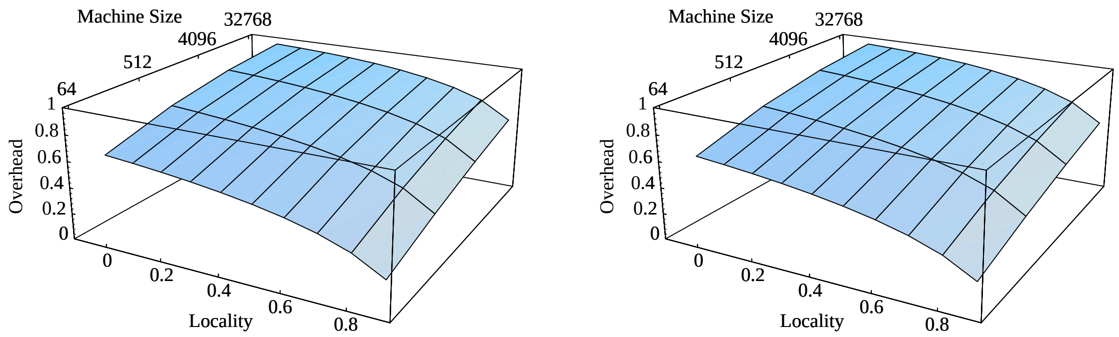
6.2.3 Architectures With A Separate Cache Controller

In order to increase the performance of the protocol, we consider the case where a separate controller exists to handle protocol requests. This situation will only be beneficial in tw cases: first, where the controller can be added to the system more cheaply than another processor, and secondly, where the controller can be designed to be significantly faster than a processor. If neither of these conditions are true, there is no benefit to using a controlle

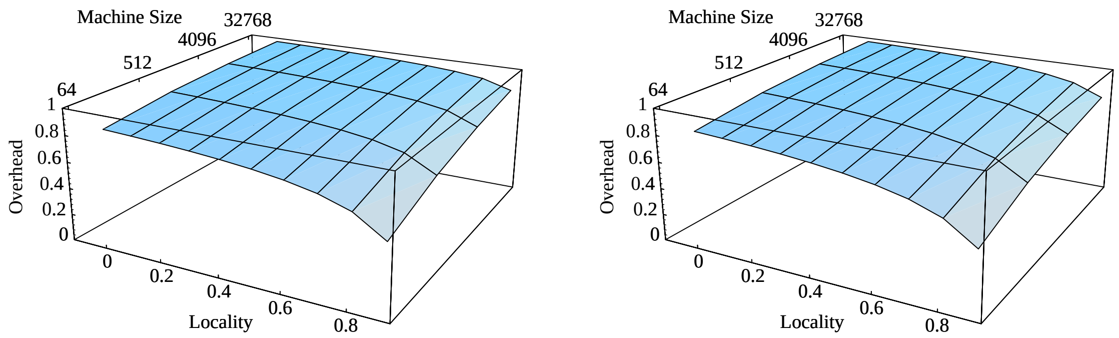
We model the architectures with a separate cache controller by allowing the inter-transaction issue time to decrease until it reaches the limits caused by either the protocol overhead or the run-length overhead. We assume that the controller operates at the same speed as the processor did in the earlier experiment; the gains all come from having a separate protocol handler, not from immense controller speed. For the Optimistic-Optimistic case ($R = 500$; $M_r = 20$), multithreading up to four results in better inter-transaction issue times. For the Optimistic-Pessimistic ($R = 500$; $M = 100$) case, up to eight threads can be profitably used to reduce latency. For the Pessimistic-Optimistic ($R = 250$; M case, only four threads provide speedup. Again, these limitations are due to the thread run length for small machines, or machines with very high locality, and to the protocol overhead for large machines with medium to low locality.



Optimistic-Optimistic ($R = 500$; $M_r = 20$)

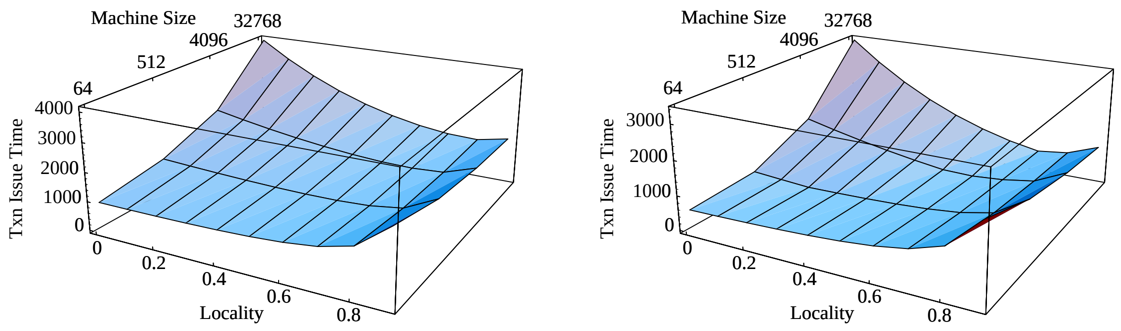


Optimistic-Pessimistic ($R = 500$; $M_r = 100$)

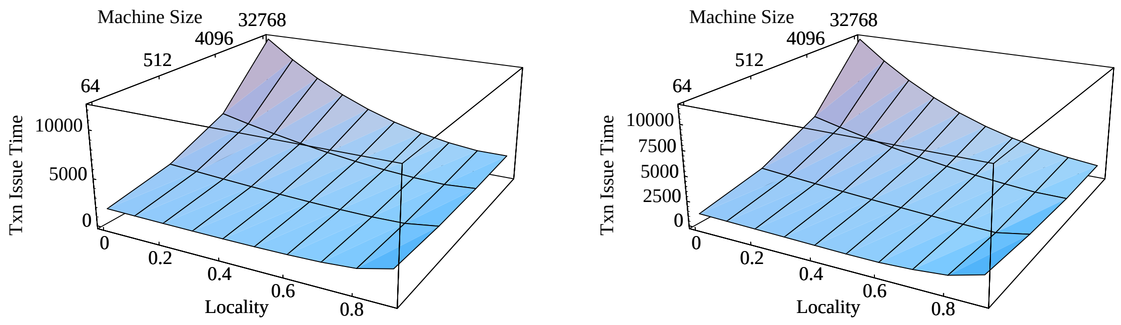


Pessimistic-Optimistic ($R = 50$; $M_r = 20$)

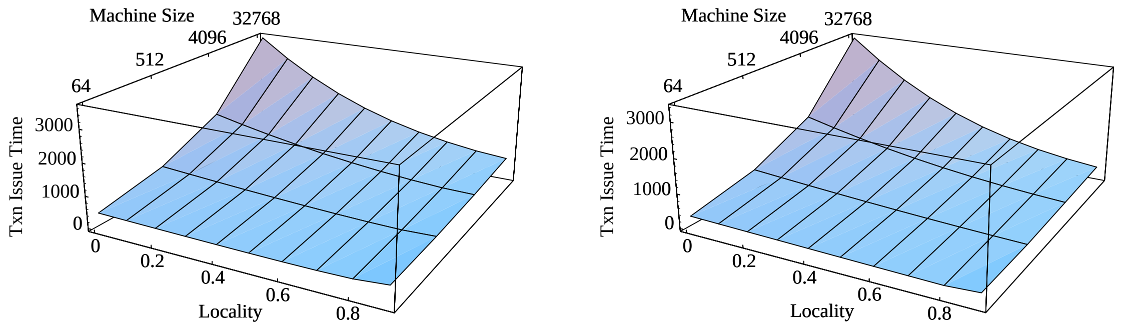
Figure 6.4: This figure shows the predictions for the protocol overhead as a function of machine size and of locality. The graphs on the left side are for no multithreading ($p_m = 1$); the ones on the right are for a multithreading of 2 ($p_m = 2$).



Optimistic-Optimistic ($R = 500$; $M_r = 20$)



Optimistic-Pessimistic ($R = 500$; $M_r = 100$)



Pessimistic-Optimistic ($R = 50$; $M_r = 20$)

Figure 6.5: This figure shows the predictions for the average inter-transaction issue time as a function of machine size and of locality. The graphs on the left side are for no multithreading ($p_m = 1$); the graphs on the right are for the largest possible useful multithreading, as described in the text.

Inter-Transaction Issue Time

Figure 6.5 shows the average inter-transaction issue time for one thread and for the maximum number of useful threads, as described above. Note that we do now see some improvement due to multithreading, which can be better observed in the overhead graphs.

Protocol Overhead

We again examine the protocol overhead in order to see how much of the transaction latency is caused by overhead and how much represents work being done. Figure 6.6 shows these results.

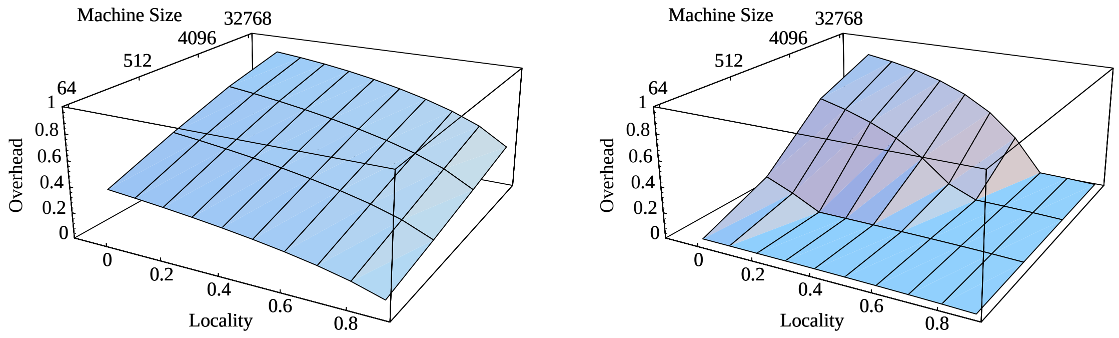
These results are much better than before. The use of a controller provides enormous gains in practicality. For an Optimistic-Optimistic architecture, we can expect to efficiently run the protocol at locality as low as 0.7 even on very large machines. The Optimistic-Pessimistic architecture performs much better than before, although it still has too much overhead for large machines. The Pessimistic-Optimistic architecture has also improved, but one would still not want to use the protocol with this embedding on such an architecture.

Most of the speedup occurs when going from one thread to two. The gains from going beyond that are small, and occur only on the boundary between too much work and too much overhead. For the Optimistic-Optimistic architecture, the gains occur on the diagonal line between large machines with lots of locality and small machines with little locality. For the Optimistic-Pessimistic case, the line moves closer to the small machines with high locality. This trend extends to the Pessimistic-Optimistic case, implying that the gains all occur only for small machines with high locality.

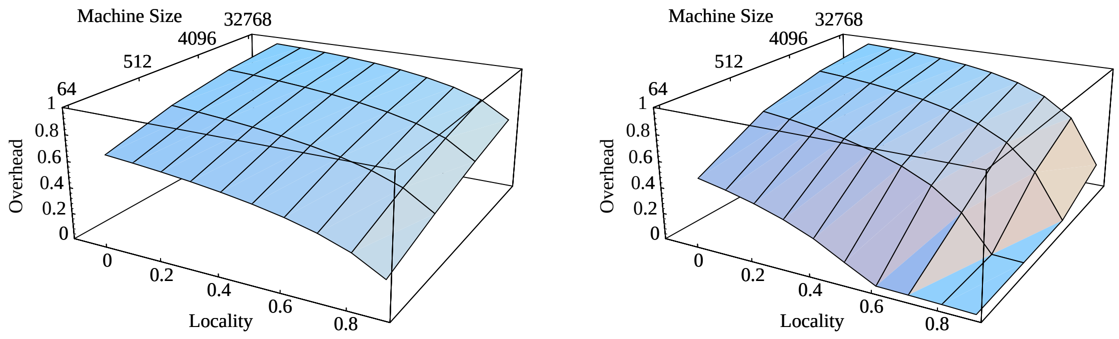
Note that these gains are of course not large enough to justify controllers which are more expensive than processors. In no case does the addition of the controllers save more time than double the number of processors.

6.3 Issues

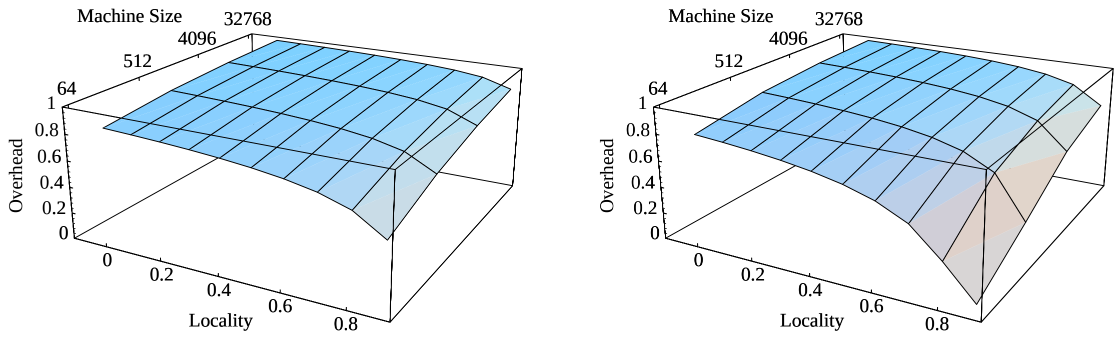
The results described in this chapter provide some insight as to how the protocol actually behaves when mapped to a k-ary n-cube in the manner described in Section 2.2. There



Optimistic-Optimistic ($R_r = 500$; $M = 20$)



Optimistic-Pessimistic ($R_r = 500$; $M = 100$)



Pessimistic-Optimistic ($R_r = 50$; $M = 20$)

Figure 6.6: This figure shows the predictions for the protocol overhead as a function of machine size and of locality. The graphs on the left side are for no multithreading ($p_m = 1$); the ones on the right are for the largest possible useful multithreading, as described in the text.

are many more interesting experiments to be done, however. First, other configurations of machines which would work better with the protocol should be studied. Second, other embeddings of the protocol to k-ary n-cubes should be considered. Finally, other cache coherence protocols should be studied to determine the competitiveness of the performance of the Protocol for Hierarchical Directories.

The main limitation of using this mapping to embed the protocol to an architecture is clearly the protocol overhead. There are several ways to fix this problem. One is to build fast controllers which can independently process the protocol requests, which is one of the goals of the MIT Alewife project [3]. The cost of adding such a controller to the machine must be balanced against the potential speed benefits.

Another way to reduce the overhead is to guarantee that high locality is maintained, with references to shared memory rare in comparison to the time needed to process protocol requests. In order to do this, compiler technology for static data placement must be improved. Programs must be compiled specifically to reduce the amount of data sharing. This technology would benefit all cache coherence protocols.

None of the architectures studied in this chapter was ever limited by the speed of the network. This indicates that either the assumptions imply a processor-network speed mismatch, and that the network is too fast, or that the protocol is fundamentally too slow. Studies using a very fast controller with fast processors, or fast controllers and slow proces sors could be used to evaluate how the protocol performance is affected by the embedding.

This study does not indicate whether or not PHD would be more useful for large machines than schemes with limited-directories, or even without caching. A study comparing these schemes for different values of the locality parameter would be very enlightening. We believe that PHD will perform best on large machines with decent hierarchical locality, and low protocol overhead. Whether or not these conditions will occur for real applications is unknown.

6.4 Summary

In this chapter we used an embedded model to show the performance of the protocol mapped onto various architectures. We looked at average inter-transaction issue time and protocol overhead for different locality parameters, multithreading, and machine sizes.

We determined that multithreading is only useful for approximately two to four threads; any additional interleaving does not decrease the overall latency. For small machines and high locality applications, this limitation is due mainly to the length of the running threads For large machines with medium to low locality, this limitation is due mainly to the high protocol overhead.

We discovered that the embedding will work well given fast protocol processing time and relatively few references to shared memory. In the best case only 9% of all cycles are taken up by protocol overhead for small machines with 0.9 locality. This increases to 28% for large machines (32768 processors) with high locality, and 39% for small machines with poor locality.

With the use of separate cache controllers, we can do even better. For a locality of 0.9, we can reduce the overhead to 1% overhead for up to 32768 processors. For a locality of 0, we can see as little as 4% overhead for 64 processors, rising rapidly as the number of processors increases. The gains from the addition of these controllers, however, are not large enough to justify hardware which is more expensive than processors. In no case does the addition of the controllers save more time than double the number of processors.

Chapter 7

Conclusion

7.1 Summary

This thesis described the Protocol for Hierarchical Directories, a hierarchical, director
based cache coherence scheme. PHD supports read, write, and test-and-set operations.
Read requests are satisfied in the smallest subtree containing both the requester and a
copy of the requested block; only three sets of messages are sent up or down that subtree.
Write requests are confined to the subtree containing the lowest common ancestor of the
requester and all copies of the requested block; four sets of messages are sent up and down
the hierarchy, two of which fan out to all nodes with copies. Test-and-set requests are
implemented as an optimized combination of read and write requests, and implement a
test-and-test-and-set operation.

An embedding of PHD into k-ary n-cubes was also proposed and evaluated. The mapping translates hierarchical locality into physical locality. The mapping also distribute
higher level tree nodes over many physical processors, both to increase bandwidth and to
prevent bottlenecks at the top of the tree.

We built a simulator to experiment with PHD. The simulator implements the full protocol plus certain extensions, such as local allocation and optional automatic allocation o
uninitialized data. The simulator is trace-driven, and can gather many types of statistics for
studying the protocol. The simulator has been used to test the protocol; additional features
for debugging include printing events and cache emptying events. A special verification

program was also designed to ensure that the protocol kept the memory consistent.

This thesis describes two analytical models: an abstract one and an embedded one. The abstract model characterizes aspects of the protocol which are not dependent on the architecture on which the protocol is run and can be used to evaluate other hierarchical protocols. The embedded model describes the behavior of PHD as it interacts with a machine which has particular network and processor characteristics. The embedded model derives its inputs from the outputs of the abstract model.

7.2 Contributions

PHD is scalable in cost and network latency. Unlike other hierarchical protocols, there is no bottleneck at the top of the hierarchy. The protocol uses fewer hierarchy traversals and a shorter critical path to satisfy read operations than do other hierarchical protocols. The protocol supports asynchronous invalidation through the notion of ownership.

We proposed a method of expressing locality in applications mapped onto hierarchical architectures and successfully used this model to predict the average height per request, the average longest path per request, and the average number of messages sent per request. We used three applications in order to validate this abstract model: a uniform reference stream, a relaxation algorithm, and a clustering data-reference stream. After validating the model, we employed it in the prediction of the behavior of the protocol on very large hierarchies, studying how the model results varied with machine size and locality.

This abstract model was used to generate the inputs to an embedded model; the embedded model described how the protocol runs when mapped onto particular machines. We looked at average inter-transaction issue time and protocol overhead for different locality parameters, degrees of multithreading, and machine sizes.

The embedding performs well when the run length between references to shared memory is at least an order of magnitude less than the time spent to process a protocol state transition. If separate controllers for processing protocol requests are included, the proto scales to 32k processor machines as long as applications exhibit hierarchical locality: at lea 22% of the global references must be able to be satisfied locally; at most 35% of the global references are allowed to reach the top level of the hierarchy. Without the use of separate controllers, latency cannot be hidden effectively by multithreading because processors spend too much of their time satisfying protocol requests.

7.3 Discussion

This thesis has exposed several major areas of research which should be pursued. The tradeoffs involved in designing good hierarchical cache coherence protocols should be characterized. The abstract model of the protocol would benefit from a better understanding of the locality parameter. With some additional work, the embedded model can be used to aid in the design of shared-memory machines.

This thesis discussed some of the decisions which were made in the design of a hierarchical cache coherence system. The effects of these decisions have not been fully explored. A comparison of PHD and another hierarchical cache coherence protocol would still be instructive.

Currently, the locality parameter set can only be determined for an application by simulation. We have performed some initial work towards deriving the set of locality parameters from a spatial locality model of an application, such as that available for the cluster application. The derivation works best, however, for applications which exhibit a very high degree of clustering. More work needs to be done in this area.

All of our large machine studies use a flat set of locality parameters to constrain the study space. Using a flat set of locality parameters, however, is not necessarily realistic. Fo large applications running on massively parallel machines, we might expect less sharing to occur near the top of the hierarchy, and more at the bottom. Although few large applications exist today, as ones are written they can be studied in order to determine reasonable locality parameters.

For applications which have a large variance in the types of data referencing, several sets of locality parameters can be used, to avoid averaging effects. This would allow one to separate widely shared data such as synchronization variables from less used ones. This separation would be useful because an application may stall due to synchronization instead of normal changed data. This method might also provide new insight into the interactions between shared data and program execution-time behavior.

Chapter 6 shows that the main limitation of using the mapping proposed in this thesis to embed the protocol to a k-ary n-cube is the potentially high protocol overhead. Several ways to fix this problem should be explored. One is to build fast controllers which can independently process the protocol requests, the approach of the MIT Alewife project [3]. The cost of adding such a controller to the machine must be balanced against the potential speed benefits.

None of the architectures studied in this thesis was limited by the speed of the network. This indicates that either there is a processor-network speed mismatch, and that the network is too fast, or that the protocol is fundamentally too slow. Studies using a very fast controlle with fast processors, or fast controllers and slow processors could be used to evaluate how the protocol performance is affected by the layout, and possibly how to build shared-memory machines.

There are many areas left to be explored. Uppermost in our minds is the question of whether or not hierarchical protocols will perform better than flat directory schemes, or even no caching at all, for actual applications. Determining exactly where the tradeoffs are in complexity, technology, application locality, compilation time, and machine size would be extremely enlightening. We believe that PHD will perform best on large machines with low protocol overhead running applications exhibiting hierarchical locality patterns. Whether or not these conditions will occur for real applications is unknown.

Regardless of what cache coherence scheme is chosen compilers must be developed to minimize data sharing. The scheduling of processes and the placement of data will be some of the most important problems in building massively parallel computer systems.

Appendix A

Nomenclature

B	Average message size (in flits).
C	Number of words in a cache line.
E_l	Probability a node accesses blocks owned by a node in its subtree of l nodes. b
L	Number of levels in the hierarchy.
M_l	Average time to satisfy a locally satisfiable request to shared memory.
M_r	Average time to process a protocol message invoked on a processor.
N	Number of processors.
N_i	Average network interface overhead
O	Overhead: Fraction of the average transaction issue time not spent running.
R	Average thread run length between successive requests to shared memory.
R_l	Number of reads which reach level l.
T_f	Non-network overhead to satisfying a non-local shared-memory request.
T_r	Average thread run length between successive non-locally satisfiable requests.
T_s	Context switch time.
W_l	Number of writes which reach level l .
$Z_r, Z_w,\ Z$	Fraction of locally satisfiable {reads, writes, requests} to shared memory.

Table A.1: Part I of the table listing all of the parameters used by the thesis. Part II is located on the next page.

b	Branching factor of the hierarchy.
c_l	Number of valid children of a valid node at level l .
c_r, c_w, c	Average number of messages in critical path of a non-local shared-memory request.
d	Average distance a message travels (in hops).
d_l	Expected number of hops between a node and its physically distinct parent.
d'_l	Expected number of hops a message which circumvents the hierarchy will take.
e	Fraction of all operations by a node which occur to its own data.
f	Number of flits per word.
g_r, g_w, g	Average number of messages per non-local shared-memory {read, write, request}.
h_r, h_w, h	Average height a {read, write, request} is expected to reach.
k	Number of processors per dimension.
k_d	Average distance a message travels in each dimension.
l_r, l_w, l	Longest path traversed during a {read, write, request}.
m_r, m_w, m	Average number of messages sent during a {read, write, request}.
n	Number of dimensions.
p_l	Value of the locality parameter at level l .
p_m	Degree of hardware multithreading.
r	Fraction of reads in the shared-memory reference stream
t_l	Probability of taking a valid path down from level l during node selection.
t_t	The average inter-transaction issue time.
v_l^c	Probability that c children of a valid node at level l are valid.
w	Fraction of writes in the shared-memory reference stream

Table A.2: Part II of the table listing all of the parameters used by the thesis.

Appendix B

Relaxation Calculations

The height of read and write operations for a given relaxation problem can be exactly calculated, as mentioned in Chapter 5. This appendix presents the analytical equations for two and three dimensional relaxation calculations.

Calculating the Characteristics of Read Operations The first characteristic of application that must be understood is the number of read operations to neighboring values that are local, and the number that cross various levels of the hierarchy. We will first show a derivation for the 2-dimensional numbers and then the 3-dimensional numbers. Call n the number of processors per dimension, N the total number of processors, x the number of data points per processor per dimension, and X the number of data points per processor. Note that in the rest of the thesis, k is the number of processors; we use n here for simplicity.

As can be seen in Figure B.1, the read references which reach the highest level in the system, B_{L-1}, will be the ones by points of data abutting the boldest lines, which represents the division between the four level 3 processors. The number of reads which cross these lines is $4nx$. The number of reads which cross the next highest level is $4(2nx)$. In general, the number of reads which cross level l is twice as many as cross level $l+1$, for all $l \in [1, L-2]$.

For three dimensions, the read reference calculation is similar. Here we are dealing with planes instead of lines. The number of reads which cross the highest plane is $6n^2x^2$. As in the 2-dimensional case, the number of reads which cross level l is twice as many as cross level $l+1$, for all $l \in [1, L-2]$.

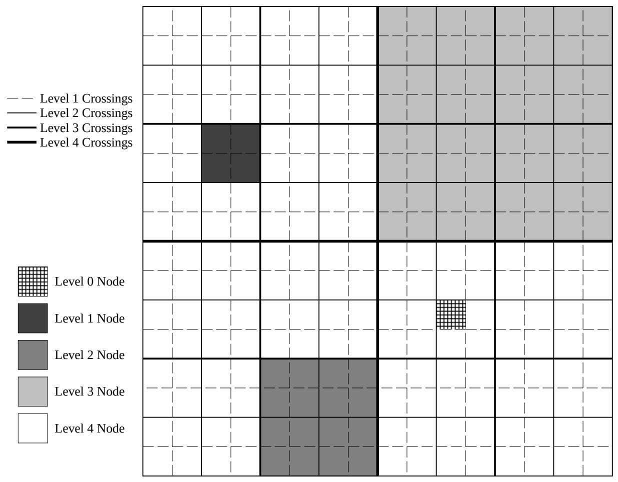


Figure B.1: The tree is mapped to the processors in such a way that crossing the bolder lines represents reaching higher levels of the tree.

In fact, we can perform this calculation for an arbitrary d dimensional embedding. The $L-1$ crossing happens for exactly $2d(nx)^{D/4}$ reads, where $D = 2^{d-2}$. The reads always double as the level decreases. The number of read references reaching each level for an arbitrary dimension is summarized in Equation B.1.

$$R_l = \begin{cases} DNX - \sum_{h=1}^{L-1} R_h & l = 0 \\ 2R_{l+1} & l \in [1, L-2] \\ 2d(nx)^{2^{d-2}} & l = L-1 \end{cases} \quad \text{(B.1)}$$

Calculating the Characteristics of Write Operations We are now prepared to calcu late the exact number of writes which must reach a particular height. Instead of summing, for every point, the heights of its neighbors, we must perform a maximum. As can be seen in Figure B.2, there are many grid points that have neighbors at varying heights. Data point a is a typical data point, with all of its neighbors local. The maximum height a write to this point could reach, therefore, is 0. Data point b has a neighbor which is across a level 1 boundary. Since the rest of its neighbors are local, the maximum height is 1. Both points c and d have neighbors across level 2 boundaries, so must be counted at height 2. The three points e, f, and g are similarly counted at height 3, and h, i, j, and k are similarly counted

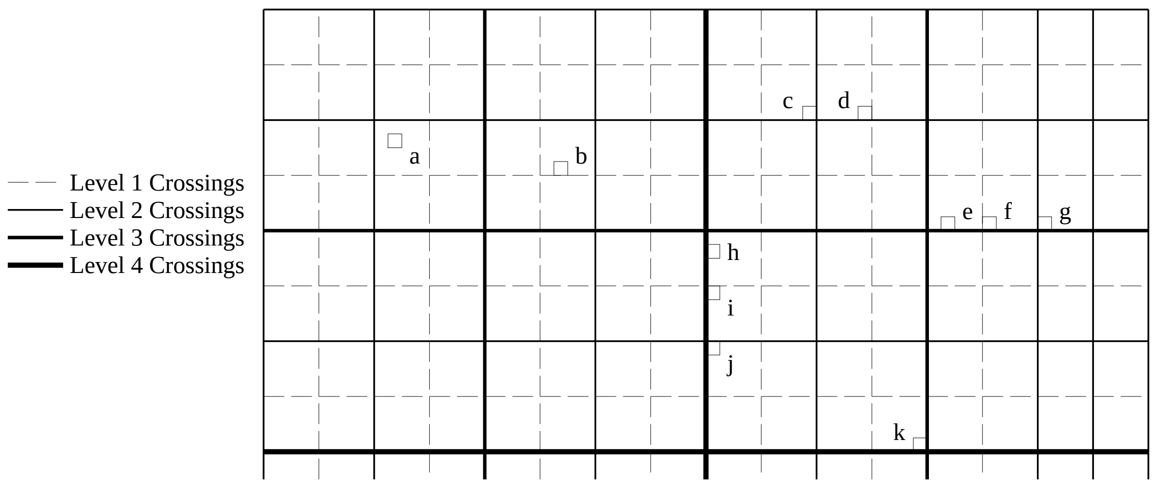


Figure B.2: The labeled points represent pieces of data which must be carefully considered when determining the height that a write to that data will reach.

at 4.

All points to be counted at $L-1$ can be easily calculated as $W_{L-1} = 4nx-4$. The $4nx$ was derived for the read case, and the subtraction of 4 refers to the four cross points each of which was double counted in the read formula.[1] To calculate the formula for $l \in [1, L-2]$, we count all of the points on a cross for size l, subtracting out the four center ones as in the $l = L-1$ case, and then multiply that quantity by the number of crosses at that level. We then must subtract off all points which are supposed to be counted as higher-level points. The resulting formula, which applies only to the two-dimensional case, is given in Equation B.2. Note that C_l, the number of crossing points associated with each level, is 2^{L-1-l}, and G_l, the number of processors at level l, is D^{L-1-l}.

$$W_l = \begin{cases} n^2(x-2)^2 + 4n(x-2) + 4 & l = 0 \\ G_l\left(\frac{4nx}{C_l} - 4\right) - 8(C_l)(G_l - 1) & l \in [1, L-2] \\ 4nx - 4 & l = L-1 \end{cases} \quad \text{(B.2)}$$

Calculating the number of data points which have completely local neighbors is fairly

[1]This double count is appropriate for a read, since more than one read occurs to every block.

simple. Note that we are assuming that points which lie on the boundary are read fewer times (as many fewer times as neighbors they lack). There are $n^2(x-2)^2$ completely local points per node, plus $x-2$ boundary points per edge node, plus an extra point (the corner one) on each corner node.

In the 3-dimensional case we must consider three intersecting planes instead of two intersecting lines. The number of writes which reach the highest level, however, is still straightforward: from the read case we know that there are $6n^2x^2$ crossings of the three planes. The three planes intersect at three separate lines, each of which generates four double-counted grid units per line unit which must be subtracted from the earlier total. The three lines, however, intersect in one point which has eight double-counted grid units around it. These eight grid units must be added back to the total, resulting in the formula given in Equation B.3.

$$W_l = \begin{cases} n^3(x-2)^3 + 6n^2(x-2)^2 + 12n(x-2) + 8 & l = 0 \\ G_l\left(\frac{6n^2x^2}{C_l^2} - \frac{12nx}{C_l} + 8\right) - 24nxC_l(C_l-1) & \\ \qquad\qquad + 24\left(C_l^2(C_l-1)\right) + \left((C_l-1)^2C_l\right) & l \in [1, L-2] \\ 6n^2x^2 - 12nx + 8 & l = L-1 \end{cases} \tag{B.3}$$

We now calculate the equation for $l \in [1, L-2]$ for the 3-dimensional case. First we consider each level l subunit. There are D^{L-1-l} such subunits. As in the $l = L-1$ case we count all the points along the three planes, subtract off the line ones, and add back in the center eight ones. We now must account for the all the points which are counted at a higher level. These points are the ones at the boundaries of the subunits. We will calculate these from looking at the whole cube, not at subunits. Consider a face of the cube, as in Figure B.3. Each dotted-line cross in the right side of the figure is the edge view of the 3-dimensional object shown at the left side of the figure. The bold lines on the left figure indicate which points have been double-counted, and must be subtracted out of the total. Each diamond and circle in the right figure represents a line that must be subtracted out. There are $C_l(C_l-1)$ circle lines, and the exact same number of diamond lines, per dimension. For every circle or diamond line, $4nx$ points must be subtracted out.

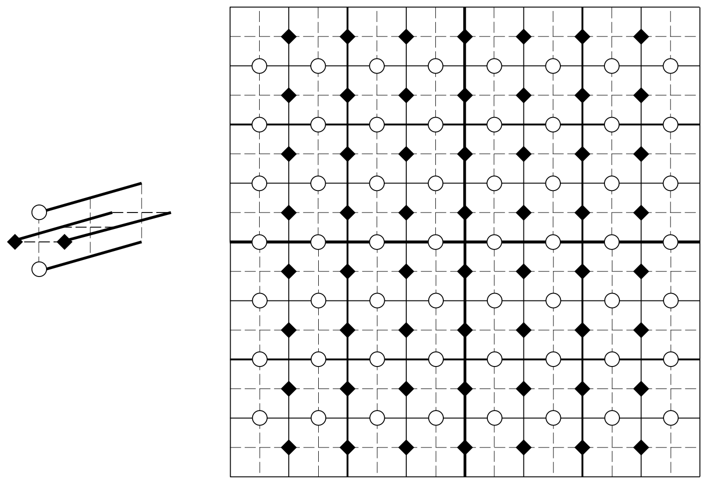


Figure B.3: Each dotted line cross is the edge of the intersection of two planes. At the boundaries of these 3-d crosses are lines (the endpoints of which are marked as circles and diamonds) which contain the points that are supposed to be counted at a higher level.

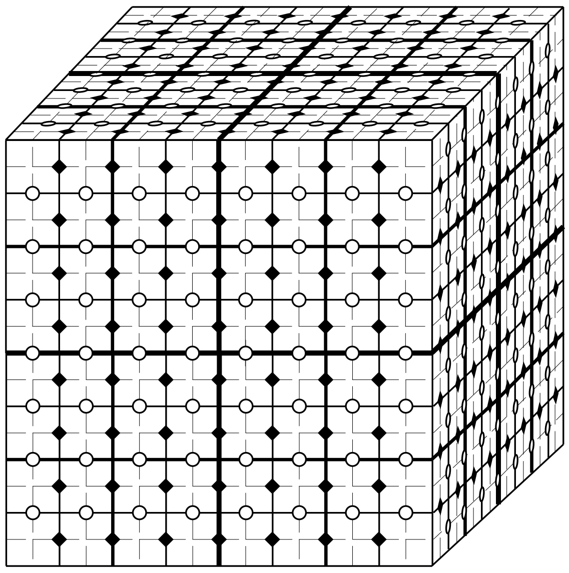


Figure B.4: The circles and diamonds represent the endpoints of lines. The intersections of these lines have been doubly subtracted in our total, and must be added back.

After subtracting out all of those points, we still do not have the correct equation. Everywhere the circle and diamond lines intersected, we double-subtracted points, and we must now add them back. Consider Figure B.4. Study the front and top faces of the cube. We can see that per front column of circles, C_l-1 front circles intersect C_l-1 top circles. There are C_l front columns of circles. Similarly, per front column of diamonds, C_l front diamonds, C diamonds intersect C_l top diamonds. There are C_l-1 front columns of diamonds. This set of intersections occurs once for every pair of dimensions (*i.e.* three times), and generates eight points to be added back per crossing. The resulting formula was already shown in Equation B.3.

We now calculate the number of data points which have completely local neighbors for the 3-dimensional case. There are $(x-2)^3$ completely local points per node, plus $(x-2)^2$ boundary points per face of the cube node, plus $x-2$ boundary points per edge node, plus an extra point (the corner one) on each corner node. Again, the resulting formula was already shown in Equation B.3.

Appendix C

Table of Protocol Behavi

The following sections detail the behavior of the Protocol for Hierarchical Directories. The first describes the transitions for leaf nodes, and the second describes the transitions fo parent nodes.

C.1 Leaf Node Transition Table

The leaf node transitions are a function of the current state and the input message. For every such combination, there is a possible new state to transition to as well as a possible message to send. The possible states are enumerated in Table C.1. Table C.2 lists all of the messages that can be received by a leaf node. These messages are explained in Table 2.4. Table C.3 explains all of the symbols used in the transition table. The messages which can be sent by a leaf node are listed in C.4, and a further expansion of the abbreviations is listed in Table C.7 . The actual transition table is split onto two pages in Table C.5.

Symbol	Expansion
invalid	invalid
r_yo_npl	readable_yowner
r_no_npl	readable_nowner
wfr_no_npl	waiting_for_read
w_yo_npl	writable
wfw_no_npl_nr	waiting_for_write_nowner_npl_nread
wfw_no_npl_yr	waiting_for_write_nowner_npl_yread
wfw_yo_npl	waiting_for_write_yowner_npl
wfw_no_ypl_nr	waiting_for_write_nowner_ypl_nread
wfw_no_ypl_yr	waiting_for_write_nowner_ypl_yread
wfwo_yo_npl	waiting_for_write_ok_yowner_npl
wfwo_yo_ypl	waiting_for_write_ok_yowner_ypl
wfwv_no_ypl_nr	waiting_for_write_value_nowner_ypl_nread
wfwv_no_ypl_yr	waiting_for_write_value_nowner_ypl_yread
wft_no_npl_nr	waiting_for_tas_nowner_npl_nread
wft_no_npl_yr	waiting_for_tas_nowner_npl_yread
wft_yo_npl	waiting_for_tas_yowner_npl
wft_no_ypl_nr	waiting_for_tas_nowner_ypl_nread
wft_no_ypl_yr	waiting_for_tas_nowner_ypl_yread
wfto_yo_npl	waiting_for_tas_ok_yowner_npl
wfto_yo_ypl	waiting_for_tas_ok_yowner_ypl
wftv_no_ypl_nr	waiting_for_tas_value_nowner_ypl_nread
wftv_no_ypl_yr	waiting_for_tas_value_nowner_ypl_yread

Table C.1: The abbreviations for states used in the leaf transition table.

Symbol	Expansion
dr	read request
r	read
rd	read_data
dw	write request
l	lock
swo	s_write_own
wo	write_ok
dt	tas request
rt	read_tas
tf	tas_failed

Table C.2: The abbreviations for input messages and requests used in the leaf transition table.

Symbol	Expansion
.	Stay in the same state.
NUMBER	Go to the state numbered NUMBER.
D	Put message in delaying queue.
X	Error.
DX	X if we issue only one request at a time, D otherwise.
z: ACTION	If value of the block is zero then ACTION.
mrq: ACTION	If current node originated the request then ACTION.
!: ACTION	Else ACTION.

Table C.3: The abbreviations for symbols used in the leaf transition table.

Symbol	Expansion
fr	Send flcfr up to parent.
rfr	Send rflcfr up to parent.
fw	Send flcfw up to parent.
a	Send a up to parent.
a1	Send a1 up to parent.
ft	Send flcft up to parent.
rft	Send rflcft up to parent.
cv	Send cv up to parent.
rd	Send rd to the reader.
sw	Send sw to the writer.
tf	Send tf to the tas requester.

Table C.4: The abbreviations for output messages used in the leaf transition table.

		dr	r	r d	dw	l	s wo	wo	dt	r t	t f
0	i nval i d	f r 3	r f r .	X	f w 5	mr q: X ! : a .	X	X	f t 14	r f t .	X
1	r _yo_npl	. 	r d .	X	f w 7	s wo a 0	X	X	f t 16	z : f w . ! : t f .	X
2	r _no_npl	.	r d .	X	f w 6	a 0	X	X	f t 15	z : f w . ! : t f .	X
3	wf r _no_npl	DX .	r f r .	c v 2	DX .	D .	X	X	X	r f t .	X
4	w_yo_npl	.	r d 1	X	.	s wo a 0	X	X	.	z : f w . ! : t f .	X
5	wf w_no_npl _nr	DX .	r f r .	X	DX .	mr q: a1 8 ! : a .	10	X	DX .	r f t .	X
6	wf w_no_npl _yr	DX .	r d .	X	DX .	mr q: a1 9 ! : a 5	10	X	DX .	r f t .	X
7	wf w_yo_npl	DX .	r d .	X	DX .	mr q: s wo a1 9 ! : s wo a 5	10	X	DX .	r f t .	X
8	wf w_no_ypl _nr	DX .	r f r .	X	DX .	mr q: X ! : D .	11	12	DX .	r f t .	X
9	wf w_no_ypl _yr	DX .	r d .	X	DX .	mr q: X ! : D .	11	13	DX .	r f t .	X
10	wf wo_yo_npl	DX .	r d .	X	DX .	mr q: a1 11 ! : X	X	X	DX .	r f t .	X
11	wf wo_yo_ypl	DX .	r d .	X	DX .	mr q: X ! : D .	X	4	DX .	r f t .	X
12	wf wv_no_ypl _nr	DX .	r f r .	X	DX .	mr q: X ! : D .	4	X	DX .	r f t .	X
13	wf wv_no_ypl _yr	DX .	r d .	X	DX .	mr q: X ! : D .	4	X	DX .	r f t .	X

Table C. 5: The transition table for leaf nodes.

		dr	r	r d	dw	l	s wo	wo	dt	r t	t f
14	wf t _no _npl _nr	DX .	r f r .	X	DX .	mr q: a1 17 ! : a .	19	X	DX .	r f t .	0
15	wf t _no _npl _yr	DX .	r d .	X	DX .	mr q: a1 18 ! : a 14	19	X	DX .	r f t .	0
16	wf t _yo _npl	DX .	r d .	X	DX .	mr q: s wo a1 18 ! : s wo a 14	19	X	DX .	r f t .	0
17	wf t _no _ypl _nr	DX .	r f r .	X	DX .	mr q: X ! : D .	20	21	DX .	r f t .	0
18	wf t _no _ypl _yr	DX .	r d .	X	DX .	mr q: X ! : D .	20	22	DX .	r f t .	0
19	wf t o _yo _npl	DX .	r d .	X	DX .	mr q: a1 20 ! : X	X	X	DX .	r f t .	X
20	wf t o _yo _ypl	DX .	r d .	X	DX .	mr q: X ! : D .	X	4	DX .	r f t .	X
21	wf t v _no _ypl _nr	DX .	r f r .	X	DX .	mr q: X ! : D .	4	X	DX .	r f t .	X
22	wf t v _no _ypl _yr	DX .	r d .	X	DX .	mr q: X ! : D .	4	X	DX .	r f t .	X

C.2 Parent Node Transition Table

The state of a parent (non-leaf) node includes the full vector describing its child subtrees For this reason, the transition table must be collapsed (just eight subtrees increases the tota number of states by a factor of 2^{16}) in order to express it in a reasonable amount of room The table therefore takes three inputs: the current state (which does not includes the state of the subtree vector), the subtree vector combination, and the input message. In response to a message, a node may send a message, perform an action, change its own state, or any combination of the above. All of these responses may be modified by a conditional expression further specifying the state of the node. The table additionally contains assertions about the state of a node for some of the entries. These assertions are not required to implement the protocol, but are useful to understand what must be happening when a node reaches a particular state.

The list of states is enumerated in Table 2.3. The possible vector combinations are listed in C.6. The messages that a parent node might receive are listed in Table C.7; an explanation of these messages is in Table 2.4. The list of actions a node may perform is enumerated in Table C.8. Table C.9 lists the messages that might be sent by a parent node. Table C.10 explains all of the assertions and predicates used in the transition table. The actual node transition table spans multiple pages, and is referred to as Table C.11.

Symbol		Expansion
c	v0_w0_cX	All subtrees are either *confirmed* or *invalid*.
v	vX_w0_c0	All subtrees are either *valid* or *invalid*.
vw	vX_wX_c0	All subtrees are *valid*, *waiting*, or *invalid*.
vc	vX_w0_cX	All subtrees are *valid*, *confirmed* or *invalid*.
vwc	vX_wX_cX	All subtrees are *valid*, *waiting*, *confirmed* or *invalid*.

Table C.6: The abbreviations for the vector combinations used in the parent transition table.

Symbol	Expansion
flcfr	find_lowest_common_for_read
rflcfr	redirected_find_lowest_common_for_read
r	read
flcfw	find_lowest_common_for_write
l	lock
a	ack
a1	ack1
ta	throwing_away
cte	change_to_exclusive
flcft	find_lowest_common_for_tas
rflcft	redirected_find_lowest_common_for_tas
cv	confirm_value
rd	read_data
uncv	unconfirm_value
rt	read_tas
wo	write_ok

Table C.7: The abbreviations for input messages used in the parent transition table.

Symbol	Expansion
L	Lock this node and change the writer index.
+v	Change the sending subtree to valid.
+c	Change the sending subtree to confirmed.
+i	Change the sending subtree to invalid.
+w	Change the sending subtree to waiting.
+e	Change this node's status to exclusive.
+s	Change this node's status to shared.

Table C.8: The abbreviations for actions used in the parent transition table.

Symbol	Expansion
flcfr	Send flcfr up to parent.
rflcfr	Send rflcfr up to parent.
r	Send r down to randomly chosen confirmed subtree.
flcfw	Send flcfw up to parent.
l	Send l down to writing subtree.
a	Send a up to parent.
a1	Send a1 up to parent.
ta	Send ta up to parent.
cte	Send cte down to only non-invalid subtree.
flcft	Send flcft up to parent.
rflcft	Send rflcft up to parent.
cv	Send cv up to parent.
uncv	Send uncv up to parent.
rt	Send rt down to randomly chosen confirmed subtree.
wo	Send wo down to only non-invalid subtree.
rdav, c	Send rd down to all valid, making them confirmed.
rdavw, c	Send rd down to all valid or waiting, making them confirmed.
rdaw, c	Send rd down to all waiting, making them confirmed.
rdavL, c	Level 1: Send rd down to all valid except locker, making them confirmed. Level >1: Send rd down to all valid including locker, making all but locker confirmed.
rdavwL, c	Level 1: Send rd down to all valid or waiting except locker, making them confirmed. Level >1: Send rd down to all valid or waiting including locker, making all but locker confirmed.
rl	Send r down to the locking subtree.
la	Send l down to all not-invalid subtrees and the writing subtree.

Table C.9: The abbreviations for output messages used in the parent transition table.

Symbol	Expansion
rC	Sending subtree is not the only confirmed subtree.
rV	Sending subtree is not the only valid subtree.
iW	Sending subtree is not waiting.
iL	Sending subtree is not the locker of this node.
ii	Sending subtree is invalid.
ivwc	Sending subtree is valid, waiting, or confirmed.
ivw	Sending subtree is valid or waiting.
ivc	Sending subtree is valid or confirmed.
ic	Sending subtree is confirmed.
iv	Sending subtree is valid.
il	Sending subtree is the locker of this node.
svc	Subtree of requester if operation is valid or confirmed.
sv	Subtree of requester of operation is valid.
sc	Subtree of requester of operation is confirmed.
lv	Locker's subtree is valid.
lc	Locker's subtree is confirmed.
lW	Locker's subtree is not waiting.
0c	Zero subtrees are confirmed.
0C	At least one subtree is confirmed.
0v	Zero subtrees are valid.
0V	At least one subtree is valid.
T	This node is not the top level node in the hierarchy.
t	This node is the top level in the hierarchy.
1	This node is at level one.
¿1	This node is above level one.
1vwc	Exactly one subtree is valid, waiting, or confirmed.
1vw	Exactly one subtree is valid or waiting.
1vc	Exactly one subtree is valid or confirmed.
1v	Exactly one subtree is valid.
1c	Exactly one subtree is confirmed.
ORP	This node is on the request path of the current operation.
NRP	This node is not on the request path of the current operation.

Table C.10: The abbreviations for assertions used in the parent transition table.

FLCFR	RFLCFR	R	FLCFW

INVALID 0
if(t){
 alloc;
 next_state 2;
} else {
 do +v;
 send flcfr;
 next_state 8;
}
	INVALID0
	assert T;
	assert NRP;
	send rflcfr;
	next_state .;
		INVALID 0
		assert T;
		send rflcfr;
		next_state .;
			INVALID 0
			if(t){
			 alloc;
			 next_state 2;
			} else {
			 send flcfw;
			 next_state .;
			}

c S_U_NOP_NGA 1
assert rC;
do +v;
send r;
next_state 20;
	c S_U_NOP_NGA1
	if(ORP){
	 assert sc;
	}
	send r;
	next_state .;
		c S_U_NOP_NGA 1
		send r;
		next_state .;
			c S_U_NOP_NGA 1
			assert T;
			send flcfw;
			next_state .;

c E_U_NOP_NGA 2
assert rC;
do +v;
send r;
next_state 21;
	c E_U_NOP_NGA2
	if(ORP){
	 assert sc;
	}
	send r;
	next_state .;
		c E_U_NOP_NGA 2
		do +s;
		send r;
		next_state 1;
			c E_U_NOP_NGA 2
			do L;
			send la;
			if(ii){
			 do +v;
			 next_state 24;
			} else {
			 next_state 5;
			}

c S_L_NOP_NGA 3
if(ii){
 send D;
 next_state .;
} else {
 assert rC;
 do +v;
 send r;
 next_state 22;
}
	c S_L_NOP_NGA3
	if(ORP){
	 assert sc;
	}
	send r;
	next_state .;
		c S_L_NOP_NGA 3
		send r;
		next_state .;
			c S_L_NOP_NGA 3
			send D;
			next_state .;

c S_L_YOP_NGA 4
if(ii){
 send D;
 next_state .;
} else {
 do +v;
 send r;
 next_state 23;
}
	c S_L_YOP_NGA4
	if(ORP){
	 assert sc;
	}
	send r;
	next_state .;
		c S_L_YOP_NGA 4
		send r;
		next_state .;
			c S_L_YOP_NGA 4
			send D;
			next_state .;

Table C.11: The transition table for parent nodes.

FLCFR	RFLCFR	R	FLCFW

c E_L_YOP_NGA 5
if(ii){
send D;
next_state .;
} else {
do +v;
send r;
next_state 24;
}
	c E_L_YOP_NGA5
	if(ORP){
	assert sc;
	}
	send r;
	next_state .;
		c E_L_YOP_NGA 5
		if(NRP){
		send D;
		next_state .;
		} else {
		send r;
		next_state .;
		}
			c E_L_YOP_NGA 5
			send D;
			next_state .;

c S_L_YOP_YGA 6
if(ii){
send D;
next_state .;
} else {
do +v;
send r;
next_state 25;
}
	c S_L_YOP_YGA6
	if(ORP){
	assert sc;
	}
	send r;
	next_state .;
		c S_L_YOP_YGA 6
		send r;
		next_state .;
			c S_L_YOP_YGA 6
			send D;
			next_state .;

c E_L_YOP_YGA 7
if(ii){
send D;
next_state .;
} else {
do +v;
send r;
next_state 26;
}
	c E_L_YOP_YGA7
	if(ORP){
	assert sc;
	}
	send r;
	next_state .;
		c E_L_YOP_YGA 7
		if(NRP){
		send D;
		next_state .;
		} else {
		send r;
		next_state .;
		}
			c E_L_YOP_YGA 7
			send D;
			next_state .;

v S_U_NOP_NGA 8
assert rV;
do +w;
next_state 14;
	v S_U_NOP_NGA8
	assert T;
	if(ORP){
	assert sv;
	}
	send rflcfr;
	next_state .;
		v S_U_NOP_NGA 8
		assert T;
		send rflcfr;
		next_state .;
			v S_U_NOP_NGA 8
			assert T;
			send flcfw;
			next_state .;

v S_L_NOP_NGA 9
if(ii){
send D;
next_state .;
} else {
assert T;
send flcfr;
next_state .;
}
	v S_L_NOP_NGA9
	assert T;
	if(ORP){
	assert sv;
	}
	send rflcfr;
	next_state .;
		v S_L_NOP_NGA 9
		assert T;
		send rflcfr;
		next_state .;
			v S_L_NOP_NGA 9
			send D;
			next_state .;

FLCFR	RFLCFR	R	FLCFW

v S_L_YOP_NGA 10
if(ii){
 send D;
 next_state .;
} else {
 if(t—il){
  send rl;
  next_state .;
 } else {
  send flcfr;
  next_state .;
 }}
	v S_L_YOP_NGA10
	if(ORP){
	 assert sv;
	}
	if(t){
	 send rl;
	} else {
	 send rflcfr;
	}
	next_state .;
		v S_L_YOP_NGA 10
		send rl;
		next_state .;
			v S_L_YOP_NGA 10
			send D;
			next_state .;

v E_L_YOP_NGA 11
if(ii){
 send D;
 next_state .;
} else {
 send rl;
 next_state .;
}
	v E_L_YOP_NGA11
	if(ORP){
	 assert sv;
	}
	send rl;
	next_state .;
		v E_L_YOP_NGA 11
		if(NRP){
		 send D;
		 next_state .;
		} else {
		 send rl;
		 next_state .;
		}
			v E_L_YOP_NGA 11
			send D;
			next_state .;

v S_L_YOP_YGA 12
if(ii){
 send D;
 next_state .;
} else {
 if(t—il){
  send rl;
  next_state .;
 } else {
  send flcfr;
  next_state .;
 }}
	v S_L_YOP_YGA12
	if(ORP){
	 assert sv;
	}
	if(t){
	 send rl;
	} else {
	 send rflcfr;
	}
	next_state .;
		v S_L_YOP_YGA 12
		send rl;
		next_state .;
			v S_L_YOP_YGA 12
			send D;
			next_state .;

v E_L_YOP_YGA 13
if(ii){
 send D;
 next_state .;
} else {
 send rl;
 next_state .;
}
	v E_L_YOP_YGA13
	if(ORP){
	 assert sv;
	}
	send rl;
	next_state .;
		v E_L_YOP_YGA 13
		if(NRP){
		 send D;
		 next_state .;
		} else {
		 send rl;
		 next_state .;
		}
			v E_L_YOP_YGA 13
			send D;
			next_state .;

vw S_U_NOP_NGA 14
assert rV;
assert iW;
do +w;
next_state .;
	vw S_U_NOP_NGA14
	assert T;
	if(ORP){
	 assert sv;
	}
	send rflcfr;
	next_state .;
		vw S_U_NOP_NGA 14
		assert T;
		send rflcfr;
		next_state .;
			vw S_U_NOP_NGA 14
			assert T;
			send flcfw;
			next_state .;

FLCFR	RFLCFR	R	FLCFW

vwS_L_NOP_NGA 15
if(ii){
 send D;
 next_state .;
} else {
 assert rV;
 do +w;
 next_state .;
}
	vwS_L_NOP_NGA15
	assert T;
	if(ORP){
	 assert sv;
	}
	send rflcfr;
	next_state .;
		vwS_L_NOP_NGA 15
		assert T;
		send rflcfr;
		next_state .;
			vwS_L_NOP_NGA 15
			send D;
			next_state .;

vwS_L_YOP_NGA 16
if(ii){
 send D;
 next_state .;
} else {
 assert iW;
 do +w;
 next_state .;
}
	vwS_L_YOP_NGA16
	if(ORP){
	 assert sv;
	}
	if(t){
	 send rl;
	} else {
	 send rflcfr;
	}
	next_state .;
		vwS_L_YOP_NGA 16
		send rl;
		next_state .;
			vwS_L_YOP_NGA 16
			send D;
			next_state .;

vwE_L_YOP_NGA 17
if(ii){
 send D;
 next_state .;
} else {
 assert iW;
 do +w;
 next_state .;
}
	vwE_L_YOP_NGA17
	if(ORP){
	 assert sv;
	}
	send rl;
	next_state .;
		vwE_L_YOP_NGA 17
		if(NRP){
		 send D;
		 next_state .;
		} else {
		 send rl;
		 next_state .;
		}
			vwE_L_YOP_NGA 17
			send D;
			next_state .;

vwS_L_YOP_YGA 18
if(ii){
 send D;
 next_state .;
} else {
 assert iW;
 do +w;
 next_state .;
}
	vwS_L_YOP_YGA18
	if(ORP){
	 assert sv;
	}
	if(t){
	 send rl;
	} else {
	 send rflcfr;
	}
	next_state .;
		vwS_L_YOP_YGA 18
		send rl;
		next_state .;
			vwS_L_YOP_YGA 18
			send D;
			next_state .;

vwE_L_YOP_YGA 19
if(ii){
 send D;
 next_state .;
} else {
 assert iW;
 do +w;
 next_state .;
}
	vwE_L_YOP_YGA19
	if(ORP){
	 assert sv;
	}
	send rl;
	next_state .;
		vwE_L_YOP_YGA 19
		if(NRP){
		 send D;
		 next_state .;
		} else {
		 send rl;
		 next_state .;
		}
			vwE_L_YOP_YGA 19
			send D;
			next_state .;

FLCFR	RFLCFR	R	FLCFW

vc S_U_NOP_NGA 20
assert rV;
do +w;
next_state 27;
	vc S_U_NOP_NGA20
	if(ORP){
	 assert svc;
	}
	send r;
	next_state .;
		vc S_U_NOP_NGA 20
		send r;
		next_state .;
			vc S_U_NOP_NGA 20
			assert T;
			send flcfw;
			next_state .;

vc E_U_NOP_NGA 21
assert rV;
do +w;
next_state 28;
	vc E_U_NOP_NGA21
	if(ORP){
	 assert svc;
	}
	send r;
	next_state .;
		vc E_U_NOP_NGA 21
		do +s;
		send r;
		next_state 20;
			vc E_U_NOP_NGA 21
			do L;
			if(ii){
			 do +v;
			}
			send la;
			next_state 24;

vc S_L_NOP_NGA 22
if(ii){
 send D;
 next_state .;
} else {
 assert rC;
 do +v;
 send r;
 next_state .;
}
	vc S_L_NOP_NGA22
	if(ORP){
	 assert svc;
	}
	send r;
	next_state .;
		vc S_L_NOP_NGA 22
		send r;
		next_state .;
			vc S_L_NOP_NGA 22
			send D;
			next_state .;

vc S_L_YOP_NGA 23
if(ii){
 send D;
 next_state .;
} else {
 do +v;
 send r;
 next_state .;
}
	vc S_L_YOP_NGA23
	if(ORP){
	 assert svc;
	}
	send r;
	next_state .;
		vc S_L_YOP_NGA 23
		send r;
		next_state .;
			vc S_L_YOP_NGA 23
			send D;
			next_state .;

vc E_L_YOP_NGA 24
if(ii){
 send D;
 next_state .;
} else {
 do +v;
 send r;
 next_state .;
}
	vc E_L_YOP_NGA24
	if(ORP){
	 assert svc;
	}
	send r;
	next_state .;
		vc E_L_YOP_NGA 24
		if(NRP){
		 send D;
		 next_state .;
		} else {
		 send r;
		 next_state .;
		}
			vc E_L_YOP_NGA 24
			send D;
			next_state .;

FLCFR	RFLCFR	R	FLCFW

vc S_L_YOP_YGA 25
if(ii){
send D;
next_state .;
} else {
do +v;
send r;
next_state .;
}
	vc S_L_YOP_YGA25
	if(ORP){
	assert svc;
	}
	send r;
	next_state .;
		vc S_L_YOP_YGA 25
		send r;
		next_state .;
			vc S_L_YOP_YGA 25
			send D;
			next_state .;

vc E_L_YOP_YGA 26
if(ii){
send D;
next_state .;
} else {
do +v;
send r;
next_state .;
}
	vc E_L_YOP_YGA26
	if(ORP){
	assert svc;
	}
	send r;
	next_state .;
		vc E_L_YOP_YGA 26
		if(NRP){
		send D;
		next_state .;
		} else {
		send r;
		next_state .;
		}
			vc E_L_YOP_YGA 26
			send D;
			next_state .;

vwc S_U_NOP_NGA 27
assert rV;
assert iW;
do +w;
next_state .;
	vwc S_U_NOP_NGA27
	if(ORP){
	assert svc;
	}
	send r;
	next_state .;
		vwc S_U_NOP_NGA 27
		send r;
		next_state .;
			vwc S_U_NOP_NGA 27
			assert T;
			send flcfw;
			next_state .;

vwc E_U_NOP_NGA 28
assert rV;
assert iW;
do +w;
next_state .;
	vwc E_U_NOP_NGA28
	if(ORP){
	assert svc;
	}
	send r;
	next_state .;
		vwc E_U_NOP_NGA 28
		do +s;
		send r;
		next_state 27;
			vwc E_U_NOP_NGA 28
			do L;
			if(ii){
			do +v;
			}
			send la;
			next_state 31;

vwc S_L_NOP_NGA 29
if(ii){
send D;
next_state .;
} else {
assert rC;
do +v;
send r;
next_state .;
}
	vwc S_L_NOP_NGA29
	if(ORP){
	assert svc;
	}
	send r;
	next_state .;
		vwc S_L_NOP_NGA 29
		send r;
		next_state .;
			vwc S_L_NOP_NGA 29
			send D;
			next_state .;

FLCFR	RFLCFR	R	FLCFW
vwc S_L_YOP_NGA 30 if(ii){ send D; next_state .; } else { do +v; send r; next_state .; }	vwc S_L_YOP_NGA30 if(ORP){ assert svc; } send r; next_state .;	vwc S_L_YOP_NGA 30 send r; next_state .;	vwc S_L_YOP_NGA 30 send D; next_state .;
vwc E_L_YOP_NGA 31 if(ii){ send D; next_state .; } else { do +v; send r; next_state .; }	vwc E_L_YOP_NGA31 if(ORP){ assert svc; } send r; next_state .;	vwc E_L_YOP_NGA 31 if(NRP){ send D; next_state .; } else { send r; next_state .; }	vwc E_L_YOP_NGA 31 send D; next_state .;
vwc S_L_YOP_YGA 32 if(ii){ send D; next_state .; } else { do +v; send r; next_state .; }	vwc S_L_YOP_YGA32 if(ORP){ assert svc; } send r; next_state .;	vwc S_L_YOP_YGA 32 send r; next_state .;	vwc S_L_YOP_YGA 32 send D; next_state .;
vwc E_L_YOP_YGA 33 if(ii){ send D; next_state .; } else { do +v; send r; next_state .; }	vwc E_L_YOP_YGA33 if(ORP){ assert svc; } send r; next_state .;	vwc E_L_YOP_YGA 33 if(NRP){ send D; next_state .; } else { send r; next_state .; }	vwc E_L_YOP_YGA 33 send D; next_state .;

L	A	A1	TA

INVALID 0
assert T;
if(ORP){
do L;
send l;
next_state 4;
} else {
send a;
next_state .;
}
	INVALID 0
	next_state X;
		INVALID 0
		next_state X;
			INVALID 0
			next_state X;

c S_U_NOP_NGA 1
do L;
if(NRP){
next_state 3;
} else {
if(ii){
do +v;
next_state 23;
} else {
next_state 4;
}}
send la;
	c S_U_NOP_NGA 1
	next_state X;
		c S_U_NOP_NGA 1
		next_state X;
			c S_U_NOP_NGA 1
			assert ic;
			do +i;
			if(0c){
			assert T;
			send ta;
			next_state 0;
			} else {
			next_state .;
			}

c E_U_NOP_NGA 2
do L;
if(NRP){
next_state 3;
} else {
if(ii){
do +v;
next_state 23;
} else {
next_state 4;
}}
send la;
	c E_U_NOP_NGA 2
	next_state X;
		c E_U_NOP_NGA 2
		next_state X;
			c E_U_NOP_NGA 2
			assert ic;
			do +i;
			assert 0C;
			if(>1&1vwc){
			send cte;
			}
			next_state .;

c S_L_NOP_NGA 3
send D;
next_state .;
	c S_L_NOP_NGA 3
	assert ic;
	do +i;
	if(0c){
	assert T;
	send a;
	next_state 0;
	} else {
	next_state .;
	}
		c S_L_NOP_NGA 3
		next_state X;
			c S_L_NOP_NGA 3
			assert ic;
			do +v;
			if(0c){
			assert T;
			send uncv;
			next_state 9;
			} else {
			next_state 22;
			}

L	A	A1	TA

c S_L_YOP_NGA 4
send D;
next_state .;
	c S_L_YOP_NGA 4
	assert ic;
	do +i;
	assert 0C;
	next_state .;
		c S_L_YOP_NGA 4
		assert ic;
		do +c;
		if(1c){
		assert T;
		send a1;
		next_state 3;
		}else {
		next_state 6;
		}
			c S_L_YOP_NGA 4
			assert 1W;
			assert ic;
			do +v;
			if(0c){
			if(T){
			send uncv;
			}
			next_state 10;
			}else {
			next_state 23;
			}

c E_L_YOP_NGA 5
send D;
next_state .;
	c E_L_YOP_NGA 5
	assert ic;
	do +i;
	assert 0C;
	next_state .;
		c E_L_YOP_NGA 5
		assert ic;
		do +c;
		if(1c){
		send wo;
		next_state 2;
		}else {
		next_state 7;
		}
			c E_L_YOP_NGA 5
			assert 1W;
			assert ic;
			do +v;
			if(0c){
			next_state 11;
			}else {
			next_state 24;
			}

c S_L_YOP_YGA 6
send D;
next_state .;
	c S_L_YOP_YGA 6
	assert ic;
	do +i;
	if(1c){
	assert T;
	send a1;
	next_state 3;
	}else {
	next_state .;
	}
		c S_L_YOP_YGA 6
		next_state X;
			c S_L_YOP_YGA 6
			assert 1W;
			assert ic;
			do +v;
			if(0c){
			if(T){
			send uncv;
			}
			next_state 12;
			}else {
			next_state 25;
			}

c E_L_YOP_YGA 7
send D;
next_state .;
	c E_L_YOP_YGA 7
	assert ic;
	do +i;
	if(1c){
	send wo;
	next_state 2;
	}else {
	next_state .;
	}
		c E_L_YOP_YGA 7
		next_state X;
			c E_L_YOP_YGA 7
			assert 1W;
			assert ic;
			do +v;
			if(0c){
			next_state 13;
			}else {
			next_state 26;
			}

L	A	A1	TA

v s_u_nop_nga 8
do L;
if(nrp){
send la;
next_state 9;
}else {
if(ii){
do +v;
}
send la;
next_state 10;
}
	v s_u_nop_nga 8
	next_state X;
		v s_u_nop_nga 8
		next_state X;
			v s_u_nop_nga 8
			next_state X;

v s_l_nop_nga 9
send D;
next_state .;
	v s_l_nop_nga 9
	assert iv;
	do +i;
	if(0v){
	assert T;
	send a;
	next_state 0;
	}else {
	next_state .;
	}
		v s_l_nop_nga 9
		next_state X;
			v s_l_nop_nga 9
			next_state X;

v s_l_yop_nga 10
send D;
next_state .;
	v s_l_yop_nga 10
	assert iv;
	do +i;
	assert 0V;
	next_state .;
		v s_l_yop_nga 10
		assert iv;
		do +c;
		if(1vc){
		assert T;
		send a1;
		next_state 3;
		}else {
		next_state 25;
		}
			v s_l_yop_nga 10
			next_state X;

v e_l_yop_nga 11
send D;
next_state .;
	v e_l_yop_nga 11
	assert iv;
	do +i;
	assert 0V;
	next_state .;
		v e_l_yop_nga 11
		assert iv;
		do +c;
		if(1vc){
		send wo;
		next_state 2;
		}else {
		next_state 26;
		}
			v e_l_yop_nga 11
			next_state X;

L	A	A1	TA

v S_L_YOP_YGA 12
send D;
next_state .;
	v S_L_YOP_YGA 12
	assert iv;
	do +i ;
	if(1v){
	 assert T;
	 assert lv;
	 do +c;
	 send a1;
	 next_state 3;
	} else {
	 next_state .;
	}
		v S_L_YOP_YGA 12
		next_state X;
			v S_L_YOP_YGA 12
			next_state X;

v E_L_YOP_YGA 13
send D;
next_state .;
	v E_L_YOP_YGA 13
	assert iv;
	do +i ;
	if(1v){
	 assert lv;
	 do +c;
	 send wo;
	 next_state 2;
	} else {
	 next_state .;
	}
		v E_L_YOP_YGA 13
		next_state X;
			v E_L_YOP_YGA 13
			next_state X;

vwS_U_NOP_NGA 14
do L;
if(NRP){
 send la;
 next_state 15;
} else {
 if(ii){
  do +v;
 }
 send la;
 next_state 16;
}
	vwS_U_NOP_NGA 14
	next_state X;
		vwS_U_NOP_NGA 14
		next_state X;
			vwS_U_NOP_NGA 14
			next_state X;

vwS_L_NOP_NGA 15
send D;
next_state .;
	vwS_L_NOP_NGA 15
	assert iv;
	do +i ;
	assert 0V;
	next_state .;
		vwS_L_NOP_NGA 15
		next_state X;
			vwS_L_NOP_NGA 15
			next_state X;

vwS_L_YOP_NGA 16
send D;
next_state .;
	vwS_L_YOP_NGA 16
	assert iv;
	do +i ;
	assert 0V;
	next_state .;
		vwS_L_YOP_NGA 16
		assert iv;
		do +c;
		assert 0V;
		next_state 32;
			vwS_L_YOP_NGA 16
			next_state X;

L	A	A1	TA

vwE_L_YOP_NGA 17
send D;
next_state .;
	vwE_L_YOP_NGA 17
	assert iv;
	do +i;
	assert 0V;
	next_state .;
		vwE_L_YOP_NGA 17
		assert iv;
		do +c;
		assert 0V;
		next_state 33;
			vwE_L_YOP_NGA 17
			next_state X;

vwS_L_YOP_YGA 18
send D;
next_state .;
	vwS_L_YOP_YGA 18
	assert iv;
	do +i;
	assert 0V;
	next_state .;
		vwS_L_YOP_YGA 18
		next_state X;
			vwS_L_YOP_YGA 18
			next_state X;

vwE_L_YOP_YGA 19
send D;
next_state .;
	vwE_L_YOP_YGA 19
	assert iv;
	do +i;
	assert 0V;
	next_state .;
		vwE_L_YOP_YGA 19
		next_state X;
			vwE_L_YOP_YGA 19
			next_state X;

vc S_U_NOP_NGA 20
do L;
if(NRP){
 next_state 22;
}else {
 if(ii){
  do +v;
 }
 next_state 23;
}
send la;
	vc S_U_NOP_NGA 20
	next_state X;
		vc S_U_NOP_NGA 20
		next_state X;
			vc S_U_NOP_NGA 20
			assert ic;
			do +i;
			if(0c){
			 assert T;
			 send uncv;
			 next_state 8;
			}else {
			 next_state .;
			}

vc E_U_NOP_NGA 21
do L;
if(NRP){
 next_state 22;
}else {
 if(ii){
  do +v;
 }
 next_state 23;
}
send la;
	vc E_U_NOP_NGA 21
	next_state X;
		vc E_U_NOP_NGA 21
		next_state X;
			vc E_U_NOP_NGA 21
			assert ic;
			do +i;
			assert 0C;
			next_state .;

L	A	A1	TA

vc S_L_NOP_NGA 22
send D;
next_state .;
	vc S_L_NOP_NGA 22
	assert ivc;
	do +i;
	if(0v){
	 if(0c){
	  next_state X;
	 }else {
	  next_state 3;
	 }
	}else {
	 if(0c){
	  send uncv;
	  next_state 9;
	 }else {
	  next_state .;
	 }}
		vc S_L_NOP_NGA 22
		next_state X;
			vc S_L_NOP_NGA 22
			assert ivc;
			do +v;
			if(0c){
			 assert T;
			 send uncv;
			 next_state 9;
			}else {
			 next_state .;
			}

vc S_L_YOP_NGA 23
send D;
next_state .;
	vc S_L_YOP_NGA 23
	assert ivc;
	do +i;
	if(0v){
	 if(0c){
	  next_state X;
	 }else {
	  next_state 4;
	 }
	}else {
	 if(0c){
	  next_state 10;
	 }else {
	  next_state .;
	 }}
		vc S_L_YOP_NGA 23
		assert ivc;
		do +c;
		if(0v){
		 next_state 6;
		}else {
		 next_state 25;
		}
			vc S_L_YOP_NGA 23
			assert 1W;
			assert ivc;
			do +v;
			if(0c&T){
			 send uncv;
			}
			if(0c){
			 next_state 10;
			}else {
			 next_state .;
			}

vc E_L_YOP_NGA 24
send D;
next_state .;
	vc E_L_YOP_NGA 24
	assert ivc;
	do +i;
	if(0v){
	 if(0c){
	  next_state X;
	 }else {
	  next_state 5;
	 }
	}else {
	 if(0c){
	  next_state 11;
	 }else {
	  next_state .;
	 }}
		vc E_L_YOP_NGA 24
		assert ivc;
		do +c;
		if(1vc){
		 send wo;
		 next_state 2;
		}else {
		 if(0v){
		  next_state 7;
		 }else {
		  next_state 26;
		 }}
			vc E_L_YOP_NGA 24
			assert 1W;
			assert ivc;
			do +v;
			if(0c){
			 next_state 11;
			}else {
			 next_state .;
			}

L	A	A1	TA

vc S_L_YOP_YGA 25
send D;
next_state .;
	vc S_L_YOP_YGA 25
	assert ivc;
	do +i;
	if(1vc){
	assert T;
	if(1v){
	do +c;
	}
	send a1;
	next_state 3;
	}else {
	if(0v){
	next_state 6;
	}else {
	if(0c){
	next_state 12;
	}else {
	next_state .;
	}}}
		vc S_L_YOP_YGA 25
		next_state X;
			vc S_L_YOP_YGA 25
			assert 1W;
			assert ivc;
			do +v;
			if(0c&T){
			send uncv;
			}
			if(0c){
			next_state 12;
			}else {
			next_state .;
			}

vc E_L_YOP_YGA 26
send D;
next_state .;
	vc E_L_YOP_YGA 26
	assert ivc;
	do +i;
	if(1vc){
	if(1v){
	do +c;
	}
	send wo;
	next_state 2;
	}else {
	if(0v){
	next_state 7;
	}else {
	if(0c){
	next_state 13;
	}else {
	next_state .;
	}}}
		vc E_L_YOP_YGA 26
		next_state X;
			vc E_L_YOP_YGA 26
			assert 1W;
			assert ivc;
			do +v;
			if(0c){
			next_state 13;
			}else {
			next_state .;
			}

vwc S_U_NOP_NGA 27
do L;
if(NRP){
next_state 29;
}else {
if(ii){
do +v;
}
next_state 30;
}
send la;
	vwc S_U_NOP_NGA 27
	next_state X;
		vwc S_U_NOP_NGA 27
		next_state X;
			vwc S_U_NOP_NGA 27
			assert ic;
			do +i;
			if(0c){
			assert T;
			send uncv;
			next_state 14;
			}else {
			next_state .;
			}

L	A	A1	TA

vwc E_U_NOP_NGA 28
do L;
if(NRP){
next_state 29;
} else {
if(ii){
do +v;
}
next_state 30;
}
send la;
	vwc E_U_NOP_NGA 28
	next_state X;
		vwc E_U_NOP_NGA 28
		next_state X;
			vwc E_U_NOP_NGA 28
			assert ic;
			do +i;
			assert 0C;
			next_state .;

vwc S_L_NOP_NGA 29
send D;
next_state .;
	vwc S_L_NOP_NGA 29
	assert ivc;
	do +i;
	assert 0V;
	if(0c){
	send uncv;
	next_state 15;
	} else {
	next_state .;
	}
		vwc S_L_NOP_NGA 29
		next_state X;
			vwc S_L_NOP_NGA 29
			assert ivc;
			do +v;
			if(0c){
			assert T;
			send uncv;
			next_state 15;
			} else {
			next_state .;
			}

vwc S_L_YOP_NGA 30
send D;
next_state .;
	vwc S_L_YOP_NGA 30
	assert ivc;
	do +i;
	assert 0V;
	if(0c){
	next_state 16;
	} else {
	next_state .;
	}
		vwc S_L_YOP_NGA 30
		assert ivc;
		do +c;
		assert 0V;
		next_state 32;
			vwc S_L_YOP_NGA 30
			assert lW;
			assert ivc;
			do +v;
			if(0c&T){
			send uncv;
			}
			if(0c){
			next_state 16;
			} else {
			next_state .;
			}

vwc E_L_YOP_NGA 31
send D;
next_state .;
	vwc E_L_YOP_NGA 31
	assert ivc;
	do +i;
	assert 0V;
	if(0c){
	next_state 17;
	} else {
	next_state .;
	}
		vwc E_L_YOP_NGA 31
		assert ivc;
		do +c;
		assert 0V;
		next_state 33;
			vwc E_L_YOP_NGA 31
			assert lW;
			assert ivc;
			do +v;
			if(0c){
			next_state 17;
			} else {
			next_state .;
			}

L	A	A1	TA

vwc S_L_YOP_YGA 32
send D;
next_state .;
	vwc S_L_YOP_YGA 32
	assert ivc;
	do +i;
	assert 0V;
	if(0c){
	next_state 18;
	} else {
	next_state .;
	}
		vwc S_L_YOP_YGA 32
		next_state X;
			vwc S_L_YOP_YGA 32
			assert 1W;
			assert ivc;
			do +v;
			if(0c&T){
			send uncv;
			}
			if(0c){
			next_state 18;
			} else {
			next_state .;
			}

vwc E_L_YOP_YGA 33
send D;
next_state .;
	vwc E_L_YOP_YGA 33
	assert ivc;
	do +i;
	assert 0V;
	if(0c){
	next_state 19;
	} else {
	next_state .;
	}
		vwc E_L_YOP_YGA 33
		next_state X;
			vwc E_L_YOP_YGA 33
			assert 1W;
			assert ivc;
			do +v;
			if(0c){
			next_state 19;
			} else {
			next_state .;
			}

CTE	FLCFT	RFLCFT	CV

INVALID 0
next_state X;
	INVALID 0
	if(t){
	alloc;
	next_state 2;
	} else {
	send flcft;
	next_state .;
	}
		INVALID0
		assert T;
		send rflcft;
		next_state .;
			INVALID 0
			next_state X;

c S_U_NOP_NGA 1
do +c;
next_state 2;
	c S_U_NOP_NGA 1
	send rt;
	next_state .;
		c S_U_NOP_NGA1
		send rt;
		next_state .;
			c S_U_NOP_NGA 1
			assert ic;
			next_state .;

c E_U_NOP_NGA 2 next_state X;	c E_U_NOP_NGA 2 send rt; next_state .;	c E_U_NOP_NGA2 send rt; next_state .;	c E_U_NOP_NGA 2 assert ic; next_state .;
c S_L_NOP_NGA 3 next_state X;	c S_L_NOP_NGA 3 send D; next_state .;	c S_L_NOP_NGA3 send D; next_state .;	c S_L_NOP_NGA 3 assert ic; next_state .;
c S_L_YOP_NGA 4 next_state X;	c S_L_YOP_NGA 4 send D; next_state .;	c S_L_YOP_NGA4 send D; next_state .;	c S_L_YOP_NGA 4 assert ic; next_state .;
c E_L_YOP_NGA 5 next_state X;	c E_L_YOP_NGA 5 send D; next_state .;	c E_L_YOP_NGA5 send D; next_state .;	c E_L_YOP_NGA 5 assert ic; next_state .;
c S_L_YOP_YGA 6 next_state X;	c S_L_YOP_YGA 6 send D; next_state .;	c S_L_YOP_YGA6 send D; next_state .;	c S_L_YOP_YGA 6 assert ic; next_state .;
c E_L_YOP_YGA 7 next_state X;	c E_L_YOP_YGA 7 send D; next_state .;	c E_L_YOP_YGA7 send D; next_state .;	c E_L_YOP_YGA 7 assert ic; next_state .;

v S_U_NOP_NGA 8
next_state X;
	v S_U_NOP_NGA 8
	assert T;
	send flcft;
	next_state .;
		v S_U_NOP_NGA8
		assert T;
		send rflcft;
		next_state .;
			v S_U_NOP_NGA 8
			assert T;
			assert iv;
			do +c;
			send cv;
			if(0v){
			next_state 1;
			} else {
			next_state 20;
			}

v S_L_NOP_NGA 9
next_state X;
	v S_L_NOP_NGA 9
	send D;
	next_state .;
		v S_L_NOP_NGA9
		send D;
		next_state .;
			v S_L_NOP_NGA 9
			assert T;
			assert iv;
			do +c;
			send cv;
			if(0v){
			next_state 3;
			} else {
			next_state 22;
			}

CTE	FLCFT	RFLCFT	CV

v S_L_YOP_NGA 10
next_state X;
	v S_L_YOP_NGA 10
	send D;
	next_state .;
		v S_L_YOP_NGA10
		send D;
		next_state .;
			v S_L_YOP_NGA 10
			assert iv;
			do +c;
			if(T){
			send cv;
			}
			if(0v){
			next_state 4;
			}else {
			next_state 23;
			}

v E_L_YOP_NGA 11
next_state X;
	v E_L_YOP_NGA 11
	send D;
	next_state .;
		v E_L_YOP_NGA11
		send D;
		next_state .;
			v E_L_YOP_NGA 11
			assert iv;
			do +c;
			if(0v){
			next_state 5;
			}else {
			next_state 24;
			}

v S_L_YOP_YGA 12
next_state X;
	v S_L_YOP_YGA 12
	send D;
	next_state .;
		v S_L_YOP_YGA12
		send D;
		next_state .;
			v S_L_YOP_YGA 12
			assert iv;
			do +c;
			if(T){
			send cv;
			}
			if(0v){
			next_state 6;
			}else {
			next_state 25;
			}

v E_L_YOP_YGA 13
next_state X;
	v E_L_YOP_YGA 13
	send D;
	next_state .;
		v E_L_YOP_YGA13
		send D;
		next_state .;
			v E_L_YOP_YGA 13
			assert iv;
			do +c;
			if(0v){
			next_state 7;
			}else {
			next_state 26;
			}

vwS_U_NOP_NGA 14
next_state X;
	vwS_U_NOP_NGA 14
	assert T;
	send flcft;
	next_state .;
		vwS_U_NOP_NGA14
		assert T;
		send rflcft;
		next_state .;
			vwS_U_NOP_NGA 14
			assert T;
			assert iv;
			do +c;
			send rdaw, c;
			send cv;
			if(0v){
			next_state 1;
			}else {
			next_state 20;
			}

CTE	FLCFT	RFLCFT	CV

vws_L_NOP_NGA 15
next_state X;
	vws_L_NOP_NGA 15
	send D;
	next_state .;
		vws_L_NOP_NGA15
		send D;
		next_state .;
			vws_L_NOP_NGA 15
			assert T;
			assert iv;
			do +c;
			send rdaw, c;
			send cv;
			if(0v){
			next_state 3;
			} else {
			next_state 22;
			}

vws_L_YOP_NGA 16
next_state X;
	vws_L_YOP_NGA 16
	send D;
	next_state .;
		vws_L_YOP_NGA16
		send D;
		next_state .;
			vws_L_YOP_NGA 16
			assert iv;
			do +c;
			if(T){
			send cv;
			}
			send rdaw, c;
			if(0v){
			next_state 4;
			} else {
			next_state 23;
			}

vwE_L_YOP_NGA 17
next_state X;
	vwE_L_YOP_NGA 17
	send D;
	next_state .;
		vwE_L_YOP_NGA17
		send D;
		next_state .;
			vwE_L_YOP_NGA 17
			assert iv;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 5;
			} else {
			next_state 24;
			}

vws_L_YOP_YGA 18
next_state X;
	vws_L_YOP_YGA 18
	send D;
	next_state .;
		vws_L_YOP_YGA18
		send D;
		next_state .;
			vws_L_YOP_YGA 18
			assert iv;
			do +c;
			if(T){
			send cv;
			}
			send rdaw, c;
			if(0v){
			next_state 6;
			} else {
			next_state 25;
			}

CTE	FLCFT	RFLCFT	CV

vwE_L_YOP_YGA 19
next_state X;
	vwE_L_YOP_YGA 19
	send D;
	next_state .;
		vwE_L_YOP_YGA19
		send D;
		next_state .;
			vwE_L_YOP_YGA 19
			assert iv;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 7;
			} else {
			next_state 26;
			}

vc S_U_NOP_NGA 20
do +e;
next_state 21;
	vc S_U_NOP_NGA 20
	send rt;
	next_state .;
		vc S_U_NOP_NGA20
		send rt;
		next_state .;
			vc S_U_NOP_NGA 20
			assert ivc;
			do +c;
			if(0v){
			next_state 1;
			} else {
			next_state .;
			}

vc E_U_NOP_NGA 21
next_state X;
	vc E_U_NOP_NGA 21
	send rt;
	next_state .;
		vc E_U_NOP_NGA21
		send rt;
		next_state .;
			vc E_U_NOP_NGA 21
			assert ivc;
			do +c;
			if(0v){
			next_state 2;
			} else {
			next_state .;
			}

vc S_L_NOP_NGA 22
next_state X;
	vc S_L_NOP_NGA 22
	send D;
	next_state .;
		vc S_L_NOP_NGA22
		send D;
		next_state .;
			vc S_L_NOP_NGA 22
			assert ivc;
			do +c;
			if(0v){
			next_state 3;
			} else {
			next_state .;
			}

vc S_L_YOP_NGA 23
next_state X;
	vc S_L_YOP_NGA 23
	send D;
	next_state .;
		vc S_L_YOP_NGA23
		send D;
		next_state .;
			vc S_L_YOP_NGA 23
			assert ivc;
			do +c;
			if(0v){
			next_state 4;
			} else {
			next_state .;
			}

vc E_L_YOP_NGA 24
next_state X;
	vc E_L_YOP_NGA 24
	send D;
	next_state .;
		vc E_L_YOP_NGA24
		send D;
		next_state .;
			vc E_L_YOP_NGA 24
			assert ivc;
			do +c;
			if(0v){
			next_state 5;
			} else {
			next_state .;
			}

CTE	FLCFT	RFLCFT	CV

vc S_L_YOP_YGA 25
next_state X;
	vc S_L_YOP_YGA 25
	send D;
	next_state .;
		vc S_L_YOP_YGA25
		send D;
		next_state .;
			vc S_L_YOP_YGA 25
			assert ivc;
			do +c;
			if(T){
			send cv;
			}
			if(0v){
			next_state 6;
			} else {
			next_state .;
			}

vc E_L_YOP_YGA 26
next_state X;
	vc E_L_YOP_YGA 26
	send D;
	next_state .;
		vc E_L_YOP_YGA26
		send D;
		next_state .;
			vc E_L_YOP_YGA 26
			assert ivc;
			do +c;
			if(0v){
			next_state 7;
			} else {
			next_state .;
			}

vwc S_U_NOP_NGA 27
do +e;
next_state 28;
	vwc S_U_NOP_NGA 27
	send rt;
	next_state .;
		vwc S_U_NOP_NGA27
		send rt;
		next_state .;
			vwc S_U_NOP_NGA 27
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 1;
			} else {
			next_state 20;
			}

vwc E_U_NOP_NGA 28
next_state X;
	vwc E_U_NOP_NGA 28
	send rt;
	next_state .;
		vwc E_U_NOP_NGA28
		send rt;
		next_state .;
			vwc E_U_NOP_NGA 28
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 2;
			} else {
			next_state 21;
			}

vwc S_L_NOP_NGA 29
next_state X;
	vwc S_L_NOP_NGA 29
	send D;
	next_state .;
		vwc S_L_NOP_NGA29
		send D;
		next_state .;
			vwc S_L_NOP_NGA 29
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 3;
			} else {
			next_state 22;
			}

CTE	FLCFT	RFLCFT	CV

vwc S_L_YOP_NGA 30
next_state X;
	vwc S_L_YOP_NGA 30
	send D;
	next_state .;
		vwc S_L_YOP_NGA30
		send D;
		next_state .;
			vwc S_L_YOP_NGA 30
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 4;
			}else {
			next_state 23;
			}

vwc E_L_YOP_NGA 31
next_state X;
	vwc E_L_YOP_NGA 31
	send D;
	next_state .;
		vwc E_L_YOP_NGA31
		send D;
		next_state .;
			vwc E_L_YOP_NGA 31
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 5;
			}else {
			next_state 24;
			}

vwc S_L_YOP_YGA 32
next_state X;
	vwc S_L_YOP_YGA 32
	send D;
	next_state .;
		vwc S_L_YOP_YGA32
		send D;
		next_state .;
			vwc S_L_YOP_YGA 32
			assert ivc;
			do +c;
			send rdaw, c;
			if(T){
			send cv;
			}
			if(0v){
			next_state 6;
			}else {
			next_state 25;
			}

vwc E_L_YOP_YGA 33
next_state X;
	vwc E_L_YOP_YGA 33
	send D;
	next_state .;
		vwc E_L_YOP_YGA33
		send D;
		next_state .;
			vwc E_L_YOP_YGA 33
			assert ivc;
			do +c;
			send rdaw, c;
			if(0v){
			next_state 7;
			}else {
			next_state 26;
			}

RD	UNCV	RT	WO

INVALID 0
next_state X;
	INVALID 0
	next_state X;
		INVALID 0
		assert T;
		send rflcft;
		next_state .;
			INVALID 0
			next_state X;

c S_U_NOP_NGA 1
next_state .;
	c S_U_NOP_NGA 1
	assert ic;
	do +v;
	if(0c){
	 assert T;
	 send uncv;
	 next_state 8;
	} else {
	 next_state 20;
	}
		c S_U_NOP_NGA 1
		send rt;
		next_state .;
			c S_U_NOP_NGA 1
			next_state X;

c E_U_NOP_NGA 2
next_state X;
	c E_U_NOP_NGA 2
	assert ic;
	do +v;
	assert 0C;
	next_state 21;
		c E_U_NOP_NGA 2
		send rt;
		next_state .;
			c E_U_NOP_NGA 2
			next_state X;

c S_L_NOP_NGA 3
next_state .;
	c S_L_NOP_NGA 3
	assert ic;
	do +v;
	if(0c&T){
	 send uncv;
	}
	if(0c){
	 next_state 9;
	} else {
	 next_state 22;
	}
		c S_L_NOP_NGA 3
		send D;
		next_state .;
			c S_L_NOP_NGA 3
			send wo;
			next_state 2;

c S_L_YOP_NGA 4
next_state .;
	c S_L_YOP_NGA 4
	assert ic;
	do +v;
	if(0c&T){
	 send uncv;
	}
	if(0c){
	 next_state 10;
	} else {
	 next_state 23;
	}
		c S_L_YOP_NGA 4
		send D;
		next_state .;
			c S_L_YOP_NGA 4
			next_state X;

c E_L_YOP_NGA 5
next_state X;
	c E_L_YOP_NGA 5
	assert ic;
	do +v;
	if(0c){
	 next_state 11;
	} else {
	 next_state 24;
	}
		c E_L_YOP_NGA 5
		send D;
		next_state .;
			c E_L_YOP_NGA 5
			next_state X;

RD	UNCV	RT	WO

c S_L_YOP_YGA 6
next_state .;
	c S_L_YOP_YGA 6
	assert ic;
	do +v;
	if(0c&T){
	 send uncv;
	}
	if(0c){
	 next_state 12;
	}else {
	 next_state 25;
	}
		c S_L_YOP_YGA 6
		send D;
		next_state .;
			c S_L_YOP_YGA 6
			next_state X;

c E_L_YOP_YGA 7
next_state X;
	c E_L_YOP_YGA 7
	assert ic;
	do +v;
	if(0c){
	 next_state 13;
	}else {
	 next_state 26;
	}
		c E_L_YOP_YGA 7
		send D;
		next_state .;
			c E_L_YOP_YGA 7
			next_state X;

v S_U_NOP_NGA 8
send rdav, c;
next_state 1;
	v S_U_NOP_NGA 8
	next_state X;
		v S_U_NOP_NGA 8
		assert T;
		send rflcft;
		next_state .;
			v S_U_NOP_NGA 8
			next_state X;

v S_L_NOP_NGA 9
send rdav, c;
next_state 3;
	v S_L_NOP_NGA 9
	next_state X;
		v S_L_NOP_NGA 9
		send D;
		next_state .;
			v S_L_NOP_NGA 9
			next_state X;

v S_L_YOP_NGA 10
send rdavL, c;
if(0c){
 next_state .;
}else {
 next_state 23;
}
	v S_L_YOP_NGA 10
	next_state X;
		v S_L_YOP_NGA 10
		send D;
		next_state .;
			v S_L_YOP_NGA 10
			next_state X;

v E_L_YOP_NGA 11 next_state X;	v E_L_YOP_NGA 11 next_state X;	v E_L_YOP_NGA 11 send D; next_state .;	v E_L_YOP_NGA 11 next_state X;

v S_L_YOP_YGA 12
send rdavL, c;
if(0c){
 next_state .;
}else {
 next_state 25;
}
	v S_L_YOP_YGA 12
	next_state X;
		v S_L_YOP_YGA 12
		send D;
		next_state .;
			v S_L_YOP_YGA 12
			next_state X;

v E_L_YOP_YGA 13 next_state X;	v E_L_YOP_YGA 13 next_state X;	v E_L_YOP_YGA 13 send D; next_state .;	v E_L_YOP_YGA 13 next_state X;

RD	UNCV	RT	WO

vw S_U_NOP_NGA 14
send rdavw, c;
next_state 1;
	vw S_U_NOP_NGA 14
	next_state X;
		vw S_U_NOP_NGA 14
		assert T;
		send rflcft;
		next_state .;
			vw S_U_NOP_NGA 14
			next_state X;

vw S_L_NOP_NGA 15
send rdavw, c;
next_state 3;
	vw S_L_NOP_NGA 15
	next_state X;
		vw S_L_NOP_NGA 15
		send D;
		next_state .;
			vw S_L_NOP_NGA 15
			next_state X;

vw S_L_YOP_NGA 16
send rdavwL, c;
next_state 23;
	vw S_L_YOP_NGA 16
	next_state X;
		vw S_L_YOP_NGA 16
		send D;
		next_state .;
			vw S_L_YOP_NGA 16
			next_state X;

vw E_L_YOP_NGA 17 next_state X;	vw E_L_YOP_NGA 17 next_state X;	vw E_L_YOP_NGA 17 send D; next_state .;	vw E_L_YOP_NGA 17 next_state X;

vw S_L_YOP_YGA 18
send rdavwL, c;
next_state 25;
	vw S_L_YOP_YGA 18
	next_state X;
		vw S_L_YOP_YGA 18
		send D;
		next_state .;
			vw S_L_YOP_YGA 18
			next_state X;

vw E_L_YOP_YGA 19 next_state X;	vw E_L_YOP_YGA 19 next_state X;	vw E_L_YOP_YGA 19 send D; next_state .;	vw E_L_YOP_YGA 19 next_state X;

vc S_U_NOP_NGA 20
next_state .;
	vc S_U_NOP_NGA 20
	assert ic;
	do +v;
	if(0c){
	 assert T;
	 send uncv;
	 next_state 8;
	} else {
	 next_state .;
	}
		vc S_U_NOP_NGA 20
		send rt;
		next_state .;
			vc S_U_NOP_NGA 20
			next_state X;

vc E_U_NOP_NGA 21
next_state X;
	vc E_U_NOP_NGA 21
	assert ic;
	do +v;
	assert 0C;
	next_state .;
		vc E_U_NOP_NGA 21
		send rt;
		next_state .;
			vc E_U_NOP_NGA 21
			next_state X;

vc S_L_NOP_NGA 22
next_state .;
	vc S_L_NOP_NGA 22
	assert ic;
	do +v;
	if(0c&T){
	 send uncv;
	}
	if(0c){
	 next_state 9;
	} else {
	 next_state .;
	}
		vc S_L_NOP_NGA 22
		send D;
		next_state .;
			vc S_L_NOP_NGA 22
			next_state X;

RD	UNCV	RT	WO

vc S_L_YOP_NGA 23
next_state .;
	vc S_L_YOP_NGA 23
	assert ic;
	do +v;
	if(0c&T){
	send uncv;
	}
	if(0c){
	next_state 10;
	} else {
	next_state .;
	}
		vc S_L_YOP_NGA 23
		send D;
		next_state .;
			vc S_L_YOP_NGA 23
			next_state X;

vc E_L_YOP_NGA 24
next_state X;
	vc E_L_YOP_NGA 24
	assert ic;
	do +v;
	if(0c){
	next_state 11;
	} else {
	next_state .;
	}
		vc E_L_YOP_NGA 24
		send D;
		next_state .;
			vc E_L_YOP_NGA 24
			next_state X;

vc S_L_YOP_YGA 25
if(1c&lc){
send rdavL, c;
next_state 6;
} else {
next_state .;
}
	vc S_L_YOP_YGA 25
	assert ic;
	do +v;
	if(0c&T){
	send uncv;
	}
	if(0c){
	next_state 12;
	} else {
	next_state .;
	}
		vc S_L_YOP_YGA 25
		send D;
		next_state .;
			vc S_L_YOP_YGA 25
			next_state X;

vc E_L_YOP_YGA 26
next_state X;
	vc E_L_YOP_YGA 26
	assert ic;
	do +v;
	if(0c){
	next_state 13;
	} else {
	next_state .;
	}
		vc E_L_YOP_YGA 26
		send D;
		next_state .;
			vc E_L_YOP_YGA 26
			next_state X;

vwc S_U_NOP_NGA 27
next_state .;
	vwc S_U_NOP_NGA 27
	assert ic;
	do +v;
	if(0c){
	assert T;
	send uncv;
	next_state 14;
	} else {
	next_state .;
	}
		vwc S_U_NOP_NGA 27
		send rt;
		next_state .;
			vwc S_U_NOP_NGA 27
			next_state X;

RD	UNCV	RT	WO

vwc E_U_NOP_NGA 28
next_state X;
	vwc E_U_NOP_NGA 28
	assert ic;
	do +v;
	assert 0C;
	next_state .;
		vwc E_U_NOP_NGA 28
		send rt;
		next_state .;
			vwc E_U_NOP_NGA 28
			next_state X;

vwc S_L_NOP_NGA 29
next_state .;
	vwc S_L_NOP_NGA 29
	assert ic;
	do +v;
	if(0c&T){
	send uncv;
	}
	if(0c){
	next_state 15;
	} else {
	next_state .;
	}
		vwc S_L_NOP_NGA 29
		send D;
		next_state .;
			vwc S_L_NOP_NGA 29
			next_state X;

vwc S_L_YOP_NGA 30
next_state .;
	vwc S_L_YOP_NGA 30
	assert ic;
	do +v;
	if(0c&T){
	send uncv;
	}
	if(0c){
	next_state 16;
	} else {
	next_state .;
	}
		vwc S_L_YOP_NGA 30
		send D;
		next_state .;
			vwc S_L_YOP_NGA 30
			next_state X;

vwc E_L_YOP_NGA 31
next_state X;
	vwc E_L_YOP_NGA 31
	assert ic;
	do +v;
	if(0c){
	next_state 17;
	} else {
	next_state .;
	}
		vwc E_L_YOP_NGA 31
		send D;
		next_state .;
			vwc E_L_YOP_NGA 31
			next_state X;

vwc S_L_YOP_YGA 32
if(1c&lc){
send rdavwL, c;
next_state 6;
} else {
next_state .;
}
	vwc S_L_YOP_YGA 32
	assert ic;
	do +v;
	if(0c&T){
	send uncv;
	}
	if(0c){
	next_state 18;
	} else {
	next_state .;
	}
		vwc S_L_YOP_YGA 32
		send D;
		next_state .;
			vwc S_L_YOP_YGA 32
			next_state X;

RD	UNCV	RT	WO

vwc E_L_YOP_YGA 33
next_state X;
	vwc E_L_YOP_YGA 33
	assert ic;
	do +v;
	if(0c){
	next_state 19;
	} else {
	next_state .;
	}
		vwc E_L_YOP_YGA 33
		send D;
		next_state .;
			vwc E_L_YOP_YGA 33
			next_state X;

Bibliography

[1] Anant Agarwal. A locality-based multiprocessor cache interference model. VLSI Memo MIT VLSI Memo 89-565, Massachusetts Institute of Technology, 1989.

[2] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. Technical Report MIT/LCS/TM 454, MIT Laboratory for Computer Science, June 1991.

[4] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*. IEEE, June 1988.

[5] J. K. Archibald. A cache coherence approach for a large multiprocessor system. In *1988 International Conference on Supercomputing*, pages 337–345. ACM, July 1988.

[6] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[7] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[8] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, June 1990.

[9] William J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G. X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[10] S. J. Frank. Tightly coupled multiprocessor system speeds up memory access times. *Electronics*, 57(1):164–169, January 1984.

[11] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[12] Seif Haridi and Erik Hagersten. The cache coherence protocol of the Data Diffusion Machine. In *PARLE '89 Parallel Architectures and Languages Europe*, volume I, pages 1–18, June 1989.

[13] Kirk L. Johnson. The impact of communication locality on large-scale multiprocessor performance. In *19th Annual International Symposium on Computer Architecture*, pages 392–402, May 1992.

[14] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[15] Scott T. Leutenegger and Mary K. Vernon. A mean-value performance analysis of a new multiprocessor architecture. *Performance Evaluation Review*, 16(1):167–176, May 1988.

[16] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, Yale University, September 1986.

[17] Kai Li. IVY: A shared virtual memory system for parallel computing. In *International Conference on Parallel Computing*, pages 94–101, 1988.

[18] Yeong-Chang Maa, Dhiraj K. Pradhan, and Dominique Thiebaut. Two economical directory schemes for large-scale cache coherent multiprocessors. *Computer Architecture News*, 19(4), September 1991.

[19] Yeong-Chang Maa, Dhiraj K. Pradhan, and Dominique Thiebaut. A hierarchical directory scheme for large-scale cache coherent multiprocessors. In *Proceedings of the 6th International Symposium on Parallel Processing*, pages 43–46, May 1992.

[20] J. Mellor-Crummey and M Scott. Synchronization without contention. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[21] Madhavan Parthasarathy. Implementing shared memory on large-scale multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign, August 1992.

[22] Isaac D. Scherson and Peter F. Corbett. Communications overhead and the expected speedup of multidimensional mesh-connected parallel processors. *Journal of Parallel and Distributed Computing*, 1991.

[23] Steven Scott. A cache coherence program for scalable, shared-memory multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 49–59, April 1991.

[24] Steven L. Scott and James R. Goodman. Performance of pruning-cache directories for large-scale multiprocessors. To appear in IEEE Transactions on Parallel and Distributed Systems, 1992.

[25] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[26] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings AFIPS National Computer Conference*, pages 749–753, 1976.

[27] Brian Totty. An operating environment for the jellybean machine. SB Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 1988.

[28] Mary Vernon, Rajeev Jōg, and Gurindar S. Sohi. Performance analysis of hierarchical cache-consistent multiprocessors. In T. Hasegawa, H. Takagi, and Y. Takahashi, editors, *Performance of Distributed and Parallel Systems*, pages 111–126. IFIP, Elsevier Science Publishers, 1989.

[29] Deborah A. Wallach. A scalable hierarchical cache coherence protocol. SB Thesis, MIT, May 1990.

[30] A. W. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessors. In *14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.

[31] Qing Yang. Performance analysis of a cache-coherent multiprocessor based on hierarchical multiple buses. In N. Rishe, S. Navathe, and D. Tal, editors, *PARBASE-90 International Conference on Databases, Parallel Architectures and Their Applications*, pages 248–257. IEEE Computer Society Press, 1990.

[32] Qing Yang, G. Thangadurai, and Laxmi N. Bhuyan. An adaptive cache coherence scheme for hierarchical shared-memory multiprocessors. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 318–325. IEEE Computer Society Press, December 1990.