Optimizing Directory-Based Cache Coherence on the RAW Architecture

by

Satish Ramaswamy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2005

Certified by..
Anant Agarwal
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Optimizing Directory-Based Cache Coherence on the RAW Architecture

by

Satish Ramaswamy

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Caches help reduce the effect of long-latency memory requests, by providing a high speed data-path to local memory. However, in multi-processor systems utilizing shared memory, cache coherence protocols are necessary to ensure sequential consistency. Of the multiple coherence protocols developed, the scalability of directory-based schemes makes them ideal for RAW's architecture [1]. Although one such system has been demonstrated as a proof-of-concept, it lacks the ability to meet the requirements of load-intensive, high performance applications. It further provides the application developer with no programming constructs to easily leverage the system. This thesis further develops shared memory support for RAW, by bringing greater programmability and performance to shared memory applications. In doing so, it reveals that shared memory is a practical programming paradigm for developing parallel applications on the RAW architecture.

Thesis Supervisor: Anant Agarwal
Title: Professor

Acknowledgments

I would like to begin by thanking Anant Agarwal, my supervisor, for introducing me to the RAW group and providing me invaluable advice while taking on this ambitious project.

James Psota, Michael Taylor, and Paul Johnson deserve a huge amount of recognition. James served as a mentor throughout the duration of my research, and helped focus my energy towards critical areas of the design. Michael was invaluable in answering questions pertaining to the RAW architecture, and providing crucial debugging assistance when problems in BTL were uncovered. And Paul was solely responsible for engineering the binary rewriter.

Then onto my friends, for showing me what life's really about. Timmie "40-hands" Hong, for his inability to operate fire extinguishers. Justin, for his ill-timed, yet quite sobering head-butts. Pedro, for his 80's bond-trader mentality. I'll see you on the buy-side. Shiva, for his unbridled tamilian-bred temper. Ishan, Chris, and Ram, for being stellar examples of studious grad students. New York City awaits us. Shyam, for our continued vagrancies through fine establishments such as IQ and Burrito Max. Neal and Adam, for aspiring to be the Hans and Frans of the Z-center. Then there's the notorious HP crew... Chris, for acquainting me with Jack, Johnny, and Jose. You're always a game of pool, and a brew away. Ben, for OBX. Gabe, for his tactical training in bottle-rockets deployment. Arun, for quite possibly being the chillest bro around. And who could forget my parents? For all the worrying and heartache I've given them over the past 23 years, it looks like i might just turn out ok.

My research was partly supported by the MIT Department of Electrical Engineering & Computer Science.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shared memory and message-passing are two popular models that have emerged in distributed computing. The message-passing model relies on a direct communication channel between processing nodes, while shared memory architectures rely on a global memory store that can be read from and written to by all nodes. Historically, architectures with independent processing modules have implemented a subset of features from both of these models. However, these features have been fixed at design time, limiting the architecture's flexibility [5].

With the introduction of the RAW architecture, low-level interconnects have been exposed to software, resulting in greater flexibility. Since the underlying network can be reconfigured for various applications, various parallel processing models can be implemented and reconfigured on the RAW fabric. A message-passing infrastructure has already been developed [6], and a prototype shared memory system was recently explored [5].

In a shared-memory multiprocessor, the purpose of the memory system is to provide access to the data to be processed. However, the uneven growth in processor and memory technologies has led to a significant delta in their relative speeds. Since memory bandwidth has become a critical resource, local high-speed memories have been provided on-chip, for the purpose of replicating frequently accessed data [2]. This memory hierarchy relies on the spatial and temporal locality of memory accesses.

However, the presence of local caches in shared-memory multiprocessor systems introduces the cache coherence problem. When a processor modifies data in its local cache, its changes will not necessarily be reflected in (1) main memory and (2) remote caches. Thus, different processing nodes will have different views of memory, leading to incoherence. It is therefore necessary for shared-memory multiprocessors to have a system that ensures coherence.

This thesis builds upon a prototype shared memory system previously developed for the RAW architecture [5]. The low-level system was developed as a proof-of-concept for implementing cache coherence on RAW, and was therefore designed with disregard to programmability and performance. It was also plagued by numerous problems that hindered the proper execution of shared memory applications. This research serves as a second attempt to address the issue of cache coherence on RAW, and fixes various problems along with incorporating several optimizations. In addition, higher-level shared memory support is provided through the introduction of new shared memory primitives.

First, I describe the RAW architecture along with the design of this low-level cache coherence system. In subsequent chapters, I detail the various problems in the system, as well as the necessary measures taken to resolve them. I then go on to describe the systematic implementation of various performance optimizations.

The final chapters outline the development of higher-level shared memory support, via the introduction of new shared memory primitives along with a stand-alone shared memory controller. I first describe the higher-level architecture, along with a proof of its correctness. Next, I describe the development of a suite of shared memory applications specifically designed for the system. After profiling the performance of these applications, I conclude that shared memory is a now a practical programming paradigm on RAW.

1.0.1 Contributions

Since this thesis is an extension of the existing low-level cache coherence architecture developed in [5], it is prudent to outline the exact contributions made by this research:

1. Creating a fully functional low-level cache coherence system.

2. Developing performance enhancing optimizations.

3. Providing higher-level shared memory support, through the creation of a library of shared memory primitives, along with the introduction of a stand-alone shared memory controller.

4. Developing a suite of shared memory applications specific to RAW, for the purpose of profiling shared memory application performance.

1.1 Shared Memory Systems

Shared memory and message-passing are two common communication paradigms for multiprocessor systems [5]. Applications on the latter architecture are often more readily optimized, since the points of communication are explicitly declared. However, the shared memory paradigm may often be more attractive, due to the model's ease of programmability.

Shared memory systems utilize an interconnection network that provides processing modules access to main memory. This memory may be a single homogenous block, or it may be physically distributed as it is in the Alewife architecture [3]. Several commercially distributed architectures utilize bus-based memory systems. Buses simplify the process of ensuring cache coherence, since all processers can observe ongoing memory transactions through snoopy mechanisms, and take the corresponding actions to maintain coherence [1]. However, this architecture fails to scale well, since it lacks the bandwidth to support an increasing number of processors. Thus, large multi-processor systems utilize point-to-point connections between processors, so as to provide a high-bandwidth, low-latency path to memory [2]. Since these systems fail to have broadcast mechanisms, they are well suited for directory-based cache coherence mechanisms. The next section provides a formal description of the coherency problem, before introducing directory-based coherence systems.

1.1.1 Cache Coherence & Sequential Consistency

The notion of a hierarchical memory system was first introduced by J.S. Liptay of IBM in 1968. When this system was extended to shared-memory multiprocessor systems, the problem of cache coherence arose, since each processor's view of memory may vary. A memory is defined as coherent if:

1. A read to a location, following a write to the very same location by the same processor, results in the newly written value being read (provided no other processor writes to this location in between).

2. A read to a location, following a write to the very same location by a different processor, results in the newly written value being read, provided the read and write are sufficiently separated in time and no other writes occur in between.

3. Writes to the same location are serialized; two writes to the same location by any two processors are seen in the same order from every other processors perspective [4].

Shared memory systems further provide "sequential consistency". This is defined as "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [4]." That is, all processors must observe the same sequence of reads and writes, and the individual accesses of a particular processor must appear in the order dictated by its program flow.

To demonstrate one such coherence issue, consider two processors P1 and P2 that both access address A. Address A in main memory holds the data value X, so when both processors read A, they locally cache this value. Now, suppose P1 writes the value Y to address A. In a write-back cache, this modification only occurs locally, and is not propagated to main memory. At this point, P1 views address A as having the value Y, while P2 views address A as having the value X. Thus, the system is

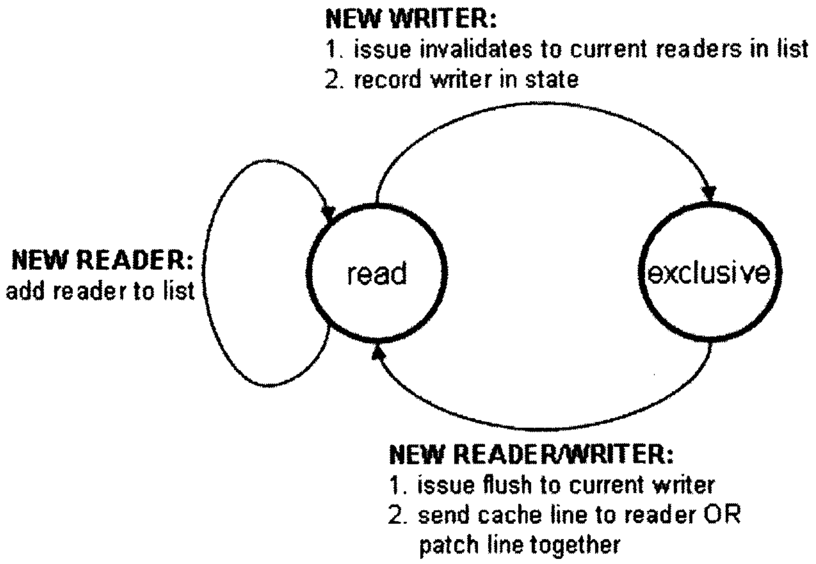

NEW WRITER:
1. issue invalidates to current readers in list
2. record writer in state
NEW READER:
add reader to list
read
exclusive
NEW READER/WRITER:
1. issue flush to current writer
2. send cache line to reader OR
patch line together

Figure 1-1: Basic Transition Diagram for Cache Coherence

incoherent, and some resolution must occur. The following section details one such system that is particular prevalent on modern large-scale multiprocessors.

1.2 Directory-Based Cache Coherence

In a directory-based cache coherence system, a central or distributed directory structure stores the state associated with each cache line. The two basic states are "exclusive" and "shared". The exclusive state is assigned to a cache-line that is owned by a particular cache; this owner has the most up-to-date version of the line, and the memory's version is in-fact stale. The shared state denotes that one or more caches have a clean copy of the memory location. The responsibility of a directory is to maintain cache coherence by issuing "writeback" and "invalidate" requests to processors whenever necessary. It achieves this by intercepting read & write requests to cache lines, issuing the appropriate requests, and updating the state. The following list outlines the basic coherence protocol, which is depicted in Figure 1-1 as well.

1. If a line is shared, a new reader results in the processor being added to the shared list.

2. If a line is shared, a new writer results in an invalidate request to the list of readers, and the line being set to exclusive on the writer.

3. If a line is exclusive, a new reader results in a flush request being sent to the writer, and the line being set to shared.

4. If a line is exclusive, a new writer results in a flush request being sent to the writer, and the line being patched together in main memory [1][4].

Archibald and Bauer first proposed a directory-based architecture that was implemented entirely in hardware. More recent architectures, such as MIT's ALEWIFE [3], have attempted to ensure coherence through both hardware and dedicated system-level software. In these architectures, the hardware handles coherence among a small subset of caches, while the software extends the capability of the system by providing support for extenuating cases. The directory-based prototype system developed for RAW is unique, in that it attempts to ensure coherence strictly through system-level software. The system is designed as a full-map directory based system, reminiscent of the system articulated by Censier and Feautrier [2]. The next section provides a brief overview of the RAW architecture.

1.3 RAW Architecture

The RAW microprocessor architecture was designed to efficiently expose the ever expanding computational resources available on silicon. Rather than improving on the concepts employed by modern super-scalar processors, the RAW architecture seeks to expand the domain of applications relevant to microprocessors. Specifically, it was designed to address data-intensive applications with fine grain parallelism (e.g. speech and video processing). Thus, the architecture is uniquely characterized by the high priority given to I/O processing, its aggressive scalability, and its parallel interface. Further, the basic underlying architecture is completely exposed to software, for the purpose of aggressively utilizing the hardware [8].

[1]Patching is necessary when distinct processors write to distinct words of the same cache line. The line needs to be reconstructed from these individual modifications

RAW is implemented as a 2-dimensional 4x4 mesh of 32-bit MIPS processors on a single die. Each edge of the die has 32 I/O pins, connected to a total of 6 Xilinx II FPGAs that interface with a variety of hardware devices including DRAM. The processors themselves are interconnected through two static networks, and two dynamic networks. The static network allows words to be routed between nearest neighbors, and is run by an on-chip switch processor. The dynamic networks are dimension ordered networks that require the use of headers to specify the length and destination of the message to be routed. The General Dynamic Network (GDN) [8].

Chapter 2

Original System

The RAW hardware is ideal for implementing cache coherence, because the architecture exposes low-level hardware mechanisms to software. By exploiting the precise level of control governing the processor and interconnect networks, traditional functionality previous implemented in hardware, can now be dictated by software. Thus, directory-based cache coherence is an ideal system to be developed on the architecture.

The RAW handheld board consists of a 16-tile RAW processor, surrounded by customizable off-chip FPGA's. These FPGA's interface with a variety of hardware devices including DRAM, and they require some modification when implementing the directory-based system. This chapter first details the RAW handheld board, and then goes on to describe the first-pass directory-based cache coherence system developed in [5].

2.1 RAW Hardware Overview

The RAW processor consists of a 2-D mesh of identical tiles. These tiles are interconnected to their nearest neighbors via static and dynamic networks, and routing is done via local static switches and dynamic routers. The tile processor uses a 32-bit MIPS instruction set, and all words, along with the width of the networks, are 32-bits as well.

2.1.1 Dynamic Networks

The dynamic networks on the RAW architecture are dimension-ordered packet routed networks. They consist of the "General Dynamic Network" (GDN) and the "Memory Dynamic Network" (MDN). The former is completely operated under the discretion of the user, while the latter is utilized by the RAW hardware for the purpose of interfacing with the memory controller.

Messages on these networks consist of a header, followed by any number of words. The header specifies the length of the message, its destination, a final routing field, and a 4-bit user field. The final routing field dictates the final direction the message should be routed *off-chip*, when it arrives at the destination tile.

When a message is issued from a particular tile, the dynamic routers first route it vertically, and then horizontally. After the message has arrived at its destination, the dynamic router at the destination strips the header, and places the message on an input buffer.

Access to the dynamic networks is provided through a register mapped interface. When a user wishes to send a message via the GDN, they must first write the header to register `$cgno`, followed by a write of the actual data words. The processor may stall on this write, if the outgoing hardware buffers are infact full. Receiving messages from the GDN requires a read from register `$cgni`; in this case, the processor will stall until a message arrives. It is important to note that stalling reads and writes prevent the tile from handling external interrupts.

The special-purpose registers `GDN_BUF` and `MDN_BUF` allow the user to determine the number of messages on the input buffer of the tile. In this manner, the user has the ability to engineer *non-blocking* reads from the dynamic networks by continually polling these registers.

2.1.2 Raw Processor Functionality

Each tile on the RAW chip resembles a MIPS R4000, and its specification is detailed in [8]. This thesis only examines three aspects of the tile's functionality: (1) cache misses (2) event counters and (3) interrupts.

Cache Misses

A cache miss can be generated by a load-word (LW) or store-word (SW) operation. When a miss occurs, the hardware generates an MDN message that is routed to the FPGA responsible for handling the specific bank of DRAM the address resides in. The FPGA then forwards the 8-word cache line to the requesting tile via the MDN. It is important to note that a tile cannot service any external interrupts while it is stalling for a cache line.

Interrupts

The RAW hardware allows tiles to be interrupted via special MDN messages that can either be generated off-chip or by another tile. To generate such a message, the user-bits of the MDN header must be set to `0b1111`; when the header arrives at its destination, the tile will jump to the appropriate handler if the "External Interrupt" bit is set in the `EX_BITS` special register. Only one outstanding interrupt per tile is allowed.

Event Counters

Tiles have various event counters that decrement every time a particular event occurs. When a particular counter transitions from 0 to -1, it asserts a "trigger" that holds its value until the user writes a new value into the counter. The triggers for the various event counters are OR'd together to form a single bit, which dictates when an event counter (EC) interrupt should fire.

When a trigger is asserted, the PC of the instruction causing the event is latched into bits [31:16] of the counter. It takes 6 cycles before an event counter interrupt will actually fire, and in this window of time, subsequent events will still be captured in the counter. This thesis is only concerned with the event counter that captures clean$\rightarrow$dirty cache line transitions.

2.2 Implementation of Directory-Based Cache Coherence

The first-pass directory-based cache coherence system for the 4x4 RAW architecture utilized the 4 rightmost tiles as system tiles. The purpose of each system-tile is twofold; it serves both as a directory and as an interrupt controller. The directory is responsible for managing a specific set of shared addresses by maintaining the state of each address in a directory table, and issuing invalidate and writeback requests whenever necessary. The purpose of interrupt controllers is to facilitate communication between user-tiles and directories; since tiles have no mechanism by which to detect the source of interrupts, they contact a pre-determined interrupt controller as part of the communication protocol [5].

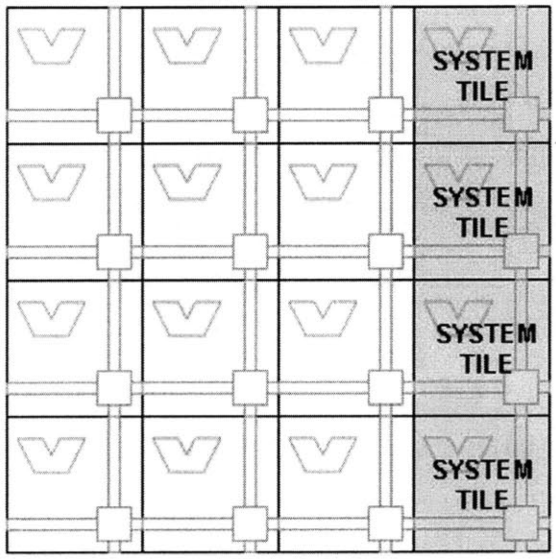
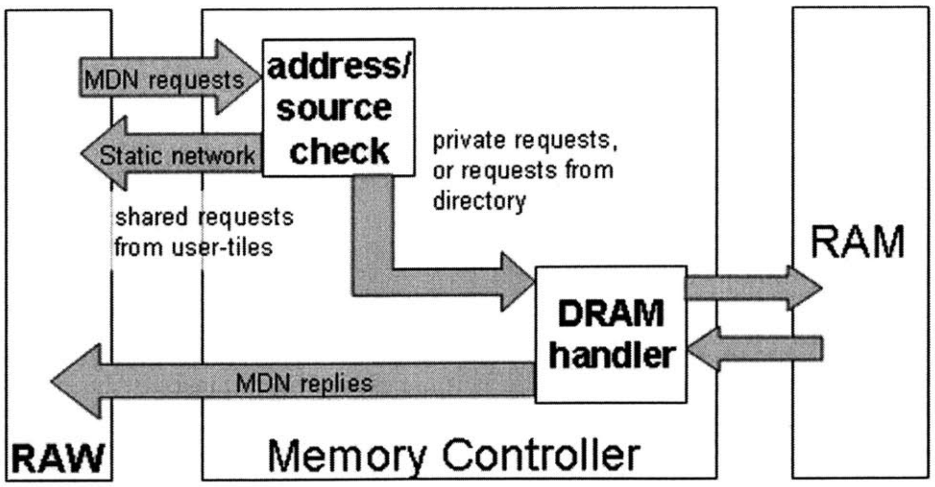


Figure 2-1: Layout of 4x4 RAW grid

2.2.1 Memory Controller

Along the east perimeter of the RAW chip, two FPGA's serve as an interface to four banks of DRAM. Memory load-word (LW) and store-word (SW) requests are issued from tiles in the form of MDN messages, and the two memory controllers are



[h]

Figure 2-2: Memory Controller Functionality

responsible for forwarding the requests along to DRAM. LW requests result in 8 words of data being retrieved, as the cache block size is exactly this width [8].

For the purpose of implementing cache coherence, the two memory controllers were modified to bounce shared memory requests to directory tiles, as depicted in Figure 2-2. In this manner, directories are able to observe all ongoing shared memory transactions. An address is declared shared if bits 25-26 are set; when this condition is satisfied, the memory controller pushes the request onto the static network for the appropriate directory-tile, determined by bits 31-30 of the address [5]. Figure 2-3 depicts how a shared address is decomposed into its constituent bits.

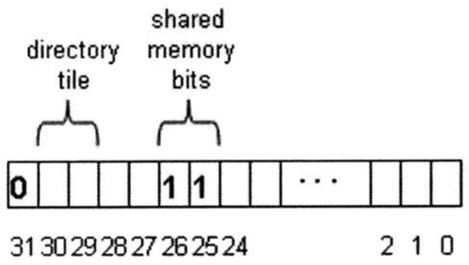


Figure 2-3: Decomposition of a Shared Address

2.2.2 Interrupt Controller Functionality

The interrupt controller is responsible for facilitating communication between user-tiles and directories. Each of the four controllers manages the three user-tiles residing on its physical row, and each tile is associated with a specific state-machine that determines when it is probably safe to interrupt it. The state-machines consist of the three states "not busy", "pinged", and "busy", as shown in figure 2.2.5. When a directory wishes to interrupt a particular tile, it must issue the request to the tile's corresponding interrupt controller.

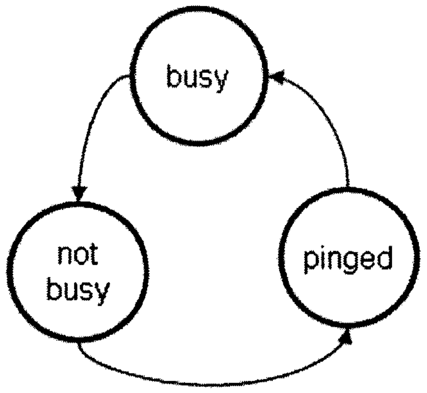


Figure 2-4: Interrupt Controller State Machine for a Particular Tile

If the state is "not busy", then the user-tile can be safely interrupted. Thus, when an interrupt is requested by a directory-tile, the controller issues the interrupt, sets the tile-state to "pinged", and records an associated timestamp. When the user-tile responds to the interrupt, the controller advances the state to "busy", and the communication protocol depicted in Figure 2-5 ensues. At the end of this protocol, the interrupted tile sends an acknowledgement to the interrupt controller, at which point the state is reset to "not busy". If a directory-tile requests an interrupt when the state is "pinged" or "busy", then the interrupt is delayed until the state resets to "not-busy".

Deadlock Recovery Handler

The cache coherence system has been proved to deadlock when a circular dependency exists between two or more tiles. These dependencies arise because MDN interrupts cannot be serviced while a tile is stalling on a memory read. In the simplest case of two tiles in deadlock, both tiles are stalling, waiting for cache lines that the other has exclusive access to. An example scenario where deadlock may arise is given below:

1. Tile X is exclusive on A1
2. Tile Y is exclusive on A2
3. Tile X reads A2, Tile Y reads A1
4. MDN interrupt sent to X for a writeback on A1
5. MDN interrupt sent to Y for a writeback on A2

A system-tile declares deadlock when a user-tile has not responded to an MDN interrupt after approximately 20,000 cycles. When this has occurred, a false cache-line is sent to the tile, so that the circular dependency is broken [1]. The following sequence of events occurs in the MDN interrupt after the spoof data has been received:

1. The tile rewinds `EX_PC` and searches for the offending load-word or store-word instruction. It then extracts the address and invalidates it.
2. The pending requests (writeback, invalidate) are immediately serviced.
3. The tile jumps to the PC of the offending LW/SW instruction, and re-executes it.

2.2.3 MDN Interrupts: Directory-to-User Communication

The following section details the communication protocol employed between directories and user tiles. For directory to user-tile communication, the GDN network is

[1]Recall that the pending MDN interrupt will fire after the line is received, since the tile is no longer stalling

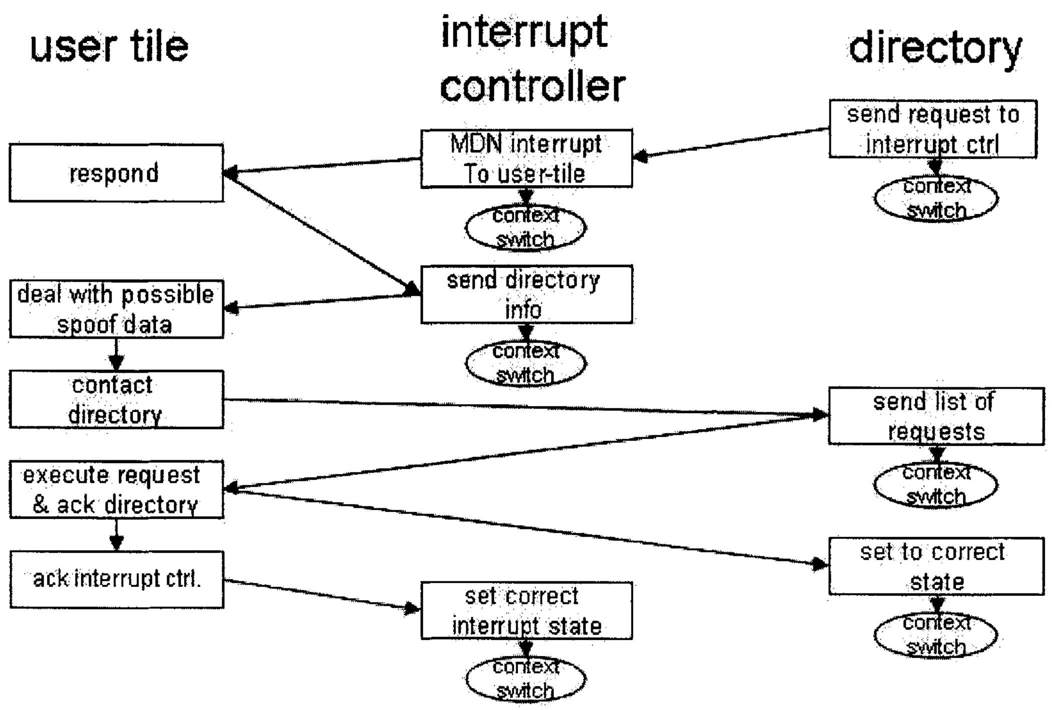


Figure 2-5: Directory to User Tile Communication

heavily utilized, and the entire protocol consists of a lengthy handshaking process as depicted in Figure 2-5. It is initiated by a directory issuing a GDN message to the appropriate interrupt controller of the tile to be interrupted.

After the user-tile has responded to its interrupt controller, the controller sends some information pertaining to deadlock, as well as a list of directories that it needs to contact. If necessary, the user-tile takes the appropriate measures to address any deadlock situation before proceeding. The user-tile next contacts each directory-tile sequentially, and receives a list of requests which it must address. A request consist of a shared memory address, along with a "writeback" or "invalidate" opcode. Once the tile has finished servicing a particular directory, it acknowledges the directory tile. And once the tile has serviced all directories, it acknowledges the interrupt controller.

2.2.4 Event Counter Interrupts: User-to-Directory Communication

Communication initiated by the user-tile is relatively straightforward, and occurs when a cache line is dirtied. To detect dirtied cache lines, user-tiles have an Event Counter (EC) interrupt configured to fire on every clean-to-dirty transition. The EC handler then extracts the address and data from the offending store-word instruction, and sends it to the appropriate directory tile via the GDN. The handler subsequently stalls until an acknowledgement is received from the directory, which can be one of the following: "flush & invalidate", "invalidate", or "nothing". Once this acknowledgement is received, the user-tile exits the handler and resumes execution of user-code. For the purpose of improving performance, The EC handler explicitly enables interrupts during the period of time in which it is waiting for a response from a directory. Thus, an MDN interrupt can be handled while already inside the EC handler.

2.2.5 System Tile Implementation

A system-tile needs to emulate both directory and interrupt controller functionality. In addition, it needs to ensure that asynchronous user requests are serviced in a timely manner; the long-latency communication protocol needs to be divided into smaller transactions, since multiple requests may arrive concurrently. To address the above two issues, the system tile continually executes a round-robin loop that invokes various handlers pertaining to directory and interrupt controller functionality. The "context switches" present in Figure 2-5 depict when the next handler is invoked. Table 4-4 lists the various handlers, along with their specific functionality.

System Handler	Logical Function
Static Network Handler	directory
GDN Handler	directory, interrupt
Interrupt Handler	interrupt
Deadlock Detector	interrupt

Table 2.1: System Handlers

Types of GDN Messages
user→interrupt reply
user→interrupt "done"
user→directory contact
user→directory "done"
directory→interrupt request

Table 2.2: GDN Messages Utilized in Coherence Protocol

Static Network Handler

The static network handler is responsible for addressing load-word and store-word messages that have been forwarded from the memory controller. A LW request signifies that a new reader for a shared address has emerged. Based on the current state associated with the address, the static network handler updates the state and takes the appropriate action. A SW message contains the most up-to-date version of the cache-line. If appropriate, the line is patched before writing it back to main memory, and forwarding it along to any new readers.

GDN Handler

The GDN handler is used for both directory and interrupt controller functionality. It is responsible for interpreting user-to-directory, user-to-interrupt, and directory-to-interrupt GDN messages utilized in the shared memory protocol. These messages are all present in Figure 2-5, and a listing of them is given in Table 2.2.

Interrupt Handler

Whenever a directory wishes to contact a user tile, it messages the tile's corresponding interrupt controller. The controller notes the request in a bit-vector, and when the interrupt handler is next invoked, it interrupts any user tiles that are in the "not busy" state.

Deadlock Detector

The deadlock detector handler checks the amount of time that has elapsed since each of the three user tiles has been pinged. If this time exceeds 20,000 cycles, then the necessary deadlock resolution measures in Section are initiated.

2.2.6 Directory-State Transitions

In addition to the **exclusive** and **read states** in Figure 1-1, the directory maintains three more transient states: **read-lock**, **read-lock-bypass**, and **exclusive-pending**. These states are necessary because the read-to-exclusive and exclusive-to-read transitions are no longer atomic in our "multi-tasked" system tile. For example, after a directory tile has issued invalidate requests to user tiles, it doesn't wait for their responses; instead, it context switches to another handler.

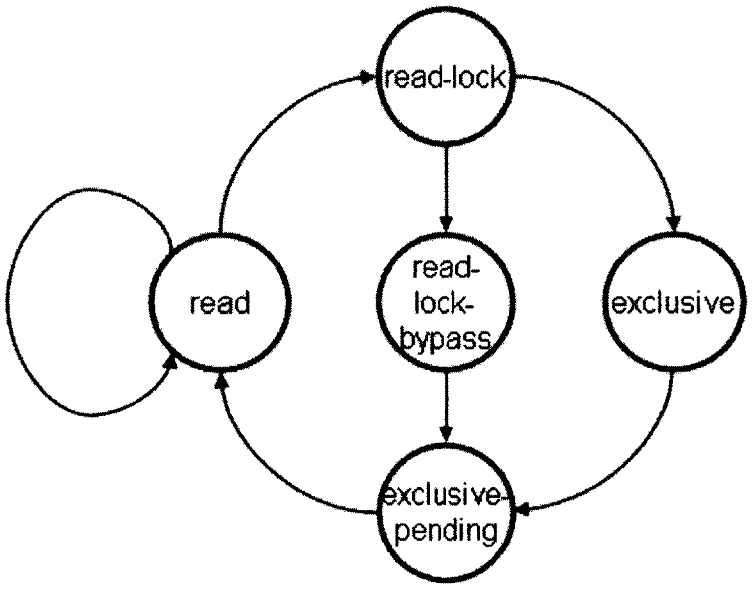


Figure 2-6: State Transition Diagram (Original)

The read-lock state signifies that a tile has requested exclusive ownership of a cache line, and that invalidate requests have been sent to all current readers. Thus, when the last reader has acknowledged the invalidate request, the state transitions from read-lock to exclusive, and the writing tile is notified that it may proceed.

The exclusive-pending state signifies that a write-back request has been sent to a tile. This state is necessary, so as to prevent multiple write-back requests from being issued. When the writeback arrives, the state will transition to read.

Figure 2-6 depicts a bypass path from read-lock to exclusive-pending. The read-lock-bypass state is necessary because asynchronous read/write requests may arrive when the state is in read-lock. Thus, the read-lock-bypass state notes that when the last invalidate has been received, the state should transition to exclusive-pending and a writeback request should be sent.

These state transitions are comprehensively detailed in Table 2.3.

Pending Address Table

Along with the intermediate states described in the previous section, some additional data structures need to be maintained. The pending address table contains addresses that: (1) are currently exclusive on a tile or (2) would like to be exclusive on a tile. Along with each address are a list of pending readers and patches. When the line is written back to main memory, the patches are applied and the readers are sent the line.

State	Stimulus	Response	Next State
read	new reader	add to list, send line	read
read	new writer	send invalidates to readers	read-lock
read-lock	last inv. ack	ack sent to writer	exclusive
read-lock	new reader	note request in pending address table	read-lock-bypass
read-lock	new writer	note patch in table send new writer invalidate	read-lock-bypass
read-lock-bypass	last inv. ack	invalidate sent to writer send writer flush & inv. request	exclusive-pending
read-lock-bypass	new reader	note request in pending address table	read-lock-bypass
read-lock-bypass	new writer	note patch in pending address table	read-lock-bypass
exclusive	new reader	note request in pending address table, ask for writeback from owner	exclusive-pending
exclusive	new writer	send invalidate, note patch, ask for write-back from owner	exclusive-pending
exclusive	writeback	spontaneous writeback - set read on writer	read
exclusive-pending	new reader	note request in pending address table	exclusive-pending
exclusive-pending	new writer	send invalidate, note patch	exclusive-pending
exclusive-pending	writeback	patch line in RAM, send to readers	read

Table 2.3: State Transitions (Original)

Chapter 3

Building a Working System

The original cache coherence system was consistent with a directory based implementation, and was the first prototype shared memory system developed for a tiled architecture. As is common with first-pass systems, there were many issues that impeded the successful execution of shared memory applications. These issues ranged from trivial oversights to protocol problems. The problems were sequentially detected and corrected as shared memory applications were developed and tested on the system. As more issues were resolved, it became possible to develop more complicated applications, and in turn more subtle errors surfaced. Table 3.1 classifies the different types of problems encountered.

The trivial problems consisted of callee registers being overwritten by callers, incorrect loop termination conditions, and overwritten return addresses. Most errors of this type were resolved with one line of assembly code.

The next class of problems were system-specific, and were primarily concerned with buffer overflows and deadlock conditions arising from the dynamic network. To address these issues, the system was tailored to be more aware of the underlying architecture. The limited hardware buffers were extended through software, and non-blocking network reads were made in the appropriate circumstances.

Protocol problems in the system, although limited, required a generous amount of time to address. One such problem dealt with an incorrect transition in the underlying FSM, and another dealt with an incorrect deadlock resolution mechanism. The latter

Table 3.1: Types of System Problems

Type	Number	Examples
Trivial	7	caller registers overwritten return address overwritten incorrect loop termination incorrect parsing of bit-vectors
System	2	static network buffering non-blocking MDN reads
Protocol	2	exclusive tile incorrectly becomes reader incorrect deadlock resolution
Race Conditions	2	deadlock while performing SW EC & MDN interrupt handle same address

problem surfaced only when the interrupt controller assigned to the tile was physically different from the directory controller responsible for the deadlocked address.

Finally, the most difficult to detect problems were race conditions that were only uncovered after running large-scale applications for millions of cycles. Although they were straightforward to address, the process of detecting and diagnosing these problems was extremely tedious. Whenever user or directory tile code was modified, these errors would be displaced elsewhere within the program execution, complicating the debugging process. One race condition occurred when a deadlock occurred on a SW (as opposed to LW), and another surfaced when the event counter (EC) and MDN interrupt handler were both operating on the same address.

3.1 Static Network Buffering

There are only 8 hardware buffers on the static network between the memory controller and any given system tile (4 buffers on EAST port, and 4 buffers on processor). Under heavy loads, any of these buffers may become full, and the memory controller will subsequently stall when attempting to push another word onto the static network. Only when the appropriate system-tile reads from the static network (and clears the buffer overflow) will the memory controller be able to resume handling

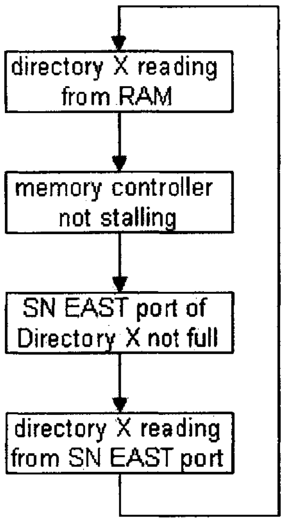

directory X reading from RAM
memory controller not stalling
SN EAST port of Directory X not full
directory X reading from SN EAST port

Figure 3-1: Dependencies between Static Network and MDN Network

memory requests. Thus, if the system-tile with a static-network buffer-overflow is currently reading from `$cmni`, deadlock will result. Figure 3-1 depicts the circular dependency between the static and MDN networks.

To prevent this deadlock condition from arising, system tiles must ensure that *all reads from* `$cmni` *are non-blocking.* Instead of executing a blocking LW instruction, the system-tile should manually issue the appropriate MDN header for the address to be retrieved, and it should then continually poll `MDN_BUF` while waiting for a response. While it is polling for a response, it should also check if the static network buffer's EAST port is full. If this is indeed the case, an arbitrary number of words should be read off the static network to prevent the memory controller from stalling. In this manner, the cyclic dependency in Figure 3-1 is broken.

Now, to read from the static network, the circular buffer should first be checked before executing a read from `$csti`. We will now prove that this mechanism is sufficient to prevent deadlock:

1. A system-tile cannot deadlock with itself; e.g. it cannot perform a blocking read from the MDN when its own static network buffer is full. This is a simple consequence of the above function.

2. A system-tile cannot permanently stall on a read from the MDN when another

Write to Buffer	31 cycles
Read (non-empty buffer)	55 cycles
Read (empty buffer)	10 cycles

Table 3.2: Latencies of Static Network Buffer Operations

system-tile's static network buffer is full. Suppose that system-tile X is performing a read from the MDN, and system-tile Y has a full static network buffer. Once every round-robin system loop, system-tile Y will check to see if its buffer is full, and will subsequently clear it. Thus the memory controller will eventually stop stalling, and system-tile X will be able to complete is read/write on the MDN.

Although these new reading & writing mechanisms bypass the aforementioned deadlock condition, they introduce additional overhead that leads to performance degradation. Table 3.2 shows the number of cycles spent reading from and writing to the buffered static network:

3.2 Exclusive Tile transitions to Reader

In the original system, when a tile performs a flush either due to a flush request or cache eviction, the directory tile automatically sets the tile as a reader. However, this is the incorrect behavior, since the line may have been patched, in which case the flushing tile no longer has a valid copy of the cache line (refer to Figure 3-2).

Therefore, a flush request should never be issued to an exclusive tile. This is because a patch may occur after the request is sent out, and before the flushed data is received at the system tile, as depicted in Figure 3-3. Since there is no way to prevent this race condition, a *flush & invalidate is the only correct request to an exclusive tile.*

1. Tile X exclusive on address A
2. i. Tile Y attempts to read A
   ii. address A goes to exclusive-pending on X
   iii. flush request sent to X
3. i. Tile Z writes to A
   ii. Tile Z's modification to A noted in pending address table
   iii. invalidate response sent to Z
4. i. Tile X flushes A
   ii. cache line A reconstructed
   iii. cache line A sent to Y
   iv. Tile X set as reader on A **(ERROR)**

Figure 3-2: Exclusive Tile incorrectly Transitions to Reader

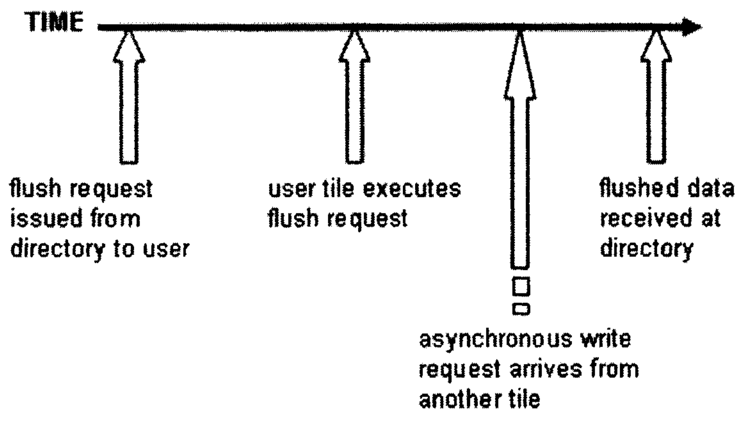

TIME
flush request issued from directory to user
user tile executes flush request
asynchronous write request arrives from another tile
flushed data received at directory

Figure 3-3: Arrival of Asynchronous Writes

1. i. Tile X performing `LW A`, and exclusive on B

   ii. Tile Y performing `LW B`, and exclusive on A

   X is in the pending address table for addr A, and Y is in the pending address table for addr B. DEADLOCK.

2. i. Tile X sent spoof data

   ii. Tile X writes back B in MDN interrupt

   iii. Tile X resets PC to `LW A`

3. i. B sent to Tile Y; Tile Y stops stalling

   ii. Tile Y flushes A in MDN interrupt

4. A sent to Tile X (from pending addr table for addr A) **(ERROR)**

5. A sent to Tile X (from LW A)

Figure 3-4: Improper Deadlock Resolution

3.3 Deadlock Resolution: Purging the Pending Address Table

In the current system, when a deadlock occurs, the appropriate interrupt controller forwards a fake cache-line along to the stalling tile. After receiving the cache-line, the tile invalidates the spoofed data, services any pending MDN interrupts, and resets its program counter to the stalling instruction.

However, one must make the observation that if a tile is in deadlock, then it must exist in some directory's pending address table [1]. Therefore, prior to sending the spoofed cache-line, any directory containing a reference to this tile must erase it in its pending address table. Otherwise, a second cache-line will unnecessarily be sent after the tile re-executes the stalling memory instruction. Figure 3-4 depicts the aforementioned error in deadlock resolution. Therefore, prior to sending any spoof data for the purpose of deadlock resolution, the interrupt controller must clear any reference to this tile in its own pending address table, as well as message the other directory tiles to do the same. Furthermore, the spoof data will not be sent until:

[1]If this were not the case, then the tile would eventually receive the cache-line which it is requesting, and thus no deadlock would exist

(1) all directory tiles have acknowledged the request and (2) the tile was found in atleast one pending address table. The former ensures that deadlock resolution will not proceed until the tile is cleared from all tables, and the latter ensures that the tile is infact still in deadlock.

3.4 Deadlock Resolution: Event Counter Check

When a tile deadlocks while performing a SW operation (as opposed to LW), the current cache-coherence system fails to correctly handle proper deadlock resolution. The error arises the moment the spoofed data is received by the user-tile - when the SW operation completes, the event counter for the clean$\rightarrow$dirty transition is decremented and the tile handles the ensuing MDN interrupt. After exiting the MDN interrupt, it will immediately branch to the event-counter interrupt as a consequence of its write to the spoofed data. Refer to Figure 3-5 for a depiction of the above events.

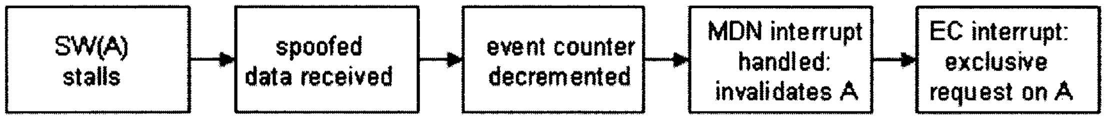

SW(A) stalls
spoofed data received
event counter decremented
MDN interrupt handled: invalidates A
EC interrupt: exclusive request on A

Figure 3-5: Improper Deadlock Resolution on SW Stall

Requesting an exclusive copy of the spoofed data not only degrades performance, but it is also functionally incorrect. The spoofed data should be disregarded, and it is actually invalidated in the MDN interrupt. There are many scenarios in which this leads to either an incoherent state or deadlock.

To rectify this error, the event counter is automatically reset to 0 whenever a user-tile has detected that it has received a false cache-line. In this manner, it will not prematurely jump to the event counter handler after it has processed the MDN interrupt.

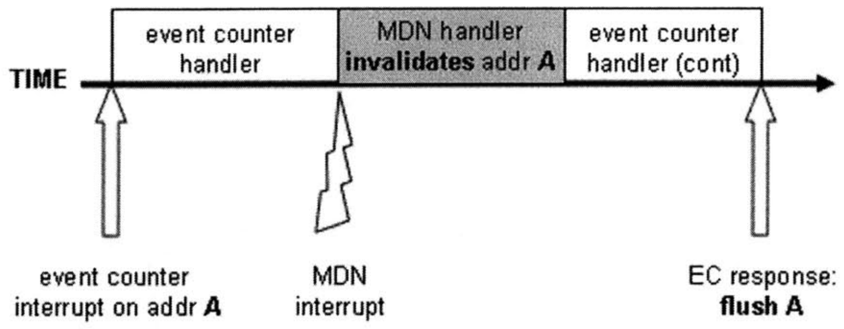

TIME
event counter handler
MDN handler invalidates addr A
event counter handler (cont)
event counter interrupt on addr A
MDN interrupt
EC response: flush A

Figure 3-6: Event Counter Interrupt & MDN Interrupt Race Condition

3.5 Event Counter & MDN Interrupt Handling Same Address

To reduce the latency of various transactions within the cache coherence protocol, the event counter interrupt is able to take MDN interrupts. The motivation behind this is that the EC interrupt will spend most of its time waiting for a response from a directory tile, so this stall time can be overlapped with the service of MDN interrupts [2].

However, a race condition exists when both the EC and MDN interrupts are servicing the same address. For example, if a particular address is invalidated in the MDN interrupt, then the EC interrupt becomes unable to fulfill a flush request for the same address, as depicted in Figure 3-6. In terms of the protocol employed, this is a logical error as well - a tile should only be in the event counter interrupt when it has written to a line that is *still resident in the cache.*

The exact race condition occurs when a directory tile issues an MDN interrupt to invalidate an address A on tile X, between the time that tile X enters the event counter interrupt for address A and the time that the EC notification is received by the directory tile. To resolve this race condition, two precautions are taken:

1. Interrupts in the EC handler are initially turned off until the EC notification is sent to the tile.

[2]refer to Sections 2.2.3 and 2.2.4 for descriptions of MDN and EC interrupts

2. The address of the offending SW instruction is stored away. When an MDN "invalidate" request occurs for the same address, it is simply ignored. However, a (false) ack that the invalidate was completed is sent.

This solution ensures that the EC handler is able to fulfill a "flush" request. However, we must prove that the behavior of the system is still correct. Suppose an MDN request for an address A arrives moments after the event counter handler is entered for the same address A. We can make the following statements regarding the state of the system:

1. If the tile enters an event counter interrupt for address A, then it must currently be a reader on A [3]. Therefore any MDN request on this address A must be an invalidate.

2. The current state is read-lock or read-lock-bypass, as these are the only states where an invalidate request is issued from a directory (refer to Table 2.3).

3. The tile will send an EC notification to the directory BEFORE it jumps to the MDN handler (since interrupts are initially disabled). Therefore the directory will receive the EC notification BEFORE it receives the (false) MDN invalidate ack from the tile.

These points imply that the state cannot transition out of read-lock or read-lock bypass before the EC notification reaches the directory tile. When the EC notification gets processed, the state transition will either be read-lock→read-lock-bypass, or read-lock-bypass→read-lock-bypass, and the EC reply will be either be: (1) invalidate or (2) flush & invalidate. Thus, at the very least, address A is invalidated by the EC interrupt, and the false MDN "invalidate" ack is harmless.

[3]for the cache line to have been dirtied in the first place, it must have been in the "read" state for the tile

Chapter 4

Performance Optimizations

Five major performance optimizations were implemented over the base cache-coherence system. The optimizations were incrementally designed according to the major bottlenecks in the system. In sequential order, they include the mapping of shared addresses to system-tiles, the optimized round-robin system-loop, hot-forwarding, read-lock-bypass→read transitions, and support for voluntary flushing.

The most glaring performance bottleneck was the inefficient manner in which shared addresses were mapped to directory-tiles. By utilizing the high-order bits of the address to determine address ownership, the spatial locality of memory references results in an individual directory tile servicing all requests. Thus, ownership was reassigned based on the low-order bits, to achieve better load-balancing.

Next, the system-tile round-robin loop was restructured to invoke the various system functions at empirically determined frequencies that resulted in optimum performance. Certain types of messages were determined to arrive at system-tiles more often than others, and therefore their corresponding handlers needed to be invoked at a greater frequency to prevent back-log. For example, since GDN messages are heavily utilized in the protocol, they arrive at system-tiles more often than static network LW/SW requests. Thus, the GDN handler needs to be invoked at a greater frequency than the static network handler.

The most extensive modification to the system was termed hot-forwarding, and it involved a major rehaul of the communication protocol between system-tiles and

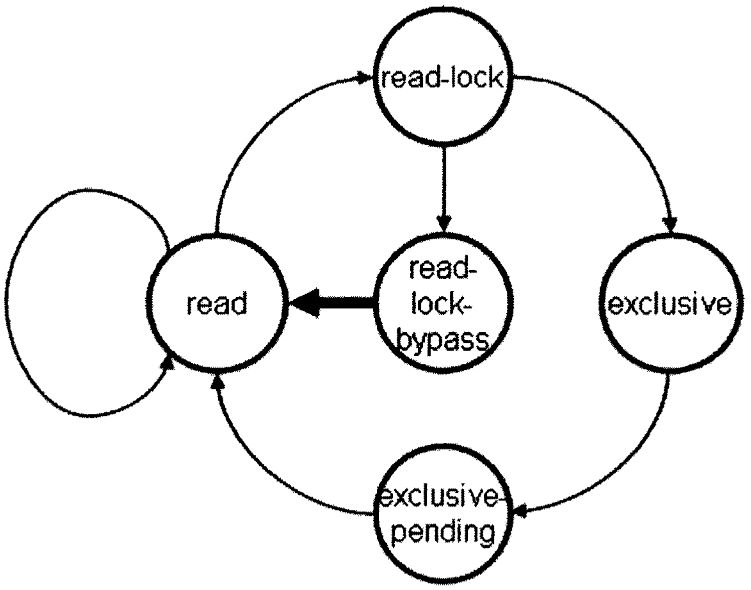


Figure 4-1: Optimized Cache-Coherence Scheme

user-tiles. Hot-forwarding effectively reduces the extent of handshaking in the cache coherence protocol, resulting in reduced latency for various coherence transactions. The premise behind this scheme is that directory requests for a particular user-tile will be throttled, so that they may be directly forwarded to the user-tile rather than batching them at the directory.

The new state transitions were superimposed on the existing FSM, and they served to bypass unnecessary states. This resulted in reduced transaction latency, as well as in the elimination of certain deadlock scenarios. Figure 4-1 depicts the new state transitions diagram of the optimized system, and the exact transitions are comprehensively detailed in Table 4.1.

Finally, the motivation behind the last optimization stemmed from the high-level design of shared memory applications. The general model for developing shared memory applications relies on a single tile that initializes all global data structures. To reduce the effect of cold-starts, the initializing tile can relinquish all ownership of shared addresses via voluntary flushing, before tiles begin to manipulate shared memory.

State	Stimulus	Response	Next State
read	new reader	add to list, send line	read
read	new writer	send invalidates to readers	read-lock
read-lock	last inv. ack	ack sent to writer	exclusive
read-lock	new reader	note request in pending address table	read-lock-bypass
read-lock	new writer	note patch in table send new writer invalidate	read-lock-bypass
read-lock-bypass	last inv. ack	invalidate sent to writer patch-line in RAM, send to readers	read
read-lock-bypass	new reader	note request in pending address table	read-lock-bypass
read-lock-bypass	new writer	note patch in pending address table	read-lock-bypass
exclusive	new reader	note request in pending address table, ask for writeback from owner	exclusive-pending
exclusive	new writer	send invalidate, note patch, ask for write-back from owner	exclusive-pending
exclusive	writeback	spontaneous writeback - clear state (no readers)	read
exclusive-pending	new reader	note request in pending address table	exclusive-pending
exclusive-pending	new writer	send invalidate, note patch	exclusive-pending
exclusive-pending	writeback	patch line in RAM, send to readers	read

Table 4.1: State Transitions (Optimized)

Shared Address Range	System Tile
0x06000000 - 0x07FFFFFF	SYSTEM-TILE 3
0x0E000000 - 0x0FFFFFFF	SYSTEM-TILE 3
0x16000000 - 0x17FFFFFF	SYSTEM-TILE 3
0x1E000000 - 0x1FFFFFFF	SYSTEM-TILE 3
0x26000000 - 0x27FFFFFF	SYSTEM-TILE 7
0x2E000000 - 0x2FFFFFFF	SYSTEM-TILE 7
0x36000000 - 0x37FFFFFF	SYSTEM-TILE 7
0x3E000000 - 0x3FFFFFFF	SYSTEM-TILE 7
...	...
0x7E000000 - 0x7FFFFFFF	SYSTEM-TILE 15

Figure 4-2: Original Mapping of Addresses to System Tiles

4.1 Mapping of Shared Addresses to System Tiles

Since there are four directory tiles in the current revision of the cache coherence system, each tile has been assigned ownership of 1/4th of the shared memory address space. In the previous system, ownership was assigned based on bits 30-29 of the address, so each directory tile was responsible for approximately 4 chunks of 17MB of contiguous memory, as depicted in Figure 4-2.

However, as various applications were profiled on the system, it became apparent that a single system-tile was servicing all EC notifications and was solely responsible for maintaining cache coherence. This bottleneck was a consequence of the manner in which shared addresses were mapped to system-tiles. Since shared memory data structures occupy contiguous regions of memory, all references to any part of the structure will go through the same system tile. Thus, if a shared memory application utilizes a global array that is manipulated by all tiles, a single system tile will be responsible for maintaining coherence. One potential solution to this load imbalance is to configure shared malloc so as to return successive addresses in different regions of memory. However, this solution is ineffective when a single shared data structure has heavy contention among different processors.

The best load-balancing solution was determined by observing the organization of hardware caches. To reduce contention for cache-lines, caches are indexed by the low bits of the address. A similar solution for the mapping of addresses to system-tiles

Shared Address	System Tile
CACHE-LINE mod 4 = 0	SYSTEM-TILE 3
CACHE-LINE mod 4 = 1	SYSTEM-TILE 7
CACHE-LINE mod 4 = 2	SYSTEM-TILE 11
CACHE-LINE mod 4 = 3	SYSTEM-TILE 15
CACHE-LINE mod 4 = 0	SYSTEM-TILE 3
...	...

Figure 4-3: Re-mapping of Addresses to System Tiles

was implemented; bits 6-5 of the address were used to determine which system-tile had ownership over the address [1]. To implement this optimization, modifications were made to the event counter handler and the memory controller to use bits 6-5 rather than bits 30-29, as well as some minor modification to the directory tile code.

4.2 Optimized System Loop

System tiles in the previous cache-coherence system executed a round-robin loop that invoked the necessary directory & interrupt controller routines. Table 4-4 lists each routine, and which category it belongs to.

The gdn_handler routine is used for both directory and interrupt controller functionality, and is responsible for correctly interpreting GDN messages from user-tiles. Because of the dual role provided by this routine, it is logical to invoke it twice during a single round-robin loop. The motivation for adding a third invocation of gdn_handler is clear when we examine the relative execution times of the various routines; the static_network_handler is almost an order of magnitude slower than the gdn_handler. Furthermore, empirical evidence suggests that GDN messages arrive at system-tiles at a faster rate than static-network LW and SW requests. It therefore makes sense to invoke the gdn_handler more frequently, to prevent a large backlog of GDN messages that eventually lead to long latency transactions. After testing various applications, it was empirically determined that 3 invocations of the handler, per round-robin loop, was optimum.

[1] bits 6-5 are the low-order bits of the cache line address

```
Original system-loop

while(1) {
    static_network_handler
    gdn_handler_begin
    pending_irq_begin
    deadlock_detector_begin
}
```

```
Optimized system-loop

while(1) {
    static_network_handler
    gdn_handler_begin
    gdn_handler_begin
    gdn_handler_begin
    pending_irq_begin
    deadlock_detector_begin
}
```

Figure 4-4: Optimized system-loop

4.3 Exclusive Tile Voluntary Flush

In the original system, when a tile voluntarily flushes a line that it has exclusive access to, the tile is set as a reader. A voluntary flush results from one of the following 3 actions:

1. an exclusive cache-line is evicted from the cache to make room for another cache-line
2. the user-code explicitly performs a flush & invalidate
3. the user-code explicitly performs a flush

In scenarios 1 & 2, the address is invalidated as well, so marking the tile as a reader unnecessarily introduces additional latency in the shared memory protocol (e.g. when a new writer arises, an invalidate request is unnecessarily sent to this tile). Now, if we prevent scenario 3 from occurring by introducing a restriction on the programmer, then we can safely remove the flushing tile as a reader. The new transitions from exclusive are present in Table 4.2.

The benefit of this optimization is more apparent when designing shared memory applications. The model for designing shared memory applications usually employs a single tile that initializes all data structures and shared memory primitives before the parallel algorithm commences. However, after initialization, the tile has exclusive

Table 4.2: State transitions from "exclusive"

State	Stimulus	Response	Next State
exclusive	new reader	send flush&inv to writer	exclusive-pending
exclusive	new writer	send inv. to new writer send flush&inv to writer	exclusive-pending
exclusive	writeback	spontaneous writeback - clear state (no readers)	read

access to all addresses. When the algorithm begins running on other tiles, the system will be bottle-necked as a flood of MDN flush requests are sent to the initializing tile.

To avoid this severe performance degradation, the programmer should take care to flush & invalidate all global data structures after they have been initialized. In this manner, when the algorithm commences, the directory table is completely empty since no tiles are declared as readers or writers. If the above optimization were not in place, then the initializing tile would be a registered reader of all addresses after the flush, and a bottleneck would result when other tiles attempt to write to addresses (although this bottleneck is far less harmful than the case where no flush is executed at all). The slight performance degradation arises from the imbalance in initial address ownership, and the bottleneck is prevented by clearing ownership of all addresses.

4.4 Cache-line Reconstruction in Read-Lock

The purpose of a flush & invalidate request is to make an exclusive cache-line globally available to all user-tiles. Since the EC handler is triggered on clean-to-dirty transitions, the handler is only invoked on the first write to an address, and is uninvoked on subsequent writes (since the line is already dirty). Thus, although the EC handler forwards along the newly written data to the appropriate directory tile, if the particular user-tile is given exclusive access, subsequent modifications to the cache-line will go un-handled. Now, the motivation behind flushing the address should be clear; the exclusive tile has a unique version of the address that needs to be distributed.

However, the current system dictates that the state transition out of read-lock-

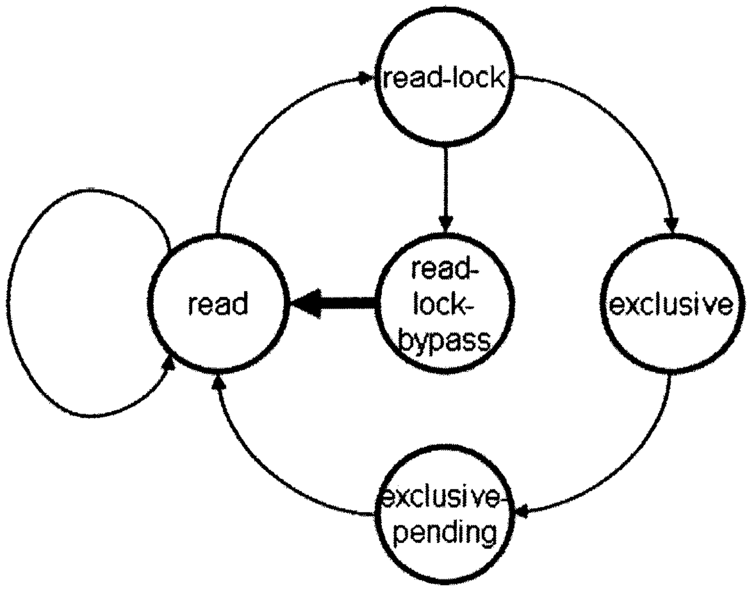


Figure 4-5: New read-lock-bypass transition

bypass should be exclusive-pending. In other words, when the last reader has invalidated in the read-lock-bypass state, the system issues a flush & invalidate response to the writing tile's EC handler. But given our above discussion regarding the motivation behind flushing, we observe that this flush is in-fact unnecessary. Since the writing tile has only made one modification to the cache-line, and since this newly written data has been forwarded to the directory tile as part of the EC notification, the directory-tile has all the necessary data to reconstruct the cache-line.

This unnecessary flush request adds a significant amount of latency to the transaction, since the flushed data is pushed onto the static network by the memory controller, and may now be queued behind other static network messages. Furthermore, it introduces more traffic on the static-network, which may lead to more aggressive buffering and therefore greater overhead when processing other transactions.

The optimized system reconstructs the cache-line in the read-lock-bypass state when the last reader has invalidated. As in the original system, it still records any writes made during read-lock-bypass, and appropriately patches the line. An invalidate is sent to the writing tile, and the new state directly transitions to read. Figure

Table 4.3: State transitions from "read-lock-bypass"

State	Stimulus	Response	Next State
read-lock-bypass	last inv. ack	invalidate sent to writer patch-line in RAM, send to readers	read
read-lock-bypass	new reader	note request in pending address table	read-lock-bypass
read-lock-bypass	new writer	send writer invalidate, note patch in pending address table	read-lock-bypass

4.3 depicts the new transition from read-lock-bypass.

4.5 Hot-Forwarding Directory Tile Requests

The original cache-coherence system relied on a system-tile having two distinct roles as (1) an interrupt controller and (2) a directory. This dual role is necessary due to the nature in which external MDN interrupts are issued on the RAW architecture; an interrupted tile has no hardware mechanism through which to detect the source of its interrupt (the motivation behind omitting this hardware feature stems from its lack of scalability for larger fabrics of RAW tiles). To address this issue within the cache-coherence system, an interrupted user-tile contacts a predetermined interrupt controller to ascertain which directory has outstanding requests for it. Furthermore, the interrupt controller is responsible for only issuing one interrupt at a time to a particular tile, and it is also central in determining and resolving deadlock scenarios.

However, since the interrupt controller acts as an intermediary between directory tiles and user-tiles, it adds another level of indirection in the cache-coherence communication protocol. When a directory tile wishes to flush and/or invalidate an address on a specific tile, it must first send a message to the corresponding interrupt controller. The interrupt controller then interrupts the user-tile (if it is not currently in MDN interrupt), which subsequently acks the controller. Once this ack is received, the interrupt controller sends the user-tile a list of directory tiles which it must con-

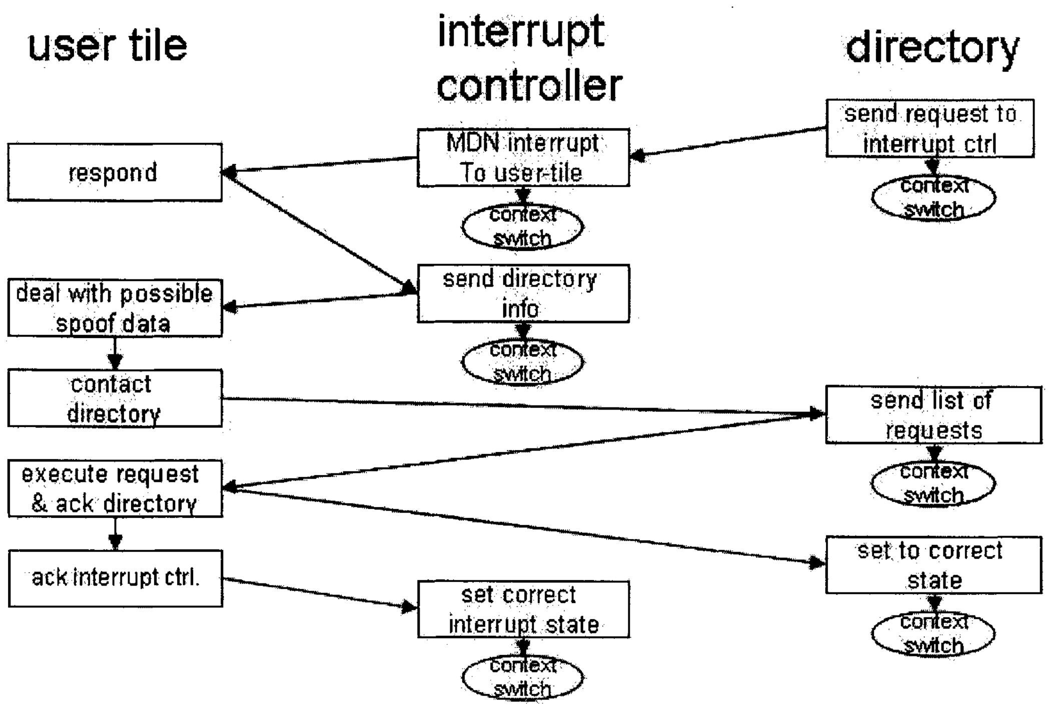


Figure 4-6: Original User-Directory Communication Sequence

tact. The user-tile then contacts each directory tile individually, and waits for a list of requests. When these requests are successfully received & executed, the user-tile acks each directory tile, and once all directory tiles are serviced, it acks the interrupt controller and exits the interrupt.

It is fairly clear that issuing a request to a particular user-tile is a process with relatively high latency. This extensive hand-shaking protocol, as seen in Figure 4-6, was deemed necessary so as to prevent deadlock on the GDN between system-tiles [1]. However, one immediate optimization can be made by removing the initial handshaking process between the interrupt controller and the user-tile. Instead of waiting for a GDN ack, the interrupt controller can send the list of directory tiles along with the actual MDN interrupt [2].

The resulting system depicted in Figure 4-7 still employs a relatively lengthy process for issuing requests. The most straightforward approach to reducing the ex-

[2]This optimization leads to a slight complication when handling deadlocks, but a relatively straightforward fix is possible

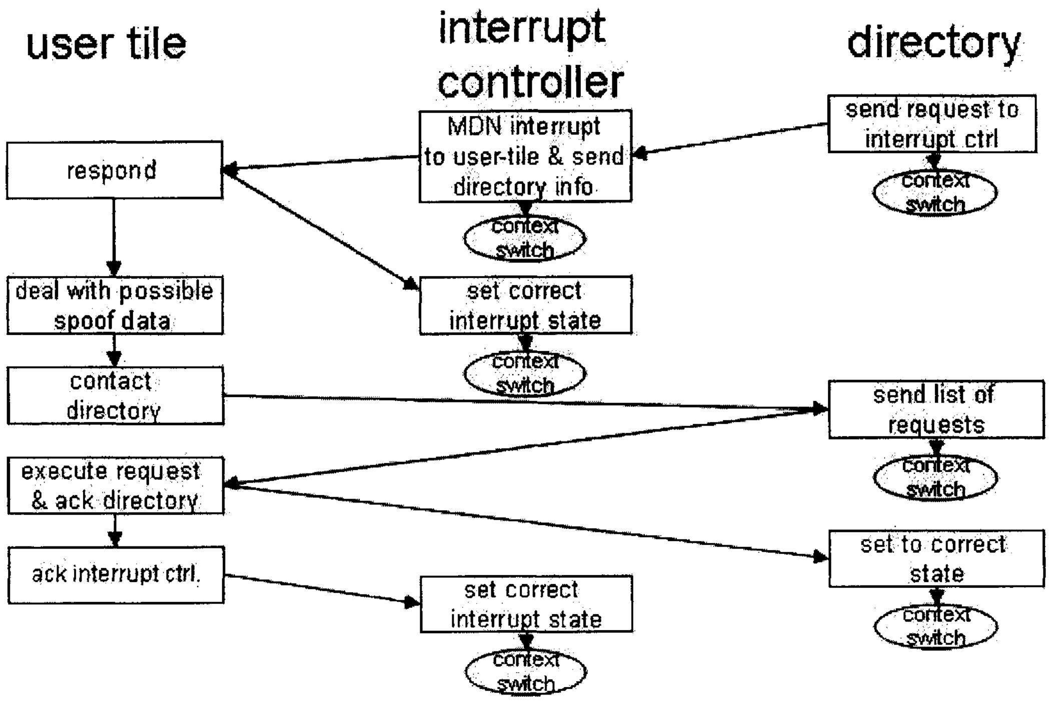


Figure 4-7: User-Directory Communication Sequence (first-pass optimization)

tent of the handshaking process would involve directory tiles directly forwarding cache requests to interrupt controllers. The interrupt controller would then be responsible for providing these requests to the user-tile, essentially removing another level of indirection present in the original scheme. But since a directory tile may have an arbitrarily large number of requests for a particular tile, two system-tiles may deadlock as they send these GDN messages to each other. We are therefore presented with a problem of performance vs. correctness. The high performance implementation tends to fail in some remote circumstances, while the the correct system exhibits significantly higher latency for all transactions. As is common in systems development, a marriage of these two is highly sought after.

The resulting optimization is termed "hot-forwarding", as depicted in Figure 4-8, since interrupt controllers have the ability to directly forward requests from directory tiles to user-tiles in certain circumstances. More precisely, a directory-tile can hot-forward exactly 1 request at a time for a particular tile, as depicted in the algorithm in Figure 4-9. Otherwise, if there are outstanding requests in the local table, the

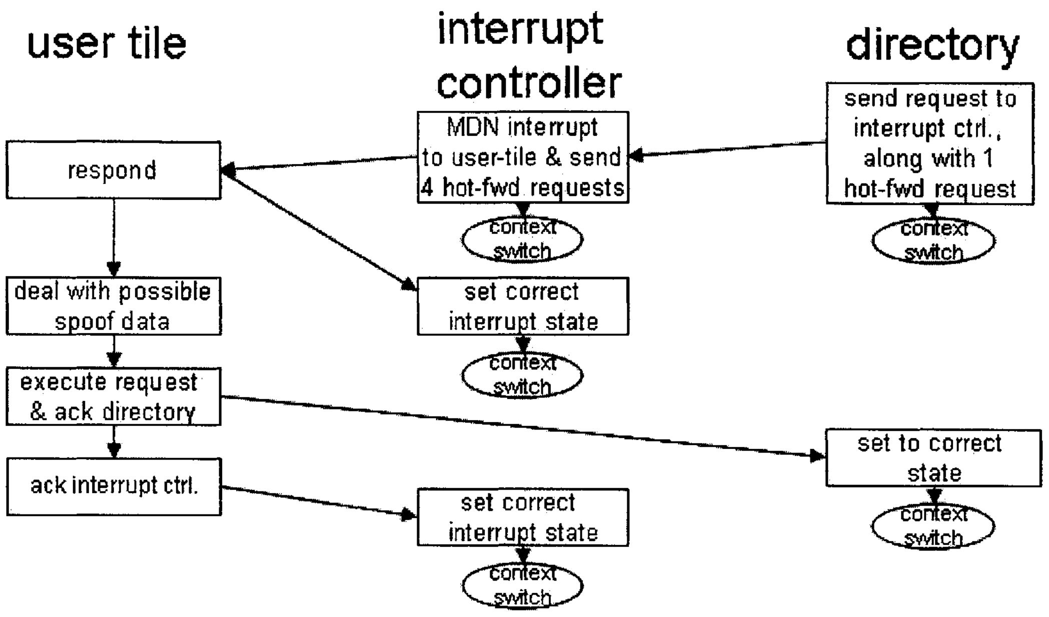

user tile
interrupt controller
directory
send request to interrupt ctrl., along with 1 hot-fwd request
MDN interrupt to user-tile & send 4 hot-fwd requests
respond
context switch
context switch
deal with possible spoof data
set correct interrupt state
context switch
execute request & ack directory
set to correct state
ack interrupt ctrl.
set correct interrupt state
context switch
context switch

Figure 4-8: User-Directory Communication Sequence utilizing Hot-Forwarding

original system depicted in Figure 4-6 is invoked; the directory tile adds the request to its outstanding request buffer, and a message is sent to the interrupt controller.

```
if(ocreq[tile] == 0 && hot_pending[tile] == 0) {
    hotforward request to interrupt controller
    add request to hot_pending_table
}
if(ocreq[tile] == 0 && hot_pending[tile] != 0) {
    notify interrupt controller of ocreq
} store ocreq in ocreq_table[tile]
```

Figure 4-9: Algorithm for Hot-Forwarding

The premise behind hot-forwarding is that directory-tiles will rarely aggressively send requests to a particular tile. In most circumstances, the time elapsed between consecutive requests for a particular tile will be greater than the time it takes for the user-tile to acknowledge the completion of an individual request. When this is the case, all requests will be sent via the hot-forwarding route. If, however, requests occur at a faster rate, then they will get batched in the outstanding request table of the directory tile, and will be sent via the protocol described by Figure 2-5.

Chapter 5

Higher-Level Shared Memory Support

To facilitate the development of shared memory applications, a library of shared memory and synchronization primitives has been developed. The existing architecture implemented a directory based cache coherence scheme, through the use of dedicated system tiles. In this scheme, addresses with bits 25 & 26 set were designated as shared, and each system tile was responsible for managing a subset of these addresses. To provide higher level support for application development in the C environment, shared malloc routines were introduced to provide the proper memory abstraction. Furthermore, to ease the programmability of distributed shared memory applications, various parallel primitives such as locks and barriers were implemented.

The proposed architecture utilizes a separate RAW tile as a shared memory controller, which is responsible for processing shared memory and synchronization requests issued by user-tiles (refer to Figure 5-1). User-tiles initiate a request by issuing a properly formatted GDN message, and possibly waiting for a subsequent response from the controller. The manner in which the GDN is multiplexed between user requests, and low-level system messages is a subject of [2].

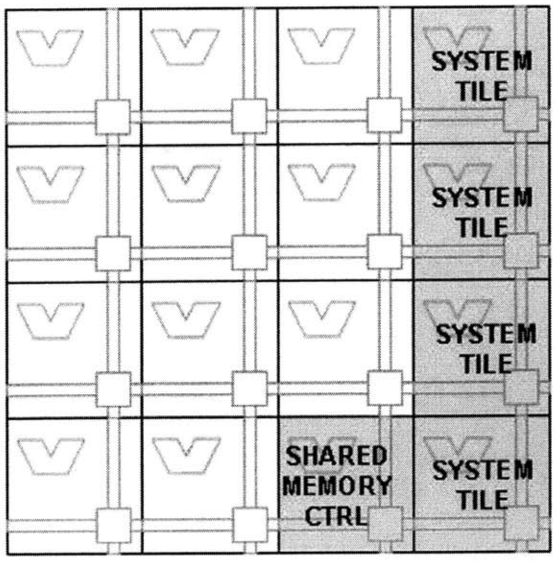


Figure 5-1: Shared Memory Controller on RAW fabric

5.1 Shared Memory Controller

The shared memory controller is a stand-alone tile that processes shared memory & synchronization requests from user-tiles. On a 4x4 RAW fabric, where tiles 3, 7, 11, and 15 are system tiles, tile 14 has arbitrarily been designated as the controller. Requests are sent to the controller via the GDN, and if necessary, replies are sent to the tiles via the GDN as well (refer to Figure 5-3. However, since the low-level cache coherence protocol relies on GDN messages sent from system to user-tiles, the GDN network must be multiplexed between these two types of messages. For the remainder of this discussion, we will designate GDN messages issued via system tiles as "system" messages, and GDN messages issued from the controller as "user" messages.

As part of managing resource allocation, the controller writes 32-byte headers to regions of shared memory. Furthermore, it zeros out newly allocated regions of memory, and promptly flushes & invalidates that region. Since user-tiles are restricted from reading/writing to these header regions, the set of shared addresses the controller modifies is disjoint from the set manipulated by user-tiles. Thus, the controller is declared as exempt from the cache-coherence protocol, and is unregistered with the

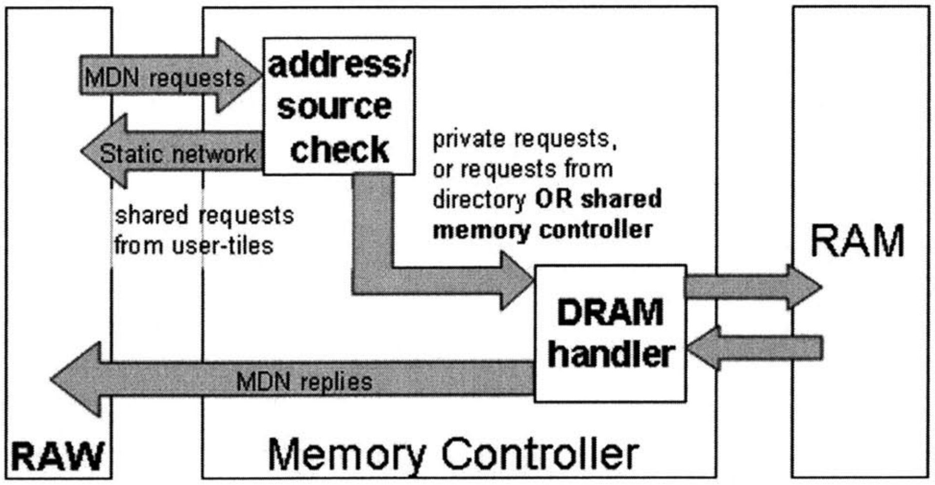

MDN requests
address/ source check
Static network
private requests, or requests from directory OR shared memory controller
shared requests from user-tiles
RAM
DRAM handler
MDN replies
RAW
Memory Controller

Figure 5-2: Modified FPGA

system tiles. Figure 5-2 depicts the modified memory controller in context with the shared memory controller.

5.1.1 Controller-to-Tile GDN Communication

Integrating the shared memory controller into the existing cache-coherence system results in complications, since GDN contention arises between directory tiles issuing low-level "system" messages, and the controller issuing "user "messages. User-tiles that are waiting for "system" messages may instead receive "user" messages, resulting in a misinterpretation of the incoming data. The reverse problem is not possible, since "system" messages only arrive when the user-tile is already in the MDN handler. Note that since the controller is exempt from the cache coherence protocol, it can receive "user" messages without complication.

One potential solution to the above issue is to simply prevent GDN contention, by preventing the possibility of receiving a "user" message when in the MDN handler. This could be done by disabling interrupts when expecting a "user" message. Although straightforward, this potential solution harbors a deadlock condition; the controller may not send a "user" message to this tile until the tile has serviced the pending MDN interrupt.

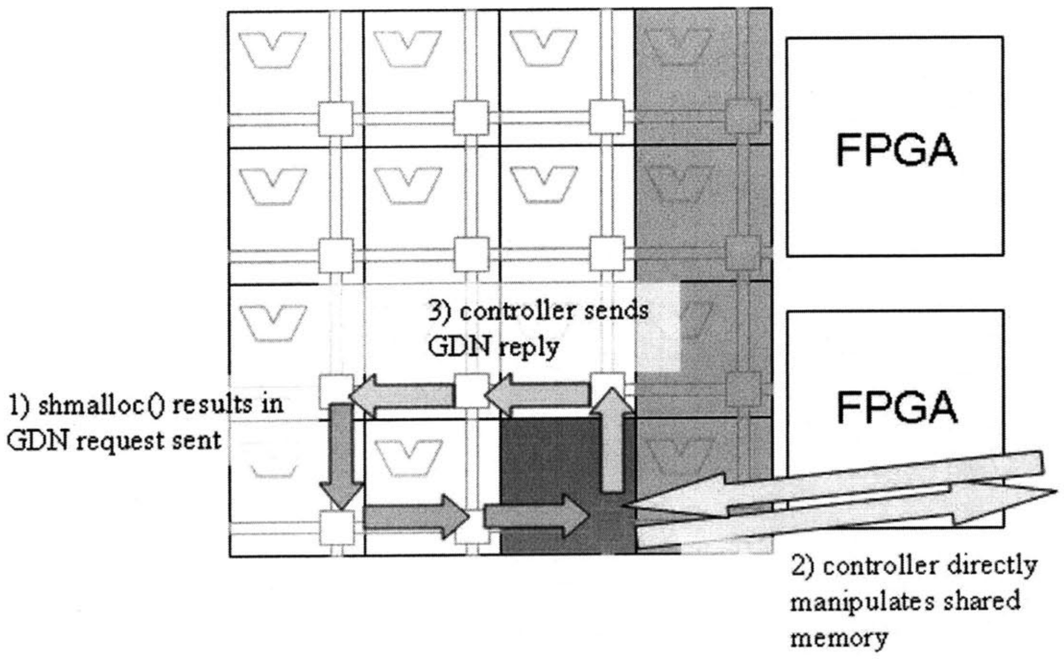

FPGA
FPGA
3) controller sends GDN reply
1) shmalloc() results in GDN request sent
2) controller directly manipulates shared memory

Figure 5-3: User Tile & Shared Memory Controller Communication

Consider the following deadlock scenario involving barriers:

1. Tile 1 exclusive on A, and waiting at `barrier()` (interrupts disabled until "user" message received to exit `barrier)`
2. Tile 2 stalls on LW A, prior to executing barrier()
3. MDN flush & invalidate sent to Tile 1

In this specific scenario, Tile 2 will not execute `barrier()` until Tile 1 does a flush & invalidate. However, Tile 1 will not flush & invalidate until it receives a "user" message and exits the barrier. Figure 5-4 clearly illustrates the cyclical dependency that results in this deadlock.

Therefore, to prevent deadlock from occurring, it is necessary that user-tiles have the ability to service MDN requests while waiting for a "user" message. The system is now faced with the aforementioned problem that a "user" message may infact arrive while servicing an MDN interrupt. It is therefore necessary for the GDN to be multiplexed between "user" and "system" messages, and this is achieved by having

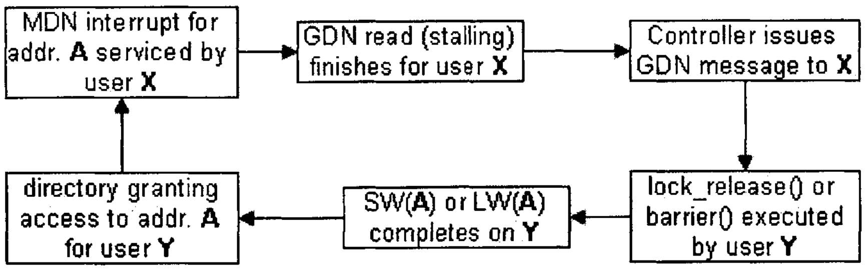

MDN interrupt for addr. A serviced by user X
GDN read (stalling) finishes for user X
Controller issues GDN message to X
lock_release() or barrier() executed by user Y
SW(A) or LW(A) completes on Y
directory granting access to addr. A for user Y

Figure 5-4: Stalling GDN Reads: Deadlock with Shared Memory Controller

"user" messages begin with the word 0xF000, which is distinct from the first word of all "system" messages. In this manner, user-tiles can immediately distinguish the two types of messages. If a "user" message does infact arrive during an MDN interrupt, then a flag is set and the message is stored in a buffer. Now, the process of waiting for a GDN "user" message involves continuously polling both GDN_BUF and the aforementioned flag.

```
while(1) {
  interrupts_off();
  if(flag == 1)
    interrupts_on();
    return buffer;
  if(GDN_BUF has msg)
    interrupts_on();
    return $cgni;
  interrupts_on();
}
```

Notice that interrupts are continuously turned on and off, and that GDN_BUF is polled in a non-blocking fashion. In this manner, the system is able to service MDN interrupts while in the process of waiting for a "user" message, thus averting the above deadlock condition.

5.1.2 Tile-to-Controller Communication

All messages sent from user-tiles to the controller are formatted as follows:

```
[ OPCODE.TILE_NUM ] [ 1st argument] ... [ nth argument ]
```

Since the first word is an opcode, the controller knows exactly how many arguments to read off the GDN for the particular request. It is also unnecessary for this message to be prefixed by the word 0xF000 since the controller isn't a registered tile in the cache-coherence system, and will thus never receive any "system" messages.

5.2 Shared Memory Allocation

Uniprocessor systems have memory allocation routines that manage memory resources on the heap. The most common implementation of malloc maintains a free-list that tracks unallocated chunks of memory. When a new chunk of memory needs to be allocated, the malloc routine traverses the free list and finds a chunk that is either the best-fit or first-fit (the former of which has lesser memory fragmentation). In a distributed system, besides this basic level of resource allocation, the ability to allocate the same chunk of memory among distinct processors is required. To implement such a system, each allocated chunk needs to be associated with a shared memory identifier, so that subsequent malloc calls with the same identifier return the same chunk.

`void* shmalloc(int shmid, int size)` If another process has not called this routine with the same shmid, then a newly zeroed out chunk of size size is allocated, and its address is returned. Otherwise, the address of an already allocated chunk associated with this shmid is returned. Note that if the size is not a multiple of 32, it is rounded up to the next multiple so that the block is cache-aligned.

`void* shumalloc(int size)` a newly zeroed out chunk of size size is allocated, and its address is returned.

`void shfree(void *addr)` when this function is invoked by all owners of addr, the chunk is returned to the free-list, so that subsequent calls to shmalloc may utilize this newly freed memory.

0x0	SIZE OF PREVIOUS CHUNK
0x4	SIZE
0x8	BIT-VECTOR OF OWNERS
0xC	SHARED MEMORY IDENTIFIER
0x10	UNUSED
...	...
0x20	DATA WORD
...	...
0x20 + $\lfloor$SIZE/32$\rfloor$*32	...

Figure 5-5: Memory Footprint of Allocated Chunk

0x0	SIZE OF PREVIOUS CHUNK
0x4	SIZE
0x8	PTR TO NEXT FREE CHUNK
0xC	PTR TO PREVIOUS FREE CHUNK
0x10	UNUSED
...	...
0x20	FREE DATA WORD
...	...
0x20 + $\lfloor$SIZE/32$\rfloor$*32	...

Figure 5-6: Memory Footprint of Free Chunk

`void shm_init()` this function needs to be called once, prior to invoking any shared memory functions. It is responsible for initializing the free-list.

To implement these various resource allocation routines, each chunk (both free and allocated) has a header containing various metadata. The headers of both types of chunks store the chunk size in bytes, as well as the size of the previous chunk. In addition, a free chunk stores pointers to the next and previous free chunks, whereas an allocated chunk stores the shmid and a bit vector of all owners of the shared chunk [1]. It is important to note that these headers are 32 bytes, and thus occupy an entire cache-line. In this manner, there is no cache contention between the header and the actual data following it.

[1]larger RAW fabrics will require some minor modification: more storage will be necessary for this bit-vector

5.3 Parallelizing Primitives

Distributed shared memory algorithms require basic synchronization primitives so that multiple processes may access and modify shared memory in a sequential manner. Furthermore, synchronization primitives allow programmers to place restrictions on program flow. Support for these primitives has been provided in the form of locks and barriers. The primitives `lock_init` and `barrier` are blocking, since a certain set of conditions needs to be satisfied before the controller acknowledges the request.

5.3.1 Locks

Queueing locks were implemented by having a lock manager executing on the shared memory controller. Each lock is designated by a 32-bit lockid. The controller maintains a hashtable whose keys are lockid's, and whose values are bitvectors corresponding to user requests for the particular lock. When a user-tile attempts to acquire a lock, its request is noted in the corresponding bitvector. Only when the bitvector is empty, or when this particular bit has been chosen at random after another tile has released the lock, will the controller message the tile signalling that the lock has been acquired.

`void lock_set(int lockid)` this function blocks until the lock has been acquired

`void lock_release(int lockid)` the lock associated with lockid is released, so that other processes that have requested this lock may acquire it.

`int lock_test(int lockid)` this non-blocking function returns 1 if the lock is currently held by another process, and 0 otherwise.

5.3.2 Barriers

Barriers are important in maintaining the program flow of distributed shared memory algorithms. It ensures that all processes have reached a certain point in the program execution, before proceeding any further. A barrier manager running on the shared memory controller records which user-tiles have hit the barrier, and once all registered tiles have done so, sends an acknowledgement to each tile. Each tile in turn waits for the ack from the controller, before exiting the barrier function.

`barrier_init(int barrierID)` this function must be called once, prior to executing `barrier()` on any tile. BarrierID is a bit-vector specifying which tiles will be participating in the barrier process.

`barrier(void)` this function blocks until all user-tiles specified by barrier_id have executed `barrier()`.

5.4 Low-level Issues

The following section examines the critical details pertaining to both the development process and the underlying runtime environment.

5.4.1 Runtime Environment Modifications

Since an address is declared shared if bits 25-26 are high, the upper 25% of an individual tile's address space is reserved for shared memory. Therefore, the C runtime environment needs to be modified to reflect this new upper bound on a tile's address space. In this manner, the stack pointer will be initialized to just below the shared region (e.g. Tile 0 initializes its stack ptr to 0x05FFFFFF), and the local `malloc` routine will be correctly bounded. Figure 5-7 shows the new memory footprint for an individual tile.

5.4.2 Binary Rewriter & Programmer Restrictions

The shared memory system currently dictates certain programmer responsibilities when designing applications. However, these responsibilities have been relaxed when

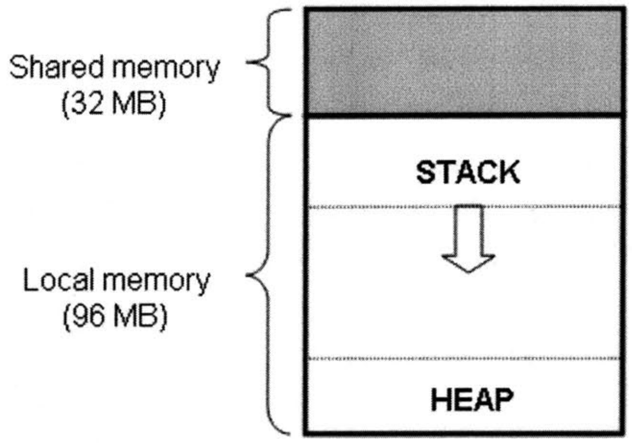


Figure 5-7: Memory Layout of an Individual Tile

developing in C, since a custom binary rewriter modifies the compiled object code to ensure correctness.

LW hazards The two instructions following a LW must be repeatable. This restriction is necessary because of the manner in which deadlocks are handled; after a tile has received spoofed data, the pending MDN interrupt only fires two cycles later. Thus, when the PC is reset to the offending LW instruction, the two instructions following it must be repeated. Furthermore, the address and data registers of the LW instruction must be distinct, so that the interrupt handler may invalidate the spoof data. The binary rewriter ensures this restriction by inserting 2 NOP's after every LW instruction, and properly rearrange the code when it encounters load-words of the form `LW` X, Y`(x)`.

SW hazards When a cache line is dirtied after a SW instruction, the event counter interrupt may take up to 6 cycles to fire. The event counter handler then examines the offending SW instruction, extracting the address and data from the proper registers. Thus, the following restrictions are required:

1. no branches may be taken in the 6 cycles following a SW

2. the SW address and data registers must not be modified in the 6 cycles following

the SW

The binary rewriter ensures these restrictions are met by inserting 6 NOP's after every SW instruction.

5.5 Designing Shared Memory Applications

Distributed shared memory applications tend to follow a general design paradigm. The global data structures at the core of all shared memory applications are usually initialized by a single-tile, and tiles synchronize their manipulation of these data structures through locks and barriers. Although there is a large degree of flexibility in designing shared memory applications, the application designer must be sure to adhere to certain program restrictions as well.

Programmer Restrictions It is important to note that shared memory applications that utilize the static or dynamic networks violate the shared memory design paradigm, as they are bypassing their natural communication pathway. Not only is it conceptually incorrect for user-tiles to initiate communication via the network, but it is also forbidden in the current cache-coherence system. The low-level system requires exclusive ownership over the MDN, GDN, and static networks; the simple introduction of the shared memory controller resulted in a plethora of complications related to network multiplexing and deadlock scenarios. However, it is conceivable that future support for GDN communication be made possible, with slight modification to the `gdn_user_receive` and `gdn_user_send` functions.

A producer-consumer model can be used to simulate point-to-point messages between user-tiles, as detailed in Figure 5-8. In this model, a shared memory circular buffer is read from and written to by different user-tiles. The producer is responsible for writing into the buffer, and updating a tail pointer that points to the first unused word. Similarly, the consumer reads from the same buffer, and increments the head pointer which points to the first unread message. Since this mechanism allows tiles to asynchronously send messages to one another, usage of the GDN network becomes

```
void send(char c) {
  *tail = c;
  tail = (tail + 1) % SIZE;
}

char receive() {
  char c;
  while(head == tail) { }
  c = *head;
  head = (head + 1) % SIZE;
  return c;
}
```

Figure 5-8: Shared Memory Producer-Consumer Model

unnecessary.

5.5.1 General Programming Model

The general paradigm for designing shared memory systems employs a single-tile (e.g. Tile 0) that initializes all global data structures used by the application. All tiles that intend to manipulate these global data structures must make a `shmalloc` call with the corresponding shared memory identifier. To synchronize which identifier is used for what data structure, a local variable can keep track of the next unused identifier, and should be incremented whenever a call is made to `shmalloc`. Care must be taken so as to ensure that all tiles are passing the same SHMID for a particular data structure. The following segment of code depicts the allocation of shared data structures.

```
    int SHMID = 0;
    struct global_1 *ptr1 = shmalloc(SHMID++,sizeof(struct global_1));
    struct global_2 *ptr2 = shmalloc(SHMID++,sizeof(struct global_2));
    struct global_3 *ptr3 = shmalloc(SHMID++, sizeof(struct global_3));
```

After these data structures are allocated, Tile 0 is responsible for properly initializing them while the other tiles stall at a `barrier`. If the member field of a particular global structure is a primitive (e.g. `int, float, char, double`), then Tile 0 directly writes to the field. If, however, the member field is a pointer to shared memory, then Tile 0 invokes `shumalloc`. Recall that `shumalloc` does not require a shared memory identifier as an argument, since it always returns a newly allocated location in global memory. Furthermore, note that although this is a new global data structure being allocated, the other tiles do not need to invoke `shmalloc` as well. This is because a reference to this new location will be stored in the original global data structure, which is accessible by all tiles. The following segment of code depicts the initialization of data structures, as well as the invocation of `shumalloc`.

```
if(TILE_NUM == 0) {
  global_struct->integer1 = int_var;
  global_struct->float1 = float_var;
  global_struct->internal_struct =
       (struct internal*) shumalloc(sizeof(struct internal_struct));
}
barrier();
```

Now, the motivation for `shumalloc` is more clear; `shmalloc` is invoked by ALL TILES for top-level global data structures, while `shumalloc` is invoked by individual tiles for nested structures.

The remaining tiles will only resume execution after Tile 0 has initialized all data structures and subsequently hit its barrier.

Locks Locks are necessary when tiles need to perform atomic transactions. For example, consider the case where 12 tiles attempt to obtain unique identification numbers from 1-12; the most straightforward implementation is for each tile to read a global integer, and subsequently increment it by 1. However, the read and write should be atomic, so the address needs to be locked during the duration of the transaction. In this case, the `lockid` argument to the lock_set function can simply be the address of the global variable itself. Infact, the `lockid` can almost always be the shared memory pointer pertaining to the atomic transaction; unlike the `shmalloc` function which requires the programmer to ensure all calls across tiles use the same identifier, the lock_set and lock_release arguments can be determined at run-time.

```
int *num = shmalloc(SHMID, sizeof(int));
lock_set(num);
myNum = *num
*num = *num + 1;
lock_release(num);
```

Barriers Barriers are necessary whenever the program flow is partitioned into separate phases. They ensure that a particular phase has been completed among all tiles, before proceeding to the next. Barriers, in essence, are a specialized mechanism to broadcast information to all tiles.

The Barnes-Hut particle simulation exemplifies the need for barriers in distributed shared memory applications (see Section 6.1.2). The particle-simulation application constantly proceeds in the following two phases:

1. insert all particles into a quad-tree based on their spatial positions.

2. calculate the new forces on each particle, and correspondingly recalculate their positions.

Only after all particles have been inserted in the quad-tree, can the application proceed to calculate the inter-particle forces. Thus, barriers are necessary to impose this restriction among all tiles.

Chapter 6

Applications & Performance Results

A suite of shared memory applications was developed for the purpose of obtaining a quantitative evaluation of the cache-coherence system. The various shared memory primitives developed greatly eased the programmability of these applications. One application for the system was coded from scratch, and was relatively straightforward to implement given the problem description. Five other applications were taken from Stanford's Parallel Applications for SHared Memory (SPLASH-2) [9], and were relatively effortless to port. The calls associated with locks and barriers had to be ported to our libraries, and the initialization of global data structures had to be explicitly executed on a single tile.

Shared memory applications that ranged from 2000-3000 lines of code only required 20-30 minutes to successfully port to the RAW architecture. The ease in portability attests to the power of the shared memory primitives developed. These primitives, along with the favorable system performance on a variety of applications, reveal that shared memory has become a practical alternative to message-passing.

6.1 Application Profiling

The shared memory applications developed were run on a varying number of tiles, to determine their scalability on the system. The cache size in the BTL simulator was increased to 512 kilobytes, so that cache evictions did not contribute to performance degradations. Furthermore, performance statistics were only collected after the first couple of iterations within the application, so as to reduce the effect of cold-starts. The uniprocessor statistics were collected by compiling the application with no shared memory support - the binary rewriter was not run on the compiled object code, and the original FPGA logic was used.

6.1.1 Jacobi Relaxation

Jacobi Relaxation is an iterative method that, given a set of boundary conditions, finds solutions to differential equations of the form $\nabla^2 A + B = 0$. The solution to the equation is found by repeatedly applying the following iterative step:

To determine the next value of a particular element, the value of the four neighboring cells need to be retrieved. The solution is achieved when the algorithm converges on a particular matrix.

Table 6.1: 72x72 Jacobi Relaxation

Tiles	Cycles	Speedup over 1-Tile	Speedup over Uniprocessor
Uniprocessor	2760773		
1	4132000		0.6681
4	1890000	2.1862	1.4607
8	1110635	3.7203	2.4857

Partitioning of Data

The $N \times N$ matrix is represented as a two-dimensional array in shared memory. Given P processors, each processor works on an $(N/P) \times (N/P)$ block of the matrix. Implicit communication between processors only occurs when computing the new value for a cell along the edge of a block. Since the values of all neighboring cells need to be retrieved, cells belonging to other processors must be accessed. Thus It should be evident that this algorithm exhibits high locality, since given a BxB block size, an individual processor can compute $(B-2) \times (B-2)$ entries from entirely local data. Table 6.1 shows that the application indeed scales well on a 72x72 matrix.

6.1.2 Barnes-Hut

This application simulates the interaction of N bodies in a gravitational system. Every body is modelled as a particle that exerts forces on all other bodies within the system. An iterative method is used to compute the net force on each particle, and to correspondingly update its position. If all forces are computed within the system, then the problem has a time complexity of $O(n^2)$. However, the barnes-hut algorithm requires only O(n*log n) complexity, and thus makes the simulation of large systems practical [7].

Table 6.2: Barnes-Hut 256 particle simulation

Tiles	Cycles	Speedup over 1-Tile	Speedup over Uniprocessor
Uniprocessor	38061464		
1	58064952	1	0.6554
2	33470724	1.7347	1.1371
4	19777125	2.9359	1.9245
6	16004382	3.6280	2.3781

The particular simulation developed operates in two dimensions, and uses a hierarchical quad-tree representation of space. The root node represents the entire space, and the tree is built by sub-dividing a cell into four children as soon as it contains more than 1 particle. Thus, the leaves of the tree represent the actual particles, and the tree is adaptive in that it extends deeper in regions that have higher particle densities.

To compute the force of an individual particle, the tree is traversed starting at the root node. The algorithm proceeds as follows: if the center of mass of the cell is far enough from the particle, then the cell is approximated as a single particle; otherwise, the individual subcells are examined. In this manner, it is unnecessary to calculate all pair-wise interactions. Most of the program execution is spent in this partial traversal of this tree.

Data Structures

Two arrays are maintained for the purpose of representing the underlying tree: one array contains the actual leaves, while the other consists of the internal cells. Ownership of these particles is assigned to processors to ensure both load balancing and data locality. The former is achieved by calculating an estimated work count for each particle, and assigning an equal amount of total work to each processor. The latter is achieved by exploiting the physical locality of particles; partitions are physically contiguous.

Program Structure

The barnes-hut algorithm continually executes the following loop. The majority of time is spent in step 4, where the forces are computed by partially traversing the tree.

1. Construct octree from particles
2. Find center of mass of cells
3. Partition particles among processors
4. Compute forces on particles
5. Advance particle positions and velocities

6.1.3 LU Factorization

The LU algorithm factors a dense $N \times N$ matrix into the product of upper and lower triangular matrices. The $N \times N$ matrix is divided into $(N/B) * (N/B)$ blocks, and ownership is assigned to processors using a 2-D scatter decomposition [9].

Table 6.3: 128x128 LU Factorization

Tiles	Cycles	Speedup over 1-Tile	Speedup over Uniprocessor
Uniprocessor	60889477.08		1
1	65472556		.93
2	39269280	1.6673	1.5506
4	22291044	2.9372	2.7312
8	14705762	4.4522	4.1405

6.1.4 FFT

The FFT algorithm is a complex 1-D version of the kernel presented in [9]. The n complex data points to be transformed are stored in a $\sqrt{n}x\sqrt{n}$ matrix. Each processor is assigned a contiguous submatrix of size $\sqrt{n/p} * \sqrt{n/p}$, and all-to-all communication occurs in three matrix transpose systems [9].

Table 6.4: 1024-pt FFT Transformation

Tiles	Cycles	Speedup over 1-Tile	Speedup over Uniprocessor
Uniprocessor	2190000		
1	3296183		0.6644
2	2011505	1.6386	1.0887
4	1181020	2.7909	1.8543
8	865077	3.8102	2.5315

6.1.5 Ocean

The Ocean application simulates large-scale ocean movements based on eddy and boundary currents. Wind provides the external force, and the frictional effects from the ocean walls and floor are included. The simulation is run until the eddies and mean ocean flow stabilize.

The ocean is simulated as a square grid of NxN points. During each timestep of the simulation, a set of spatial partial differential equations are setup and solved. The continuous functions are first transformed into finite difference equations, which are then solved on two-dimensional grids representing horizontal slices of the ocean basin.

The ocean basin is divided into physical subdomains that are assigned to each processor. The algorithm exhibits high locality, in that each processor only accesses grid points within its subdomain and those of its neighbors. Furthermore, the only data shared lies along the boundaries of these subdomains. Thus, in a similar fashion to Jacobi Relaxation, the communication-to-computation ratio can be arbitrarily decreased by increasing the domain of the problem.

Future Work

Although the application has been successfully ported, the program does not fit within the instruction cache provided in BTL. Software instruction caching was therefore attempted, but compilation failed due to unknown reasons. Further work is necessary to bring this application to working status.

6.1.6 Radix

The integer radix sort is based on an iterative method, and performs one iteration for each radix r digit. Each processor examines its assigned keys and computes a local histogram, which is subsequently accumulated into a global histogram. Next, the global histogram is passed on to each processor, so that the local keys can be permuted into a new array for the next iteration [9].

Future Work

The performance results for radix are unsatisfactory since the application fails to scale with an increasing number of tiles. This may be a result of the initially high communication-to-computation ratio required. Although larger data-sets may well be the solution to observing better scaling, they would require an inordinate amount of time to simulate on BTL. Thus, profiling this application on the hardware itself may be the only practical alternative.

6.1.7 Discussion of Results

There is a noticeable performance degradation when running an application on a single-tile with shared memory support, as opposed to a uniprocessor. This discrepancy arises even after 1. the cache size has been increased to prevent cache evictions, and 2. the effect of cold-start has been accounted for. The last two conditions imply that the single tile will never interface with the system-tiles. Thus, the degradation in performance arises as a sole result of the programmer restrictions imposed by the binary rewriter. The current rewriter is primitive in that it inserts 6 NOP's after every SW, resulting in a noticeable drop in processor utilization. A more sophisticated rewriter would actually examine the instructions following a SW in the compiled object code, and only insert NOP's if the restrictions in Section 5.4.2 are violated.

The speedups obtained for the various applications appear to initially linearly scale with the number of tiles. However, as the number of tiles increases, the marginal improvement in performance slightly decreases - this is a result of the increased communication-to-computation ratio. The data-sets used for the applications yielded a shared memory overhead of approximately 60%; that is, N tiles approximately had an N/2 speedup over a uniprocessor. If larger data-sets were used, the shared memory overhead could be reduced to an arbitrarily small amount, due to the decreased communication-to-computation ratio. However, since these applications were simulated on BTL, large data-sets would require an exorbitant amount of time to execute.

The performance results make it clear that shared memory implementations of parallel applications are practical on the RAW architecture. And since shared memory applications tend to have greater programmability than their message-passing counterparts, the motivation for using shared memory is even more pronounced.

Chapter 7

Conclusion

This research served to expand the capabilities of the low-level cache coherence system previously provided in [5] along three dimensions: (1) correctness, (2) programmability, and (3) performance. Although the system previously designed was noticeably the first of its kind, it lacked the maturity to make shared memory a compelling design paradigm for application development on the RAW architecture. By addressing the aforementioned issues, this thesis successfully demonstrates that shared memory applications are practical on RAW.

The successful execution of various load-intensive applications suggests that many issues relating to the correctness of the low-level coherence protocol have been addressed. The initial protocol was sound in its design, but the system had to be tailored to be more aware of the underlying hardware, as well as accommodative to various race conditions.

The higher-level of shared memory support, provided by the shared memory controller and library of shared memory primitives, is evident in the ease by which SPLASH applications [9] were ported to the system. Low-level coherence mechanisms and programmer restrictions are now abstracted away, paving the way for rapid application development.

The performance results suggest that the system scales very well with an increasing number of tiles, and that shared memory applications provide an unmistakable benefit over uniprocessor implementations. Furthermore, most of the applications profiled will exhibit more optimal scaling when larger data sets are used.

This thesis has successfully shown that the increased programmability and performance provided by shared memory makes it a compelling design paradigm on RAW. The mature level of support has made shared memory ripe for comparison with other systems on RAW, such as the software-based message passing system [6]. The successful deployment of the system should highlight the power and flexibility of the underlying RAW architecture.

Appendix A

System Code

All code pertaining to the shared memory controller, shared memory libraries, and low-level cache coherence protocol are provided below.

A.1 Shared Memory Controller Code

The following code is loaded onto the shared memory controller. The controller continually polls the GDN to check for incoming requests, then performs the appropriate action, and finally issues GDN responses if necessary. It invokes various functions provided by the dshm, dlock, and dbarrier libraries.

```
#include "shm_constants.h"
#include "dshm.h"
#include "dlock.h"
#include "dbarrier.h"
#include "raw.h"
#include "raw_user.h"

void begin(void) {

  int r, tile, lock, key, size, addr, header;

  // initialise shared memory address space
  shm_init();
  // initialise locks
```

```
lock_init(301);

while(1) {
  // receive command in OPCODE | TILE_NUM format
  r = gdn_receive();

  // extract opcode and tile
  tile = r & 0x7F;
  header = construct_dyn_hdr(0, 1, 0, 0, 0, (tile&0xC)>>2, tile&0x3);

  switch(r & 0xF00) {
    case LOCK_RELEASE_OP:
  lock = gdn_receive();
  lock_release(tile, lock);
  break;
    case LOCK_SET_OP:
  lock = gdn_receive();
  lock_set(tile, lock);
  break;
    case LOCK_TEST_OP:
  lock = gdn_receive();
  lock_test(tile, lock);
  break;
    case MALLOC_OP:
  key = gdn_receive();
  size = gdn_receive();

  // add "1" to designate shared
  addr = shmalloc(key, size, tile) + 1;
  // send response with addr back
  raw_test_pass_reg(addr);
  gdn_user_send(tile, addr);

  break;
    case FREE_OP:
  addr = gdn_receive();
  shfree(addr, tile);
  break;
    case BARRIER_OP:
  barrier(tile);
  break;

    case BARRIER_INIT_OP:
  key = gdn_receive();
  barrier_init(key);
```

```
    break;

      default:
    break;
    }
  }
  return;
}
```

A.1.1 DShm Library

This code is utilized by the shared memory controller to initialize, allocate, and de-allocate shared memory.

```
#include "dshm.h"
#include "raw.h"

struct malloc_chunk {
  int size_prev;
  int size;
  struct malloc_chunk *fd_owners;
  struct malloc_chunk *bk_shmid;
};


typedef struct malloc_chunk malloc_chunk;

#define PREV_INUSE 0x1 #define HEAD 0x6000000 #define HDR_SIZE 32
#define SIZE_SZ 4

#define mem2chunk(mem) ((malloc_chunk*)((char*)(mem) - HDR_SIZE))
#define chunk2mem(p) ((char*)(((char *)(p)) + HDR_SIZE))

#define is_head(p) ((((int)p) & 0x6FFFFFF) == 0x6000000) #define
is_tail(p) ((((int) (((char *)p) + (p->size & ~PREV_INUSE))) &
0x7FFFFFF) == 0x0000000)

#define next_chunk(p) ((malloc_chunk*)(((char*) p) + (p->size &
~PREV_INUSE) + is_tail(p)*0x6000000))
```

```
#define prev_chunk(p) \ ((malloc_chunk*)(((char*) p) -
(p->size_prev) - is_head(p)*0x6000000))

#define inuse(p)\ ((((malloc_chunk*)(((char*)(p))+((p)->size &
~PREV_INUSE) + is_tail(p)*0x6000000))->size) & PREV_INUSE)

/* extract inuse bit of previous chunk */

#define prev_inuse(p)  ((p)->size & PREV_INUSE)


/* set/clear chunk as in use without otherwise disturbing */

#define set_inuse(p)\ ((malloc_chunk*)(((char*)(p)) + ((p)->size &
~PREV_INUSE) + is_tail(p)*0x6000000))->size |= PREV_INUSE

#define clear_inuse(p)\ ((malloc_chunk*)(((char*)(p)) + ((p)->size
& ~PREV_INUSE) + is_tail(p)*0x6000000))->size &= ~(PREV_INUSE)

/* check/set/clear inuse bits in known places */

#define inuse_bit_at_offset(p, s)\ (((malloc_chunk*)(((char*)(p))
+ (s)))->size & PREV_INUSE)

#define set_inuse_bit_at_offset(p, s)\
(((malloc_chunk*)(((char*)(p)) + (s)))->size |= PREV_INUSE)

#define clear_inuse_bit_at_offset(p, s)\
(((malloc_chunk*)(((char*)(p)) + (s)))->size &= ~(PREV_INUSE))


#define chunksize(p) (p->size & ~PREV_INUSE)


#define set_head(p, s) ((p)->size = s)

/* Set size at head, without disturbing its use bit */

#define set_head_size(p, s)   ((p)->size = (((p)->size &
PREV_INUSE) | (s)))


#define set_foot(p, s)\ (((malloc_chunk*)(((char*)(p)) + (s) +
is_tail(p)*0x6000000))->size_prev = (s))
```

```
#define chunk_at_offset(p,s) ((malloc_chunk *)(((char*)(p)) +
(s)))

#define unlink(p, bk, fd) \ {                          \
  bk = p->bk_shmid;   \
  fd = p->fd_owners;  \
  fd->bk_shmid = bk;  \
  bk->fd_owners = fd; \
} \

#define insert(p, bk, fd) \ {                          \
  bk->fd_owners = p;  \
  p->bk_shmid = bk;   \
  fd->bk_shmid = p;   \
  p->fd_owners = fd;  \
} \

malloc_chunk *top;

void check_inuse_chunk(malloc_chunk *p, int tile) {
  int mask = (1<<tile);
  //make sure it is a shared mem address
  //assert(((((int)p)>>26) & 1) && ((((int)p)>>25) & 1));
  //assert(inuse(p));
  //assert(((int)p->fd_owners) & mask);
}

void clearchunk(malloc_chunk *p) {
  int *i;
  int *mem = (int *) chunk2mem(p);
  int *mem_end = (int *)((int) mem + chunksize(p) - 32);

  for(i = mem; i < mem_end; i++) {
    *i = 0;
  }
  // now, flush & invalidate the data
  // (to eliminate incoherence w/ this tile)
  raw_flush_cache_range(mem, chunksize(p) - 32);
  raw_invalidate_cache_range(mem, chunksize(p) - 32);
}

// initialise free-list
void shm_init() {
  int i, addr;
  malloc_chunk *tmp, *p;

  tmp = 0x6000000+15*0x8000000;
  //create doubly-linked circular free-list
  for(i=0; i<16; i++) {
    addr = 0x6000000+i*0x8000000;
    p = (malloc_chunk*)addr;

    p->size = 0x2000000;
    p->size_prev = 0x2000000;

    //back link
    p->bk_shmid = tmp;
    //forward link
    p->bk_shmid->fd_owners = p;

    tmp = p;
  }

  top = (malloc_chunk*) (0x6000000+15*0x8000000);
  top->fd_owners = 0x6000000;
  return;
}

void *shfree(const void *shmaddr, int tile) {
  malloc_chunk *p, *next, *prev, *bk, *fd;
  int sizeprev, sz, mask;

  if(shmaddr == 0)
    return;

  //MALLOC_LOCK

  p = mem2chunk(shmaddr);
  check_inuse_chunk(p, tile);

  sz = p->size & ~PREV_INUSE;
  next = next_chunk(p);

  if(next == top && !is_tail(p)) {
    //consolidate with top
    sz += next->size & ~PREV_INUSE;

    if(!(p->size & PREV_INUSE) && !is_head(p)) {
      //first consolidate backwards
      sizeprev = p->size_prev;
      p = chunk_at_offset(p, -sizeprev);
      unlink(p, bk, fd);
      sz += sizeprev;
    }

    unlink(top, bk, fd);
    insert(p, top->bk_shmid, top->fd_owners);
    set_head_size(p, sz);
    // can't call set_foot because "top" points to
    // ending region of shared mem
    top = p;

    return;
  } // if(next == top)

  //clear inuse bit
  clear_inuse(p);

  if(!(p->size & PREV_INUSE) && !is_head(p)) {
    // consolidate backwards
    sizeprev = p->size_prev;
    p = chunk_at_offset(p, -sizeprev);
    sz+= sizeprev;
    unlink(p, bk, fd);
  }

  // must prematurely set_head_size so the following call
  // to 'is_tail' executes correctly
  set_head_size(p, sz);

  if(!inuse(next) && !is_tail(p)) {
    // consolidate forwards
    sz+= next->size & ~PREV_INUSE;
    unlink(next, bk, fd);
  }

  set_head_size(p, sz);
  set_foot(p, sz);
  insert(p, top, top->fd_owners);

}

// returns the shmid of a newly allocated block
```

```
int shmalloc(int key, int size, int tile) {
  malloc_chunk *p, *q;
  malloc_chunk *fd;
  malloc_chunk *bk;

  // add malloc overhead to obtain actual size, and align
  // to cache boundary
  if(size % 32 != 0)
    size += 32 - size % 32;
  size += HDR_SIZE;

  // add random spacer, for better load-balancing
  size += (raw_get_status_CYCLE_LO() % 4) * 32;

  p = (malloc_chunk *) HEAD;

  while(p!= top && (!inuse(p) || (inuse(p) && (p->bk_shmid != key)))) {
    p = next_chunk(p);
  }

  if(key != -1 && p != top && inuse(p) && p->bk_shmid == key) {
    // this key has already been allocated, so add it to 'owners'
    p->fd_owners = ((int) p->fd_owners) | 1<<tile;
    return chunk2mem(p);
  }

  else {
    // actually allocate the memory by traversing the free-list.

    // if chunksize(p) == size then we have an exact fit.
    // otherwise, chunksize(p) must be greater than size+MIN_SIZE,
    // because we must create a new free block of MIN_SIZE.

    for(p = top->fd_owners ; ; p = p->fd_owners) {
      // exact fit
      if((chunksize(p) == size) && p != top) {
    unlink(p, bk, fd);
    p->bk_shmid = key;
    p->fd_owners = 1 << tile;
    set_inuse(p);

    clearchunk(p);
    return chunk2mem(p);
      }
      // remainder present
      else if((chunksize(p) > size)) {
    q = chunk_at_offset(p, size);

    // if we are allocating in the top block, then top
    // moves to the remainder of whats left.

    if(p == top)
      top = q;

    set_head(q, chunksize(p) - size);
    set_foot(q, chunksize(p) - size);

    set_head_size(p, size);
    set_foot(p, size);
    clear_inuse(q);
    set_inuse(p);

    unlink(p, bk, fd);
    insert(q, p->bk_shmid, p->fd_owners);

    p->bk_shmid = key;
    p->fd_owners = 1 << tile;

    clearchunk(p);

    return chunk2mem(p);
      }
      if(p == top) break;
    } // for
  }
  //malloc failed
  while(1) { }
  return 0;
}
```

A.1.2 DLock Library

This code is utilized by the shared memory controller to provide locking functionality for user-tiles.

```
#include "dlock.h"
```

```
#include "raw.h"
#include "raw_user.h"

#include "hashtable.h"
#include "raw.h"

hashtable *lockTable;

void lock_init(int num_locks) {
  lockTable = hashtable_construct(num_locks);
}

void lock_release(int tile, int lock) {
  int n, i, val;

  // mask out the releasing user
  val = hashtable_get_key(lockTable, lock);
  val &= ~(1<<tile);
  hashtable_set_key(lockTable, lock, val);

  // randomly choose a pending request, by examining the
  // number of requests, and comparing it to the cycle count

  if(raw_popc(val) == 0) return;

  // n is a random # between 1 and total_tiles
  n = (raw_get_status_CYCLE_LO() % raw_popc(val)) + 1;

  i = -1;
  while(n>0) {
    i++;
    if(val & (1 << i))
      n--;
  }

  // send positive acknowledgement of 1
  gdn_user_send(i, 1);
  return;
}

void lock_set(int tile, int lock) {
  int header;
  int val;

  val = hashtable_get_key(lockTable, lock);
  if(val == 0) {
    gdn_user_send(tile, 1);
  }
  // OR request in
  hashtable_set_key(lockTable, lock, val | 1 << tile);
}

void lock_test(int tile, int lock) {
  int header;
  int val = hashtable_get_key(lockTable, lock);

  gdn_user_send(tile, val == 1);

  // OR request in
  hashtable_set_key(lockTable, lock, val | 1 << tile);
}
```

A.1.3 DBarrier Library

This code is utilized by the shared memory controller to provide barrier synchronization for user-tiles.

```
#include "barrier.h"
#include "raw_user.h"
#include "raw.h"

int init = 0;

// bit vector of all tiles currently at barrier
int barrier_hit = 0;

// bit vector of all tiles using barrier
int barrier_mask = 0x7777;


void barrier_release() {
  int i;
  for(i=0; i<32; i++) {
    if(barrier_hit & (1 << i)) {
      gdn_user_send(i, 1);
    }
  }
  barrier_hit = 0;
}

// sets the pool of tiles in the barrier
void barrier_init(int bmask) {
  barrier_mask = bmask;
  init = 1;

  if((barrier_hit & barrier_mask) == barrier_mask)
    barrier_release();
}

void barrier(int tile) {
  barrier_hit |= 1<<tile;
  if(((barrier_hit & barrier_mask) == barrier_mask) && init)
    barrier_release();
}
```

A.2 Raw_User Library

Because the GDN is now multiplexed between "user" and "system" messages, the following functions are utilized by user-tiles whenever they need to read/write to the GDN.

```
#include "raw_user.h"
#include "raw.h"

int gdn_user_flag = 0; int gdn_user_msg;

int gdn_user_receive() {
  int r, i;

  while(1) {
    raw_interrupts_off();
    if(gdn_user_flag) {
      gdn_user_flag = 0;
      r = gdn_user_msg;
      break;
    }
    r = raw_get_status_GDN_BUF();

    // check if GDN message in buffer, and if
    // MDN-int not pending
    if(((r>>5)&0x7) && !(raw_get_status_EX_BITS() & 0x8)) {

      // read initial opcode
      gdn_receive();
      // read actual data
      r = gdn_receive();
      break;
    }
    raw_interrupts_on();
    }
  raw_interrupts_on();
  return r;
}

void gdn_user_send(int tile, int data) {
  int hdr = construct_dyn_hdr(0, 2, 0, 0, 0,
                              (tile&0xC)>>2, tile&0x3);
  gdn_send(hdr);

  //opcode for GDN user message
  //(all sys->user GDN messages have bits 12-15 cleared)

  gdn_send(0xF000);
  gdn_send(data);
}
```

A.3 User Libraries

The libraries provided below provided higher-level shared memory support, and should be used by application developers.

A.3.1 Lock Library

The following are the lock routines called by user-tiles.

```
#include "lock.h"
#include "shm_constants.h"
#include "raw.h"
#include "raw_user.h"

static int tile = -1;

void lock_set(int key) {
  // send message to shmlockmanager
  int r, cycle, header;

  cycle = raw_get_status_CYCLE_LO();
  header = construct_dyn_hdr(0, /*length*/ 2, 0, /*row*/ 0,
                /*col*/0, CTRL_ROW, CTRL_COL);
  if(tile == -1)
    tile = raw_get_tile_num();

  raw_interrupts_off();
  __asm__ volatile("nop
                  nop
                  nop");
  gdn_send(header);
  gdn_send(LOCK_SET_OP | tile);
  gdn_send(key);
  raw_interrupts_on();

  r = gdn_user_receive();

}

void lock_release(int key) {
  // send message to shmlockmanager
  int header = construct_dyn_hdr(0, /*length*/ 2, 0, /*row*/ 0,
                /*col*/0, CTRL_ROW, CTRL_COL);
  if(tile == -1)
    tile = raw_get_tile_num();

  raw_interrupts_off();
  __asm__ volatile("nop
                  nop
                  nop");
  gdn_send(header);
  gdn_send(LOCK_RELEASE_OP | tile);
  gdn_send(key);
  raw_interrupts_on();
}

int lock_test(int key) {
  int r;
  // send message to shmlockmanager
  int header = construct_dyn_hdr(0, /*length*/ 2, 0, /*row*/ 0,
                /*col*/0, CTRL_ROW, CTRL_COL);
  if(tile == -1)
    tile = raw_get_tile_num();

  raw_interrupts_off();
  __asm__ volatile("nop
                  nop");
  gdn_send(header);
  gdn_send(LOCK_TEST_OP | tile);
  gdn_send(key);
  raw_interrupts_on();

  r = gdn_user_receive();

  return r;
}
```

A.3.2 Barrier Library

The following are the barrier routines invoked by user-tiles

```
#include "shm_constants.h"
#include "barrier.h"
#include "raw.h"
#include "raw_user.h"

static int tile = -1;

void barrier() {
  int r, header;
  header = construct_dyn_hdr(0, 1, 0, 0, 0, CTRL_ROW, CTRL_COL);

  if(tile == -1)
    tile = raw_get_tile_num();
  raw_interrupts_off();
  __asm__ volatile("nop
                    nop
                    nop");
  gdn_send(header);
  gdn_send(BARRIER_OP | tile);
  raw_interrupts_on();

  r = gdn_user_receive();
}

void barrier_init(int bmask) {
  int header = construct_dyn_hdr(0, 2, 0, 0, 0, CTRL_ROW, CTRL_COL);
  if(tile == -1)
    tile = raw_get_tile_num();

  raw_interrupts_off();
  __asm__ volatile("nop
                    nop
                    nop");
  gdn_send(header);
  gdn_send(BARRIER_INIT_OP | tile);
  gdn_send(bmask);
  raw_interrupts_on();
}
```

A.3.3 Shared Memory Allocation Library

The following are the shared malloc routines invoked by user-tiles.

```
#include "shm.h"
#include "shm_constants.h"
#include "raw.h"
#include "raw_user.h"

static int tile = -1;

int shmalloc(int key, int size) {
  // send message to shmlockmanager
  int r, bit;
  int header = construct_dyn_hdr(0, /*length*/ 3, 0, /*row*/ 0,
                  /*col*/0, CTRL_ROW, CTRL_COL);
  if(tile == -1)
    tile = raw_get_tile_num();

  raw_interrupts_off();
  __asm__ volatile("nop
                  nop
                  nop");

  gdn_send(header);
  gdn_send(MALLOC_OP | tile);
  gdn_send(key);
  gdn_send(size);
  raw_interrupts_on();

  r = gdn_user_receive();
  return r;
}


int shumalloc(int size) {
  return shmalloc(-1, size);
}

int shfree(const void *shmaddr) {
  int r;
  int header = construct_dyn_hdr(0, 2, 0, 0, 0, CTRL_ROW, CTRL_COL);

  if(tile == -1)
    tile = raw_get_tile_num();

  // turn interrupts off
  raw_interrupts_off();
  __asm__ volatile("nop
                  nop
                  nop");
  gdn_send(header);
  gdn_send(FREE_OP | tile);
  gdn_send(shmaddr);

  // turn interrupts on
  raw_interrupts_on();
  return r;
}
```

A.4 Raw Code

This section contains the low-level assembly code involved in the directory-based cache coherence protocol. Both the system-tile code, and the user-tile interrupt handlers are provided.

A.4.1 Directory-Tile Code

The following code is loaded onto all system tiles, and contains the routines for directory and interrupt controller functionality.

```
// directory tile.  Part of a shared-memory implementation.
// Levente Jakab.
// 5/11/04 18.40
// Modified by Satish Ramaswamy
// 5/11/05

// number of cycles between an interrupt and a response at which point
// we declare deadlock.  Times four, because the cycle counts are always
// stored starting at bit 2.

//#define DEADLOCK_TRIGGER_CYCLES 80000
#define DEADLOCK_TRIGGER_CYCLES 20000

#define DI_LIST_OP 0x300
#define DI_HOT_OP 0x800

#define ID_DEAD 0x500
#define DI_DEAD 0x700
#define IU_HOT_OP 0x80

// differentiated from $16, since bit 7 is never on

#define STATIC_BUFFER_SIZE 512

// states:
// 0 - read (readers in bits 8-31)
// 2 - read lock (readers in 8-31, writer in 4-7)
// 3 - read lock bypass (readers in 8-31, writer in 4-7)
// 2 - exclusive (writer in 4-7)
// 3 - exclusive pending (writer in 4-7)

// read lock and exclusive are the same, except for the presence of
// reading tiles needing to be negotiated

#include "module_test.h"

    .text
    .align 2

    .global begin
.ent begin

// register convention (above and beyond hardware restrictions)
// $2 is swap space... usually a tile number or an address
// $3 is swap space... an address in state space
// $4 is swap space
// $5 is swap space... a test condition
// $6 is swap space... an iterator
// $7 is swap space... a tile number
// $8 is swap space
// $9 is swap space... a tile number iterator

// $12 contains the constant 0xFF
// $13 is the static network stall condition
// $14 is one-hot directory number
// $15 contains the number "256"
// $16 contains a U-D map.  Which D-tile wants to speak to which U-tile.
// $17 contains an interrupt header constant
// $18 contains the offset of ocreq_table
// $19 contains the STATIC BUFFER CONDITION
// $20 contains the number "1"
// $21 contains the number "-1"
// $22 contains the BASE_ADDRESS
// $23 contains the offset of pending_address_table
// **$28 contains the offset of pending_request_table
// $28 contains the U-I hot-map. Which user-tiles the I-tile wishes
//                               to hot request.
//   syntax: xxxxxxxUUU
// **$29 contains the number of pending addresses remaining
// **$30 is a linkage pointer to one-away routines.  It is also swap space.
// Be careful!
// $31 is the link register.  Must be careful with this one.

// init routine.  Called exactly once per execution, so we don't care
// what regs we use.
init_begin:

    // SCAFFOLD this overjump is bad

    // j reset_state_done

    // set up some things in RAM.  Address bit 31 is always zero.
    // Address bits 30-27 identify the owner of the RAM.  Bits
    // 26 and 25 define it as shared or not.

    // if bits 26 and 25 of an address are both 1, then the RAM is
    // shared.  Therefore, pages 6, 7, 14, 15.. 126, 127 (each page
    // is 2^16 bytes here) are all filled with state space.  We jump
    // from big table to big table (6/7, 14/15, ... 126/127),
    // clearing out all 2^17 bytes of data stored there.

    li $2, 0x400000         // the big jump from pageset to
                    // pageset

    li $3, BASE_ADDRESS     // the beginning.

    li $6, 16           // $6 is a counter

    li $4, 0x3f00000        // $4 now has the address of the
    addu $4, $3, $4         // final pageset 03f00000

reset_state_big_loop:
    addu $7, $4, $0         // $7 iterates over addresses
    li $5, (1 << 15)        // 2^19 addresses need to be
                    // cleared. scaffold 2^12
    addu $5, $4, $5         // starting with the current big

.global reset_state_small_loop      // jump start
reset_state_small_loop:

    sw $0, 0($7)
    addiu $7, $7,   4       // add a word

    // if we just cleared out what is in address $4 (the beginning
    // of the current big table) then we must fetch a new big
    // table.
    bne $7, $5, reset_state_small_loop

    subu $4, $4, $2         // subtract big jump
    subu $6, $6, 1

    // if this was the first pageset, we are done... else, we
    // are not.
    bne $6, $0, reset_state_big_loop

    // SCAFFOLD line label, remove

reset_state_done:

    // SCAFFOLD priming stuff that needs to be removed as well

    // load into local cache all the addresses we'll be dealing with
    // la $2, SCAFFOLD_ADDR
    // la $3, SCAFFOLD_ADDR+0x400000
    //
.global reset_scaffold_loop reset_scaffold_loop:

    //sw $0, 0($2)
    //addiu $2, $2, 4           // step to next cache line
    //bne $2, $3, reset_scaffold_loop

    // END SCAFFOLD priming

    // write into pending address table the first free entry
    la $2, pending_address_table
    addiu $2, $2, 40

    swsw $2, %lo(pending_address_table)($0)

    // initialize static_input_buffer
    la $2, static_input_buffer
    addiu $3, $2, 8
    swsw $3, 0($2)
    swsw $3, 4($2)

    or $19, $0, $0

    // reset static network stall
    or $13, $0, $0
    // reset U-D map
    li $16, 0x10000000

    // load constants
    li $12, 0xFF
    li $14, TILE_ONEHOT
    li $15, 256
    li $17, (0x00F00000 | (TILE_ROW << 3))
                    // constant meaning "interrupt"
                    // and the tile row, since each
                    // I-tile tracks only columns

    li $20, 1
    subu $21, $0, $20

    or $28, $0, $0
    // load addresses
    la $18, ocreq_table
    la $22, BASE_ADDRESS
    la $23, pending_address_table
// BUGFIX: no longer modifies $28
//  la $28, pending_request_table


// BUGFIX: no longer uses $29
    li $2, 51
    isw $2, %lo(pending_addresses_remaining)($0)
    //li $29, 51            // 64, off-by-one, minus
                    // number of user tiles is the
                    // max number of pending
                    // addresses we can handle

    j init_done



// static network routine.  Destroys every swap-space register.
static_network_begin:

    // if we do not have any outstanding OCREQ space, then we must
    // not attempt a static-network read.
    bne $13, $0, static_network_done

    // BUGFIX - process if buffer is non-empty
    beq $19, $1, static_network_process
    // check to see if there is anything on static network
        mfsr $4, SW_BUF1        // pending incoming data

    // note that we do not knock off any stray bits on the static
    // network since everything on the static network is coming in
    // from the east, so we save an instruction here.  Also, we may
    // reduce latency by checking for an early message (one that
    // just hit the switch), and by the time the BEQ is passed, it's
    // viable.  Nifty.

        beq $4, $0, static_network_done // no data?  oh well, goodbye
.global static_network_process static_network_process:
    ori $7, $0, 0x0A        // static word -> $10
    or $30, $0, $31
    jal static_network_buffer_read
    or $31, $0, $30

    ori $7, $0, 0x2         // read in the address into $2
    or $30, $0, $31
    jal static_network_buffer_read
    or $31, $0, $30


    srl $3, $2, 5
    sll $2, $3, 5           // truncate low 5 bits to zero

    srl $3, $3, 2           // for comparison purposes - $3
    sll $3, $3, 2           // has the short version, also
                // needed.

    addu $3, $22, $3        // cache line metadata address

static_network_load_status_loop:
    mfsr $11, SW_BUF1
    rrm $11, $11, 16, 0x1
    beq $11, $0, static_network_load_status

    // buffer the static messages on the east port
    or $30, $0, $31         // store the old linkage pointer
    jal static_network_buffer_write
    or $31, $0, $30         // and restore LP

    j static_network_load_status_loop
```

```
static_network_load_status:
    lw $4, 0($3)            // load the status of the line

    // now, make the lw-sw distinction.
    andi $6, $10, 0x8000

    bne $0, $6, static_network_sw

    // looks like this is a lw, so go forth and continue the lw

    andi $6, $4, 3          // load the state bits

    beq $6, $0, static_network_lw_read
    // zero implies none or read, so therefore nonzero is one of the
    // other states.

    // we must be in some state that is not read.  In this case, the
    // protocol is to request a writeback, if we have not done so
    // already.  (This is signified by the low bit of the state.)

    andi $6, $4, 1          // low bit is on?
    bne $6, $0, static_network_lw_writeback_set

    // we must request a writeback and also update the state as
    // either going from read-lock to read-lock-bypass, or from
    // exclusive to exclusive-pending.  This is done by setting the
    // low bit.

    ori $4, $4, 1           // set that bit
    sw $4, 0($3)            // and store back.

    // if we are in the read-lock state, then the writeback will
    // occur later, and we do not need to force it.

    srl $6, $4, 8           // any readers implies read-lock
    bne $6, $0, static_network_lw_writeback_set

    // Set up a writeback request.  $2 contains the address.  $9
    // must contain the tile number, which (given that we are not
    // in "read") is in bits 4-7 of the state vector.

    rrm $9, $4, 4, 0xF      // tile number
    li $11, 1           // request number 1 - coughup

    or $30, $0, $31         // store the old linkage pointer
    jal ocreq_begin         // make function call
    or $31, $0, $30         // and restore LP

static_network_lw_writeback_set:

    // now, deal with the requestor.  Convert his number into some
    // civilised form.

    rrm $8, $10, 3, 0xC     // $8 has yy00 of requestor
    rrmi $8, $10, 0, 0x3        // $8 has yyxx (tile number)

    or $6, $23, $0          // load the beginning of
                    // the pending-address table

    // at this point
    // $2 has the address in question
    // $3 is its address in state space
    // $4 is its state
    // $6 has the beginning of the pending address table
    // $8 has the requesting tile's number
    // $5, $6, $7, $10, $11 are swap space

    // here, what we do is check all the addresses in the pending
    // address table.  If we have a match, then we go to that one.
    // Otherwise, we dribble off the bottom and go to the next one.

    swlw $9, 0($6)          // this is the address of the
                    // first free entry in the table.

static_network_lw_address_loop:
    addiu $6, $6, 40        // add 40 to this entry
    swlw $7, 4($6)          // load the address there
    //BUGFIX: previous impl. failed when $7=$2 and $6=$9

    // new entry needs to be allocated
    beq $6, $9, static_network_lw_address_new

    // exact match, bail out
    beq $7, $2, static_network_lw_address_exact

    j static_network_lw_address_loop

static_network_lw_address_new:
    // must be a new entry.  Where we are now, plus 40, is the next
    // free entry.
    addiu $7, $6, 40
    swsw $7, 0($23)         // store this new past-end
                    // pointer

    swsw $2, 4($6)          // and our current block must
                    // have the new address

    swsw $0, 0($6)          // clear out any possible
                    // detritus control bits.

static_network_lw_address_exact:

    // at this point, $6 must contain a correct slot.

    sllv $7, $15, $8        // requestor bit mask
    swlw $5, 0($6)
    or $7, $7, $5           // note new requestor

    swsw $7, 0($6)          // header spot = 1<<(requestor+8)

    j static_network_done

.global static_network_lw_read static_network_lw_read:

    // we are in the "read" state, and someone attempted an "lw", so
    // all we really do is add him to the list of readers and give
    // him the cache line.  $2 contains the address, $10 the header
    // with the U-tile number.  $3 contains the cache-line metadata
    // corresponding to this address, and $4 the data at $3.

    // first, forge an MDN header so that we get the data from RAM.
    // send to ourselves, but route funny to the east

    li $11, CONSTRUCT_DYNAMIC_HEADER(4, 1, 0, (TILE_ROW >> 2), 3, 0, 3)
    rrmi $11, $2, 24, 0x60      // insert correct tile row
    or $cmno, $11, $0

    or $cmno, $2, $0        // send the address

    // and now, the MDN will send the line.  We must then forward
    // it along.  First, prepare an MDN header.

static_network_lw_read_sn_wait:
    mfsr $11, SW_BUF1
    rrm $11, $11, 16, 0x1
    beq $11, $0, static_network_lw_read_mdn_wait

    // buffer the static messages on the east port
    or $30, $0, $31         // store the old linkage pointer
    jal static_network_buffer_write
    or $31, $0, $30         // and restore LP

//BUGFIX: (actually, bug was introduced by shmcontroller on tile14)
.global static_network_lw_read_mdn_wait
static_network_lw_read_mdn_wait:
    mfsr $5, MDN_BUF
    rrm $5, $5, 5, 0x7
    beq $5, $0, static_network_lw_read_sn_wait

    or $cmno, $10, $0

    // and now read in the data words, and pass them back out
    // if static buffer to the EAST is full, the memory controller
    // may be stalled. Therefore, buffer the incoming messages.

    ori $6, $0, 8
.global static_network_lw_read_buffer
static_network_lw_read_buffer:

    mfsr $11, SW_BUF1
    rrm $11, $11, 16, 0x1
    beq $11, $0, static_network_lw_read_cmni

    // buffer the static messages on the east port
    or $30, $0, $31         // store the old linkage pointer
    jal static_network_buffer_write
    or $31, $0, $30         // and restore LP

    j static_network_lw_read_buffer

.global static_network_lw_read_cmni static_network_lw_read_cmni:
    or $cmno, $cmni, $0
    addiu $6, $6, -1
    bne $6, $0, static_network_lw_read_buffer

    // END SCAFFOLD

    // we have now sent the data to the new reader.  We have to
    // add it to the list of readers.
    rrm $8, $10, 3, 0xC     // $8 has yy00 of requestor
    rrmi $8, $10, 0, 0x3        // $8 has yyxx (tile number)

    sllv $8, $15, $8        // shift "256" over by tile
                    // number
    or $4, $4, $8           // set the bit
    sw $4, 0($3)            // and write it back

    // and goodbye
    j static_network_done

.global static_network_sw static_network_sw:

    // this is what happens when we've gotten a store-word.  This
    // could be in response to a coughup, or not.  In all possible
    // cases, the data being stored is the freshest line around, so
    // we should send it to any tiles stalling out waiting for the
    // read.  Also, we should write it back to main RAM.

    andi $8, $4, 1          // grab the low bit of the state
                    // to differentiate between
                    // pending and regular

    // if not pending, then that means there was no possibility of
    // extra writers, and no need to worry about patching.  Also,
    // there are no outstanding readers, so no need to worry about
    // that either.

    beq $8, $0, static_network_sw_nopatch_noreq

    // we must have either readers or patches.


    or $6, $23, $0          // load the beginning of
                    // the pending-address table

    // $2 contains the current transaction address
    // $3 the address of that cache lines state
    // $4 contains the entire state line
    // $6 has the beginning of the pending address table
    // $10 contains a MDN header corresponding to the writing tile
    // $5, $6, $7, $8, $9, $11 are swap space

    // We must find the entry in the pending address table
    // corresponding to our address.
    swlw $9, 0($6)          // this is the address of the
                    // first free entry in the table.

static_network_sw_address_loop:
    addiu $6, $6, 40        // add 40 to this entry
    swlw $7, 4($6)          // load the address there

    beq $6, $9, static_network_error
    // exact match, bail out
    bne $7, $2, static_network_sw_address_loop

    // note that the loop does not bail out unless it finds an
    // exact match, because the match must be in the table, given
    // the state.

    // at this point, $6 must contain a correct slot.

    swlw $4, 0($6)          // address match, load header

    andi $5, $4, 255        // load patch condition

    addiu $9, $6, 8         // beginning of patch data
    addiu $6, $6, 40        // point past end of patch data

    // at this point, we have found the patch and reader data in
    // the pending address table.
    // $2 contains the address of all this trouble.
    // $3 contains the cache line metadata's address
    // $4 contains the patch/reader bits
    // $5 contains only the patch bits
    // $6 contains the end of the section in the patch table
    // $9 contains the first patch line
    // $10 contains a MDN header corresponding to the writing tile
    // $7, $8, $11 swap space

static_network_sw_patchloop:
    ori $7, $0, 0x7         // load the current data word into $7
    or $30, $0, $31
    jal static_network_buffer_read
    or $31, $0, $30

    andi $8, $5, 1          // check current patch bit
    srl $5, $5, 1           // and shift down

    bne $8, $0, static_network_sw_patchloop_patched
    // this word is not patched, so the latest version is coming
    // off the static network.  Store it to memory.
    swsw $7, 0($9)

static_network_sw_patchloop_patched:

    addiu $9, $9, 4         // next patch point
    bne $6, $9, static_network_sw_patchloop

    addiu $6, $6, -40       // $6 has a little before the
                    // first point in cache line.

    // at this point, the correct cache line is stored in the
    // directory.  We must write it to its correct location in
    // RAM.

    // The actual address is in $2.

    andi $5, $4, 255        // load patch condition
    // we only monitor addresses that are patched, because there
    // is a finite number of readers, so if this line was free of
    // patches it was not accounted for, and thus does not need to
    // be freed explicitly.

    beq $5, $0, static_network_sw_notpatched

    // But if it was, free it.
//BUGFIX: previously (addiu $29, $29, 1)
    ilw $5, %lo(pending_addresses_remaining)($0)
    addiu $5, $5, 1
    isw $5, %lo(pending_addresses_remaining)($0)
    //addiu $29, $29, 1     // one more free address

.global static_network_sw_notpatched static_network_sw_notpatched:

    // send an MDN message to the FPGA with the new cache line.
    // we must be careful to drain the static network to prevent
    // deadlock!

    // send to ourselves, but route funny to the east
    li $11, CONSTRUCT_DYNAMIC_HEADER(4, 9, 4, (TILE_ROW >> 2), 3, 0, 3)
    rrmi $11, $2, 24, 0x60
    or $cmno, $11, $0
    or $cmno, $2, $0        // send the address

    // end loop at $6 + 28
    addiu $7, $6, 32

static_network_sw_notpatched_send_loop:
    mfsr $11, SW_BUF1
    rrm $11, $11, 16, 0x1
    beq $11, $0, static_network_sw_notpatched_send

    // buffer the static messages on the east port
    or $30, $0, $31         // store the old linkage pointer
    jal static_network_buffer_write
    or $31, $0, $30         // and restore LP

    j static_network_sw_notpatched_send_loop

static_network_sw_notpatched_send:

    swlw $cmno, 8($6)       // amd write the data
    addiu $6, $6, 4
    bne $6, $7, static_network_sw_notpatched_send_loop

    // reset $6 (designates beginning ptr)
    addiu $6, $6, -32

    // here, we have taken care of sending the patched data back
    // to main RAM.  Now, we must send it to any possible
    // recipients.
    // $2 contains the address of all this trouble
    // $3 contains the cache line metadata's address
    // $4 contains the patch/reader bits
    // $6 contains the beginning of the section in the address table
    // $10 contains a MDN header corresponding to the writing tile
    // $5, $7, $8, $9, $11 swap space

    srl $5, $4, 8           // waiting tiles

    // if no tiles, then exit
    beq $5, $0, static_network_sw_senddone

    sll $9, $20, 27         // size = 8

    addiu $7, $0, -29       // current tile.  -29 because
                    // we will add 1, and then 28
                    // more as the first part of
                    // the iterator.
static_network_sw_sendloop:

    addiu $7, $7, 1         // increment tile number
    andi $11, $7, 3         // tile multiple of 4?

    bne $11, $0, static_network_sw_noadjust

    addiu $7, $7, 28        // 0, 1, 2, 3, 32, 33, 34, ...

static_network_sw_noadjust:

    andi $8, $5, 1          // is this tile waiting?
    srl $5, $5, 1
    beq $8, $0, static_network_sw_sendloop

    // looks like this tile is waiting on data... send it along.
    // the header is the current tile number plus "size 8".
    addu $cmno, $7, $9

    swlw $cmno, 8($6)       // and write the data
    swlw $cmno, 12($6)
    swlw $cmno, 16($6)
    swlw $cmno, 20($6)
    swlw $cmno, 24($6)
    swlw $cmno, 28($6)
    swlw $cmno, 32($6)
    swlw $cmno, 36($6)

    // more tiles requesting?
    bne $5, $0, static_network_sw_sendloop

static_network_sw_senddone:

    // here, we are done with this address.  So what we must do
    // is clear it.  If it is the last address, then we simply
    // let it fall off the bottom.  If not, we take the last entry,
    // and move it into the entry being cleared.  In all cases,
    // we decrement the next-empty-spot pointer.

    swlw $8, 0($23)         // find the last address
    addiu $8, $8, -40       // subtract 40 because there
                    // is one fewer address.
    swsw $8, 0($23)          // store it back

    beq $8, $6, static_network_sw_last

    // must not have been the last entry.  So therefore, take what
    // is the last entry (*$8) and move it to here (*$6).  Now
    // the last entry is truly blank and the pointer $8 is correct.

    // yes, this code is a brute, but I invite you to come up with
    // a better memcpy call!

    swlw $5, 0($8)
    swsw $5, 0($6)
    swlw $5, 4($8)
    swsw $5, 4($6)
    swlw $5, 8($8)
    swsw $5, 8($6)
    swlw $5, 12($8)
    swsw $5, 12($6)
    swlw $5, 16($8)
    swsw $5, 16($6)
    swlw $5, 20($8)
    swsw $5, 20($6)
    swlw $5, 24($8)
    swsw $5, 24($6)
    swlw $5, 28($8)
    swsw $5, 28($6)
    swlw $5, 32($8)
    swsw $5, 32($6)
    swlw $5, 36($8)
    swsw $5, 36($6)

static_network_sw_last:

    rrm $8, $10, 3, 0xC      // $8 has yy00 of writer
    rrmi $8, $10, 0, 0x3         // $8 has yyxx (tile number)

    // now, write the new state, which is "read" on all the tiles
    // we just sent data to (list is in $4), and also the writing
    // tile.

    // BUGFIX: THE WRITING TILE MAY NOT HAVE FRESH DATA, SINCE THE
    // LINE COULD HAVE BEEN PATCHED.

    //sllv $6, $15, $8       // 1 << (writingtile + 8)
    //FIX:
    or $6, $0, $0

    andi $5, $4, 255        // patch bits
    subu $4, $4, $5         // all but patch bits

    or $6, $4, $6           // new state

    sw $6, 0($3)            // store new state in state space

    j static_network_done

.global static_network_sw_nopatch_noreq
static_network_sw_nopatch_noreq:

    // this is a bypass in which we simply write the line back
    // to main RAM, and set the reader as the old writer, and
    // go to the read state.

    // $2 contains the address of all this trouble.
    // $3 contains the cache line metadata's address
    // $4, $5, $6, $7, $8, $9, $10, $11 swap space

    // send to ourselves, but route funny to the east
    li $11, CONSTRUCT_DYNAMIC_HEADER(4, 9, 4, (TILE_ROW >> 2), 3, 0, 3)
    rrmi $11, $2, 24, 0x60
    or $cmno, $11, $0

    or $cmno, $2, $0        // send the address

    // must drain the static network to prevent deadlock!

    li $6, 8
static_network_sw_nopatch_noreq_loop:
    mfsr $11, SW_BUF1
    rrm $11, $11, 16, 0x1
    beq $11, $0, static_network_sw_nopatch_noreq_send

    // buffer the static messages on the east port
    or $30, $0, $31         // store the old linkage pointer
    jal static_network_buffer_write
    or $31, $0, $30         // and restore LP

    j static_network_sw_nopatch_noreq_loop
```

```
static_network_sw_nopatch_noreq_send:
    ori $7, $0, 0x1B        // foward 8 data words (reg27 = cmno)
    or $30, $0, $31
    jal static_network_buffer_read
    or $31, $0, $30

    addiu $6, $6, -1
    bne $6, $0, static_network_sw_nopatch_noreq_loop

    sw $0, 0($3)

    // now, store the new state (read on this tile)
    //rrm $8, $10, 3, 0xC       // $8 has yy00 of writer
    //rrmi $8, $10, 0, 0x3      // $8 has yyxx (tile number)

    //sllv $4, $15, $8      // 256 << this number
    //sw $4, 0($3)          // store this state, which is
                    // read on this one tile
                    // only.

static_network_done:
    jr $31

static_network_error:
    j .

// $10, $4, $3 must be preserved
// $5, $8, $9 are modified
.global static_network_buffer_write static_network_buffer_write:
    // set the STATIC BUFFER CONDITION
    or $19, $20, $0

static_network_buffer_write_loop:
    // read data off static network
    or $5, $csti, $0

    la $9, static_input_buffer
    swlw $8, 4($9)          // first free address in buffer
    swsw $5, 0($8)          // store input in first free location

    // calculate new end-of-buffer ptr (must bounds check)
    addiu $8, $8, 4
    addiu $5, $8, -4*STATIC_BUFFER_SIZE

    // the end-of-buffer wraps around to static_input_buffer+8
    bne $5, $9, static_network_buffer_write_update
    addiu $8, $9, 8

static_network_buffer_write_update:
    swlw $5, 0($9)
    beq $8, $5, static_network_buffer_write_overflow

    // otherwise, end-of-buffer = old-end + 4
    swsw $8, 4($9)

    // clear rest of the words on EAST port
    mfsr $8, SW_BUF1
    rrm $8, $8, 14, 0x1
    bne $8, $0, static_network_buffer_write_loop

    jr $31

static_network_buffer_write_overflow:
    j .


// $7 contains register to read into
// $7, $8, $11 are destroyed
// result is returned in register defined in $7

.global static_network_buffer_read static_network_buffer_read:
    sll $7, $7, 3           // multiply reg# by 8, to index into switch

    la $11, static_network_buffer_read_table
    addu $11, $7, $11       // $11 now contains index into table

    // if buffer is empty, then read directly off static network.
    // otherwise, read from the buffer.

    beq $19, $0, static_network_buffer_read_bypass
.global static_network_buffer_read_real
static_network_buffer_read_real:
    // load the beginning-ptr from memory into $8
    la $7, static_input_buffer
    swlw $8, 0($7)

    // load the word at the beginning of the buffer
    swlw $8, 0($8)

    // move the word into the proper register
    jr $11


static_network_buffer_read_bypass:
    or $8, $csti, $0        // read from static network into $8
    jr $11              // move word into proper register


static_network_buffer_read_inc:
    // must store these scratch registers, because they may contain
    // the return value initially defined in $7

    isw $7, %lo(reg_1)($0)
    isw $8, %lo(reg_2)($0)
    isw $11, %lo(reg_3)($0)

    // if this was a read off the static network, then we are done
    beq $19, $0, static_network_buffer_read_done

    // reload initial $7 value
    la $7, static_input_buffer

    // reload the beginning ptr
    swlw $8, 0($7)

    // increment beginning ptr, and bounds-check
    addiu $8, $8, 4
    addiu $11, $8, -4*STATIC_BUFFER_SIZE

    bne $11, $7, static_network_buffer_read_updateptr
    addiu $8, $7, 8

static_network_buffer_read_updateptr:
    // store new beginning ptr
    swsw $8, 0($7)

    // load end-of-buffer ptr
    swlw $7, 4($7)

    // if begin-ptr == end-ptr, then reset STATIC BUFFER CONDITION
    bne $7, $8, static_network_buffer_read_done
    or $19, $0, $0

static_network_buffer_read_done:
    // reload scratch registers
    ilw $7, %lo(reg_1)($0)
    ilw $8, %lo(reg_2)($0)
    ilw $11, %lo(reg_3)($0)

    jr $31


static_network_buffer_read_table:
    or $0, $0, $8
    j static_network_buffer_read_inc
.set noat
    or $1, $0, $8
.set at
    j static_network_buffer_read_inc
    or $2, $0, $8
    j static_network_buffer_read_inc
    or $3, $0, $8
    j static_network_buffer_read_inc
    or $4, $0, $8
    j static_network_buffer_read_inc
    or $5, $0, $8
    j static_network_buffer_read_inc
    or $6, $0, $8
    j static_network_buffer_read_inc
    or $7, $0, $8
    j static_network_buffer_read_inc
    or $8, $0, $8
    j static_network_buffer_read_inc
    or $9, $0, $8
    j static_network_buffer_read_inc
    or $10, $0, $8
    j static_network_buffer_read_inc
    or $11, $0, $8
    j static_network_buffer_read_inc
    or $12, $0, $8
    j static_network_buffer_read_inc
    or $13, $0, $8
    j static_network_buffer_read_inc
    or $14, $0, $8
    j static_network_buffer_read_inc
    or $15, $0, $8
    j static_network_buffer_read_inc
    or $16, $0, $8
    j static_network_buffer_read_inc
    or $17, $0, $8
    j static_network_buffer_read_inc
    or $18, $0, $8
    j static_network_buffer_read_inc
    or $19, $0, $8
    j static_network_buffer_read_inc
    or $20, $0, $8
    j static_network_buffer_read_inc
    or $21, $0, $8
    j static_network_buffer_read_inc
    or $22, $0, $8
    j static_network_buffer_read_inc
    or $23, $0, $8
    j static_network_buffer_read_inc
    or $24, $0, $8
    j static_network_buffer_read_inc
    or $25, $0, $8
    j static_network_buffer_read_inc
    or $26, $0, $8
    j static_network_buffer_read_inc
    or $27, $0, $8
    j static_network_buffer_read_inc
    or $28, $0, $8
    j static_network_buffer_read_inc
    or $29, $0, $8
    j static_network_buffer_read_inc
    or $30, $0, $8
    j static_network_buffer_read_inc
    or $31, $0, $8
    j static_network_buffer_read_inc


// pending IRQ handler.  Destroys all swap space.
.global pending_irq_begin pending_irq_begin:

    sll $3, $16, 8          // clear upper 8-bits of UD map
    srl $3, $3, 8

    or $7, $28, $0          // load hot-map
    or $8, $0, $0           // current U-tile number (times
                    // four)
pending_irq_u_loop:
    andi $5, $7, 0x1        // current hot U-tile request
    andi $4, $3, 0xFF       // current U-tile requests

    or $4, $4, $5
    // if there is nothing for this U-tile, then skip attempting to
    // process it, and just go to the next one.
    beq $4, $0, pending_irq_next_u

    ilw $6, %lo(int_status_0)($8)   // load the tile state from
                    // instruction memory.

    andi $6, $6, 3          // get only the state bits

    bne $6, $0, pending_irq_next_u  // tile busy.  Go away.  Either
                    // we just interrupted it, or we
                    // are communicating with it via
                    // MDN, so things are happening.
.global pending_irq_do_interrupt pending_irq_do_interrupt:

    // tile may be interrupted, so lets interrupt it.
    mfsr $6, CYCLE_LO       // grab cycle count
    sll $6, $6, 2           // shift over
    addiu $6, $6, 2         // and add state bits (state=2 is
                    // "interrupt pending")

    isw $6, %lo(int_status_0)($8)   // store new state

    srl $4, $8, 2           // shift tile-number into place

    // $17 is a pre-made interrupt header, waiting for a tile number.
    or $cmno, $17, $4       // instant interrupt, booya

    beq $5, $0, pending_irq_standard_int // if not hot-request, must
                                         // be normal request

    ////////////////////////////////////////////////////////
    // THIS U-TILE HAS SOME PENDING HOT-REQUESTS
    ////////////////////////////////////////////////////////
.global pending_irq_do_hot pending_irq_do_hot:
    // clear entry in hot-map
    srl $6, $8, 2           // tilenumber x 1
    sllv $6, $20, $6        // 1 << tile_number
    xor $6, $6, $21         // everything except this

    and $28, $6, $28        // clear bit in hot-map

    // load hot requests
    la $5, hot_request_table
    sll $6, $8, 2           // current U-tile number x 16
    addu $5, $6, $5         // entry in hot_request_table
    addiu $9, $5, 0x10      // $9 iterator over hot_requests

    li $4, ((4 << 24) + TILE_ROW << 3)
    srl $6, $8, 2           // U-tile number
    or $cgno, $4, $6        // send GDN header

pending_irq_hot_loop:
    addiu $9, $9, -4

    swlw $6, 0($9)          // load request from particular dir-tile
    or $cgno, $6, $0        // send actual request
    swsw $0, 0($9)          // clear request from table

pending_irq_hot_loop_end:
    bne $9, $5, pending_irq_hot_loop

    j pending_irq_next_u

.global pending_irq_standard_int pending_irq_standard_int:
    // now send list of requests
    sll $5, $20, 24         // $5 = 1 << 24
    ori $5, $5, (TILE_ROW << 3)
    addu $cgno, $5, $4      // header... message is of
                    // length 2


    or $cgno, $16, $0       // send along the big list of
                    // requests (let user process
                    // it!)
                    // this includes the possible
                    // "you've been spoofed!" bit.

    sll $4, $8, 1           // tile number times 8 (needed
                    // later).

    sllv $5, $12, $4        // shift over constant 0xFF
                    // ($12) appropriately

    xor $2, $5, $21         // generate mask
                    // ($21 = 0xFFFFFFFF save 2
                    // instrs)

    and $16, $2, $16        // and zero out just-sent
```

```
                    // requests

pending_irq_next_u:
    addiu $8, $8, 4         // increment tile number being
                    // handled

    srl $3, $3, 8           // next U-tile is pending
    srl $7, $7, 1           // next hot U-tile pending
    or $4, $3, $7

    bne $4, $0, pending_irq_u_loop  // loop if there are more
                    // requests

pending_irq_done:
    jr $31


// gdn message handler.  This takes requests from user tiles to either
// D-tile or I-tile functionality of the directory tile.
// Destroys all swap space
gdn_handler_begin:

    // check to see if there is anything on the GDN
        mfsr $4, GDN_BUF        // pending incoming data
    rrm $4, $4, 5, 0x7      // knock off "out" and other crap

        beq $4, $0, gdn_handler_done    // no data?  oh well, goodbye

    or $4, $cgni, $0        // read in a memory word

    andi $6, $4, 0xF00      // get the request type
    andi $7, $4, 0x7F       // get the tile number (in
                    // somewhat wrong GDN form)
                    // into $7 (this is kept a while)

    // $$$ make request number equal jump length and do a jr.

    // request types are:
    // 0x000 - "okay, done" from U-tile to I-tile
    // 0x100 - "yes?" from U-tile to I-tile
    // 0x200 - "yes?" from U-tile to D-tile ("hello")
    // 0x300 - list from D-tile to I-tile ("call this tile")
    // 0x400 - user to directory acknowledgement
    // 0x500 - I-tile to D-tile deadlock invalidate
    // 0x700 - D-tile to I-tile deadlock invalidate done
// LOCK requests
// 0x900 - test lock
// 0xA00 - set lock
// 0xB00 - release lock

// SHM requests
// 0xC00 - malloc
// 0xD00 - free

beq $6, $0, gdn_handler_ui_done

addiu $6, $6, -0x100        // 0x100 equals zero now
beq $6, $0, gdn_handler_ui_call

addiu $6, $6, -0x100        // 0x200 equals zero now
beq $6, $0, gdn_handler_ud_call

addiu $6, $6, -0x100        // 0x300 equals zero now
beq $6, $0, gdn_handler_di_list

addiu $6, $6, -0x200        // 0x500 equals zero now
beq $6, $0, gdn_handler_id_dead

// SCAFFOLD branch to "MDN" land
addiu $6, $6, -0x100        // 0x600 equals zero now
beq $6, $0, mdn_gdn_bypass

addiu $6, $6, -0x100        // 0x700 equals zero now
beq $6, $0, gdn_handler_di_dead_reply

addiu $6, $6, -0x100        // 0x800 equals zero now
beq $6, $0, gdn_handler_di_hot

addiu $6, $6, -0x600        // 0xE00 equals zero now
beq $6, $0, gdn_handler_id_hot_done

// default, must be a user-to-directory acknowledgement.  In
// this case, all of the addresses that needed to be invalidated
// were invalidated, and we can clear some bits from their
// state.

// from the tile number in YY000XX form, get a pending request
// begin address, which is YYXX0000000.
rlm $6, $7, 4, 0x600        // YY000000000
```

```
    rlmi $6, $7, 7, 0x1FF       // YYXX0000000

    // generate a mask that has all bits set, except the bit
    // corresponding to our tile number, so that we can OR it with
    // a state and mask out acked readers.
    srl $7, $6, 7           // this is the requesting
                    // tile number

    sllv $2, $15, $7        // $2 = 1 << (this number + 8)
    xor $7, $21, $2         // $7 = all bits but this one

// BUGFIX: previously addu $5, $6, $28 (no longer uses $28)
    la $5, pending_request_table
    addu $5, $6, $5         // load beginning of pending
                    // requests

gdn_handler_ud_nextaddress:
    swlw $3, 0($5)          // load address
    addiu $5, $5, 4         // and increment pointer

    // if this is the null-terminator then we have no more actual
    // addresses.
    beq $3, $0, gdn_handler_ud_done

    // otherwise, $3 contains an address.

    srl $3, $3, 7
    sll $3, $3, 2           // address shifted right
    addu $3, $22, $3        // cache line metadata address

    lw $4, 0($3)            // load the status of the line
    // PERF-FIX(1):  if this bit is already zero, goto next address.
    // We do not want to ack the writing twile more than once.

    and $8, $4, $2
    beq $8, $0, gdn_handler_ud_nextaddress

    and $4, $4, $7          // mask out the acknowledging
                    // tile
    sw $4, 0($3)            // store it back

    // PERF-FIX(1): if we are in READ state, then dont act upon this
    andi $8, $4, 3
    beq $8, $0, gdn_handler_ud_nextaddress
    srl $8, $4, 8           // get only a list of readers
    bne $8, $0, gdn_handler_ud_nextaddress

    // looks like that was the last reader, and we are NOT read.
    // Meaning, we are in a new state (exclusive or excl-pending)
    // and thus must send an ack to the writing tile via the GDN.
    // First, generate an MDN header to send back to the writing
    // tile.  Its address is in the form YYXX0000 in the state
    // and we must convert it to 0YY000XX.

    rrm $9, $4, 1, 0x60     // 0YY00000
    rrmi $9, $4, 4, 0x3     // 0YY000XX

    sll $11, $20, 24        // $11 = 1 << 24
    addu $cmno, $11, $9     // header... message is of
                  // length 1

    // now, send the ack - which is either a plain ack (0) or an
    // ack-and-flush (1).  This corresponds precisely (I am so
    // clever) with the low bit of the state.

    andi $cmno, $4, 1

    j gdn_handler_ud_nextaddress

gdn_handler_ud_done:
    // now that the U-tile has ack'd the directory, we clear its
    // pending requests.

    la $5, pending_request_table
    addu $5, $6, $5
    swsw $0, 0($5)

    j gdn_handler_done

.global gdn_handler_id_hot_done gdn_handler_id_hot_done:
    // get hot_pending_request address by converting to YYXX00
    rrm $6, $7, 1, 0x30
    rlmi $6, $7, 2, 0xC

    // generate a mask that has all bits set, except the bit
    // corresponding to our tile number, so that we can OR it with
    // a state and mask out acked readers.
    srl $7, $6, 2           // this is the requesting
                  // tile number
```

```
sllv $2, $15, $7        // 1 << (this number + 8)
xor $7, $21, $2         // all bits but this one

la $5, hot_pending_request_table
addu $5, $6, $5

swlw $3, 0($5)          // load entry
swsw $0, 0($5)          // clear entry in table

// if entry == 0x10000000, then request was a flush, and we don't
// need to clear any bits in the state.

li $6, 0x10000000
and $6, $3, $6
bne $6, $0, gdn_handler_done

srl $3, $3, 7
sll $3, $3, 2           // address shifted right
addu $3, $22, $3        // cache line metadata address

lw $4, 0($3)            // load the status of the line
// PERF-FIX(1):  if this bit is already zero, goto next address.
// We do not want to ack the writing tile more than once.

and $8, $4, $2
beq $8, $0, gdn_handler_done

and $4, $4, $7          // mask out the acknowledging
                // tile
sw $4, 0($3)            // store it back

// PERF-FIX(1): if we are in READ state, then dont act upon this
andi $8, $4, 3
beq $8, $0, gdn_handler_ud_nextaddress

srl $8, $4, 8           // get only a list of readers
bne $8, $0, gdn_handler_done

// looks like that was the last reader.  Meaning, we are in
// a new state (exclusive or excl-pending) and thus must
// send an ack to the writing tile via the GDN.  First, generate
// an MDN header to send back to the writing tile.  Its address
// is in the form YYXX0000 in the state and we must convert it
// to 0YY000XX.
```

```
    rrm $9, $4, 1, 0x60      // 0YY00000
    rrmi $9, $4, 4, 0x3      // 0YY000XX

    sll $11, $20, 24         // $11 = 1 << 24
    addu $cmno, $11, $9      // header... message is of
                    // length 1

    // now, send the ack - which is either a plain ack (0) or an
    // ack-and-flush (1).  This corresponds precisely (I am so
    // clever) with the low bit of the state.

    andi $cmno, $4, 1

    j gdn_handler_done

.global gdn_handler_ui_done gdn_handler_ui_done:

    rlm $4, $7, 2, 0xC       // tile number's horizontal
                    // offset, times four

    // this is the correct offset in the interrupt status table.

    isw $0, %lo(int_status_0)($4)   // store a zero, state of "not
                    // in interrupt" because it just
                    // finished talking.
    j gdn_handler_done

gdn_handler_ui_call:

    // U-I communication ("who was that???")

    rlm $4, $7, 2, 0xC       // find tile numbers horiz offset
                    // times four.  We now have the
                    // correct address in interrupt
                    // status table land.

    isw $20, %lo(int_status_0)($4)  // store a "1" to the location,
                    // meaning that a phone-call was
                    // successful so the tile is now
                    // in interrupt.

    sll $4, $4, 1            // tile number horiz offset x 8

    li $5, 0x10          // $5 corresponds to forcer bit
    sllv $5, $5, $4          // forcer bit shifted over
    xor $5, $5, $21         // everything but this forcer bit
    and $16, $5, $16

    j gdn_handler_done

.global gdn_handler_ud_call gdn_handler_ud_call:

    // U just pinged D (the important phone call), so send along a
    // list of requests.  For each of these requests, if it is an
    // invalidate, write it to the pending (already sent) request
    // table.

    // from the tile number in YY000XX form, get an OCREQ begin
    // address, which is YYXX0000000.
    rlm $6, $7, 4, 0x600        // YY000000000
    rlmi $6, $7, 7, 0x1FF       // YYXX0000000

    srl $2, $6, 7           // tile number we are dealing
                    // with (needed later)

    addu $4, $6, $18        // load beginning of OCREQs for
                    // this particular tile

// BUGFIX: previously addu $5, $6, $28 (no longer uses $28)
    la $5, pending_request_table
    addu $5, $5, $6         // and list of pending requests
                    // which we will be storing
                    // into.

    swlw $3, 0($4)          // number of requests for
                    // this tile.

    srl $3, $3, 2           // shift right to get rid of
                    // off-by-four
    swsw $0, 0($4)          // and reset that counter

    addiu $9, $3, 1         // message length is the number
                    // of requests plus one, so
                    // shift that
    sll $9, $9, 24          // over 24 slots so it is in the
                    // right place in the header.

    or $cgno, $9, $7        // add tile number to message
                    // length, voila header
    or $cgno, $3, 0         // send along number of requests
                    // coming

gdn_handler_ud_loop:

    addiu $4, $4, 4         // next slot

    addiu $3, $3, -1        // decrement number of requests
                    // remaining
    swlw $9, 0($4)          // load the actual request

    or $cgno, $9, $0        // and send it along

    // here, check to see if the request is an invalidate or a
    // flush.  If it's an invalidate (ends in 2, as opposed to in 1)
    // then we must store it in the outstanding buffer space.

    andi $10, $9, 1         // low bit
    bne $10, $0, gdn_handler_ud_flush

    // must be an invalidate.
    swsw $9, 0($5)          // store the address

    addiu $5, $5, 4         // next slot in pending
                    // request table

gdn_handler_ud_flush:

    bne $3, $0, gdn_handler_ud_loop // more UD OCREQs to deal with?

    // null-terminate the list of pending addresses.  Since 0 is
    // not a possible shared address, it will never exist
    // legitimately in this table, and thus works as a terminator.
    swsw $0, 0($5)

    // is there a static-network stall due to OCREQ backlog?  Well,
    // we just sent some OCREQs so maybe we can clear it.
    beq $13, $0, gdn_handler_done

    // $2 is the tile we just processed
    // $9 is a blank slot

    sllv $9, $20, $2        // 1 << tile number
    xor $9, $21, $9         // invert (all "1"s except bit
                    // at tile# is a zero)
    and $13, $13, $9        // knock off a bit

    j gdn_handler_done

.global gdn_handler_di_list gdn_handler_di_list:

    // a list from a D-tile to an I-tile that the D-tile expects can
    // help it.  The message contains the D-tile's one-hot index and
    // the U-tile it wishes to communicate with.

    or $4, $cgni, $0        // read in a GDN word (UUDDDD)

    rrm $2, $4, 1, 0x18     // shift by 1 (UUDDD) and get the
                    // column (UU000).
    andi $3, $4, 0xF        // get request numbers of dirtile
                    // (OODDDD)
    sllv $2, $3, $2         // shift dirtile and header
                    // correctly.
                    // (OODDDD << UU000)
    or $16, $2, $16         // mask in the new request

    j gdn_handler_done


// pending_hot:

// 0x000: # of requests for User0
// 0x004: User0 request from dir0
// 0x008: User0 request from dir1
// 0x00c: User0 request from dir2
// 0x010: User0 request from dir3

// 0x020: # of requests for User1
// 0x024: User1 request from dir0
// etc.

.global gdn_handler_di_hot gdn_handler_di_hot:
    or $2, $cgni, $0        // $2 = uCOL.uCOL.dROW.dROW
    or $4, $cgni, $0        // $4 = dir-tile request
    sll $8, $2, 2           // shift left 2, to index into hot_table

    la $3, hot_request_table    // store request in hot_table
    addu $3, $3, $8

    swsw $4, 0($3)
    srl $8, $2, 2           // $8 = uCOL.uCOL
    sll $8, $20, $8         // $8 = 1 << uCOL.uCOL

    or $28, $28, $8         // update hot-map

    j gdn_handler_done

.global gdn_handler_id_dead gdn_handler_id_dead:
    or $2, $cgni, $0        // read in user-tile in YY000XX format
    or $8, $cgni, $0        // read in requesting I-tile YY000XX

    or $30, $0, $31
    jal deadlock_bypass
    or $31, $0, $30

    beq $3, $0, gdn_handler_done

    // Address found in local dir-tile. In this case, we must
    // send back an acknowledgement, along with the tile number
    // we just addressed.

    sll $11, $20, 25        // message length 2

    or $cgno, $11, $8       // send header
    addiu $cgno, $0, DI_DEAD    // send ack opcode
    addu $cgno, $2, $0      // send user-tile

    j gdn_handler_done

.global gdn_handler_di_dead_reply gdn_handler_di_dead_reply:
    // read in tile (YY000XX)
    or $4, $cgni, $0

    sll $10, $20, 24

    //wait for GDN output to be cleared, before sending spoof
gdn_handler_di_dead_reply_loop:
    mfsr $3, GDN_BUF
    andi $3, $3, 0x1F
    bne $3, $0, gdn_handler_di_dead_reply_loop

    // send spoof message
    or $cgno, $4, $10
    or $cgno, $0, $0
    // message length of 8
    sll $10, $20, 27

    or $cmno, $4, $10
.rept 8
    ori $cmno, $0, 666      // and 8 bogus words
.endr
    j gdn_handler_done



gdn_handler_done:

    jr $31


// mdn message handler.  This takes unexpected event counter events from
// U-tile to D-tile.
// Destroys all swap space
mdn_handler_begin:

    // if we do not have any outstanding OCREQ space, then we must
    // not attempt an MDN read
    bne $13, $0, mdn_handler_done

//BUGFIX: previously was (bne $29, $0, mdn_handler_done)
    ilw $2, %lo(pending_addresses_remaining)($0)
    // same thing with no outstanding address space
    bne $2, $0, mdn_handler_done

    // check to see if there is anything on the MDN
        mfsr $4, MDN_BUF        // pending incoming data
    rrm $4, $4, 5, 0x7      // knock off "out" and other crap

        beq $4, $0, mdn_handler_done    // no data?  oh well, goodbye

// SCAFFOLD mdn-gdn bypass...

.global mdn_gdn_bypass mdn_gdn_bypass:

    // SCAFFOLD should be cmni
    or $7, $cgni, $0        // read in a memory word
                    // this is the tile number in
                    // YY000XX form (and it also
                // has a MDN return handler -
                // self addressed stamped
                // envelope!)

// an event counter event.  This implies that a cache line went
// exclusive.

// SCAFFOLD mdn
or $2, $cgni, $0        // read in the address

// SCAFFOLD mdn
or $10, $cgni, $0       // and the data

rrm $8, $7, 3, 0xC      // $8 has yy00 of writer
rrmi $8, $7, 0, 0x3     // $8 has yyxx (tile number)

// $2 is an address.
// $7 has an MDN header
// $8 is an offending tile (YYXX)
// $10 is a data word
// $3, $4, $5, $6, $9, $11 free

srl $3, $2, 7
sll $3, $3, 2           // address shifted right
addu $3, $22, $3        // cache line metadata address

andi $5, $2, 0x1C       // low bits of address (which
                // part of the cache line is
                // getting modified)

srl $4, $2, 5           // address, high bits only
sll $2, $4, 5

lw $4, 0($3)            // load the status of the line

andi $11, $4, 7         // grab state number

// here, if the state is "read" then we do one thing, and if
// it isn't, another.

beq $11, $0, mdn_handler_read

// must not be read.  Check to see if the writer is the owner.

rrm $11, $4, 4, 0xF     // tile number only (YYXX)
```

```
    // if the writer is the owner, then we cannot sen4 d it an
    // invalidate because then it would lose precious data!
    bne $11, $8, mdn_handler_noclash

    or $cmno, $0, $7        // send the ack.
    or $cmno, $0, $0

    j mdn_handler_done

mdn_handler_noclash:

    // In this case, the state is not read, and the writer is not
    // the owner, so we must record its data in the patch table,
    // and send it an invalidate request.

    or $cmno, $0, $7        // send the ack.
    ori $cmno, $0, 2        // message is "invalidate"

    // $2 contains the address
    // $3 contains its state address
    // $4 the actual state
    // $5 the word of the cache line
    // $10 is a data word
    // $6, $7, $9, $11 free
    // PERF-FIX: the writing tile must be cleared as a reader, so as to
    // prevent deadlocking with itself (we already send an invalidate,
    // so we dont need to wait for the tile to ACK an MDN invalidate)

    //sllv $6, $15, $8      // $6 = ( 1 << tile_number + 8)
    //xor $7, $21, $6           // $7 = everything but this bit
    //and $9, $6, $4
    //beq $9, $0, mdn_handler_noclash_skip

    // we are in RL or RLB, and writing tile is a current reader
    //and $4, $7, $4            // clear the writer as a reader
    //sw $4, 0($3)
    //srl $9, $4, 8
    //bne $9, $0, mdn_handler_noclash_skip

//.global mdn_hit
//mdn_hit:
    // in this case, we have just moved from RL -> EXP or
    // from RLB -> EXP so send an ack to the writing tile.
    //ori $4, $4, 1          // set state to EXP
    //sw $4, 0($3)

    //rrm $9, $4, 1, 0x60        // 0YY00000
    //rrmi $9, $4, 4, 0x3        // 0YY000XX

    //sll $11, $20, 24       // $11 = 1 << 24
    //addu $cmno, $11, $9        // header... message is of
                    // length 1
    // now, send the ack-and-flush
    //ori $cmno, $0, 1
    //j mdn_handler_writeback_set

//mdn_handler_noclash_skip:
    // First, update the state and get a writeback if needed.

    andi $6, $4, 1           // low bit is on?
    bne $6, $0, mdn_handler_writeback_set

    // we must request a writeback and also update the state as
    // either going from read-lock to read-lock-bypass, or from
    // exclusive to exclusive-pending.  This is done by setting the
    // low bit.

    ori $4, $4, 1            // set that bit
    sw $4, 0($3)             // and store back.

    // if we are in the read-lock state, then the writeback will
    // occur later, and we do not need to force it.

    // BUGFIX (should be SRL)
    srl $6, $4, 8            // any readers implies read-lock
    bne $6, $0, mdn_handler_writeback_set

    // Set up a writeback request.  $2 contains the address.  $9
    // must contain the tile number, which (given that we are not
    // in "read") is in bits 4-7 of the state vector.

    rrm $9, $4, 4, 0xF       // tile number
    li $11, 1            // request number 1 - coughup

    or $30, $0, $31          // store the old linkage pointer
    jal ocreq_begin          // make function call
    or $31, $0, $30          // and restore LP
```

```
mdn_handler_writeback_set:

    // Now that that is out of the way, we record the patch.
    // $2 and $5 have the full address, $10 the data.

    or $6, $23, $0          // load the beginning of
                    // the pending-address table

    // here, what we do is check all the addresses in the pending
    // address table.  If we have a match, then we go to that one.
    // Otherwise, we dribble off the bottom and go to the next one.

    swlw $9, 0($6)          // this is the address of the
                    // first free entry in the table.

mdn_handler_address_loop:
    addiu $6, $6, 40        // add 40 to this entry
    swlw $7, 4($6)          // load the address there

    //BUGFIX: previous impl. failed when $7=$2 and $6=$9
    // new entry needs to be allocated
    beq $6, $9, mdn_handler_address_new

    // exact match, bail out
    beq $7, $2, mdn_handler_address_exact

    j mdn_handler_address_loop

mdn_handler_address_new:

    // must be a new entry.  Where we are now, plus 40, is the next
    // free entry.
    addiu $7, $6, 40
    swsw $7, 0($23)         // store this new past-end
                    // pointer

    swsw $2, 4($6)          // and our current block must
                    // have the new address

    swsw $0, 0($6)          // clear out any possible
                    // detritus control bits.
BUGFIX:
    ilw $3, %lo(pending_addresses_remaining)($0)
    addiu $3, $3, -1
    isw $3, %lo(pending_addresses_remaining)($0)
    //addiu $29, $29, -1        // one more patch slot used

mdn_handler_address_exact:

    // at this point...
    // $2 contains the transactions address
    // $6 an offset in the pending address table
    // $5 an offset to a word in the cache line
    // $10 the data word
    // $3, $4, $7, $8, $9 free

    addu $9, $6, $5         // add the cache subpart to
                    // get the correct offset

    swsw $10, 8($9)         // and store the data word
                    // so kindly provided by the
                    // writing tile.  "8" is because
                    // the first two words in the
                    // block are header.

// BUGFIX - subpart one hot-encoding must use the offset into the
// cacheline (previously used $8, the tile number)

    srl $5, $5, 2           // number of subpart
    sllv $5, $20, $5        // one-hot encoding

    swlw $9, 0($6)          // grab header
    or $9, $5, $9           // mask in bit

    swsw $9, 0($6)          // and store new header

    j mdn_handler_done

mdn_handler_read:

    // $4 contains the state.  In order to check for extra readers
    // (defined as those that are not requesting exclusive access),
    // the state word is manipulated to pull out the requesting
    // tile, and this word is compared to zero.

    // If there are no other readers, then our life is simple, and
    // we set the tile to be exclusive ($9 has a tile number) and
    // send the ack.  Otherwise, we go into read lock.
    // $3 has the address of the state
    // $4 has the state
    // $7 has a return envelope
    // $8 has the offending tile

    sllv $6, $15, $8        // 1 << offender
    xor $6, $21, $6         // zero in that position, ones
                    // everywhere else
    and $4, $4, $6          // modified state line

    bne $4, $0, mdn_handler_read_lock

    // looks like the requestor was the only reader.

    // set state to exclusive.
    sll $9, $8, 4           // move to proper position
    ori $6, $9, 2           // state is exclusive on the tile

    sw $6, 0($3)

    // $7 has the first word sent by the tile to the directory, and
    // it is a return-addressed MDN header with its number as the
    // recipient, and the length 1.

    or $cmno, $7, $0        // send back the ack
    or $cmno, $0, $0        // which is zero for "okay"

    // that's all, folks.
    j mdn_handler_done

mdn_handler_read_lock:

    // this is the more complicated case.  We have readers.
    // Therefore, we have to set the state to read lock, on the
    // extant readers.  We do not set it on the new writer, because
    // we cannot send the new writer an invalidate (it's got the
    // fresh data!).

    sll $9, $8, 4           // put in the correct position
    ori $9, $9, 2           // set the read-lock state
                    // bit.

    or $4, $9, $4           // and OR in the writer and
                    // lock bit
    sw $4, 0($3)            // store new state

    // We must send a bunch of tiles an invalidate message, as a
    // newer version of the cache line was just found.

    srl $3, $4, 8           // list of readers

    or $9, $0, $0           // current potential tile number
                    // (start at zero)
    li $11, 2           // request 2 - invalidate

    // BUGFIX ($8 modified by ocreq)
    or $5, $8, $0
mdn_handler_read_loop:

    // this is the writing tile.  It has the latest version of
    // the cache line.  Can't send it an invalidate or we lose
    // data!

    beq $9, $5, mdn_handler_read_done

    andi $6, $3, 1          // low-bit mask

    // this tile is not on the list... bypass the call
    beq $6, $0, mdn_handler_read_done

    // make up an invalidate request for this tile, since it is on
    // the list.  $2 contains the address to be invalidated, $11 has
    // constant "2" because we are asking for an invalidate, and $9
    // is the tile number.  (three params to OCREQ call)

    or $30, $0, $31         // store the old linkage pointer
    jal ocreq_begin         // make function call
    or $31, $0, $30         // and restore LP

mdn_handler_read_done:
    srl $3, $3, 1           // knock off a tile
    addiu $9, $9, 1         // keep two counters synced

    // once the list is empty, it will be all zeroes so we must
    // continue if $3 is not zero, meaning more tiles need an
    // invalidate
    bne $3, $0, mdn_handler_read_loop

mdn_handler_done:
    jr $31


// deadlock detector routine

.global deadlock_detector_begin deadlock_detector_begin:

    // here, we detect possible deadlock by seeing if a tile that
    // got pinged N cycles ago still has not responded.  If this is
    // the case, we set up a special force condition, and also send
    // along a fake cache line.

    li $8, 8            // load this constant; it is
                    // the tile number we are
                    // servicing (times four).  An
                    // iterator.

deadlock_detector_loop:

    ilw $6, %lo(int_status_0)($8)   // load the tile state

    andi $4, $6, 2          // get only the high state bit,
                    // which is set only when state
                    // is 2 (there is no state 3)

    // if not in state 2 (pinged but no response yet), we are not
    // interested.
    beq $4, $0, deadlock_detector_next_u

    // check to make sure we haven't already sent the forcer, as
    // we want to do this precisely once!

    sll $4, $8, 1           // tile number (times 8)

    li $2, 0x10         // bit 4

    // BUGFIX: previously was sllv $2, $4, $2
    sllv $2, $2, $4         // shift over by 8 times the
                    // tile number - this is a
                    // forcer bit.

    and $11, $2, $16        // check this forcer bit.  If
                    // $5 is one, then we've already
                    // sent a forcer.
    bne $11, $0, deadlock_detector_next_u

    // check to see how long ago we've pinged.  This is in $6 (with a
    // footer of "10" but oh well)

    mfsr $7, CYCLE_LO         // grab cycle count
    sll $7, $7, 2             // shift over

    subu $7, $6          // subtract new time minus old
                    // time, giving the number of
                    // elapsed cycles since the last
                    // ping

    // subtract off the threshold
    li $11, DEADLOCK_TRIGGER_CYCLES
    subu $7, $7, $11

    // if this number is less than zero, then we have not exceeded
    // the number of cycles and all is well.
    bltz $7, deadlock_detector_next_u

.global deadlock_detector_force deadlock_detector_force:
    // oops, looks like we've exceeded our timing.
    sll $4, $8, 1             // tile number (times 8)

    li $2, 0x10          // bit 4

    // BUGFIX: previously was sllv $2, $4, $2
    sllv $2, $2, $4           // shift over by 8 times the
                    // tile number - this is a
                    // forcer bit.

    or $16, $2, $16           // set this bit.  The interrupt
                    // forcer is set.  This bit will
                    // be received by the tile and
                    // it knows to deal with the
                    // force.

    srl $11, $8, 2            // number of tile (times 1)
    ori $2, $11, (TILE_ROW<<3)  // correct tile in YY000XX

// BUGFIX: MUST ERASE THIS TILE'S PENDING REQUEST IN THE APPROPRIATE
// DIRECTORY TILE.

    or $30, $0, $31
    jal deadlock_bypass      // check local directory first
    or $31, $0, $30

    bne $3, $0, deadlock_detector_force_found

    // the local directory does not have a pending address.
    // therefore, we must message the other directory tiles.
    // Only after we receive a successful ack, can we proceed
    // to send the fake cache-line.

    li $11, 3            // length of message for MDN hdr.
    sll $11, $11, 24

    ori $10, $0, 3           // tile number of this tile
    ori $10, $10, (TILE_ROW<<3)

    ori $4, $0, 3            // $4 = row of current dir-tile
deadlock_detector_force_sendloop:
    sll $9, $4, 5
    ori $9, $9, 3            // tile in $9 = YY00011 format
    // don't send to ourselves!
    beq $9, $10, deadlock_detector_force_sendloop_next

    or $cgno, $9, $11        // send header to dir tile
    ori $cgno, $0, ID_DEAD       // send ID_DEAD opcode
    or $cgno, $2, $0         // send tile number in YY000XX
    or $cgno, $10, $0        // send self-addressed envelope
deadlock_detector_force_sendloop_next:
    addiu $4, $4, -1
    bgez $4, deadlock_detector_force_sendloop
    j deadlock_detector_next_u

deadlock_detector_force_found:

    sll $10, $20, 24
deadlock_detector_force_found_loop:
    mfsr $3, GDN_BUF
    andi $3, $3, 0x1F
    bne $3, $0, deadlock_detector_force_found_loop

    //////////////////
    or $cgno, $2, $10
    or $cgno, $0, $0
    ///////////////////
    sll $10, $20, 27

    or $cmno, $2, $10       // send header
.rept 8
    ori $cmno, $0, 666      // and 8 bogus words
.endr


deadlock_detector_next_u:
    addiu $8, $8, -4        // iterate

// BUGFIX: used to be bne $8, $0, deadlock_detector_loop
    bgez $8, deadlock_detector_loop

deadlock_detector_done:
    jr $31


// DEADLOCK bypass subroutine

// one arg is passed in

// $2 has the tile we are requesting for (YYXX)
// $3, $4, $5, $6, $7, $9 are destroyed
// return value in $3 (1 found, 0 not found)

.global deadlock_bypass deadlock_bypass:
    // convert YY000XX to YYXX format
    rrm $4, $2, 3, 0xC
    rrmi $4, $2, 0, 0x3

    // convert to one-hot tile number
    sllv $4, $20, $4

    sll $4, $4, 8           // shift left 8 to match reader-bits
    xor $3, $4, $21         // everything but this bit

    or $6, $23, $0
    swlw $9, 0($6)          // end ptr
    addiu $6, $6, 40

deadlock_bypass_loop:
    beq $6, $9, deadlock_bypass_notfound

    swlw $7, 0($6)
    and $5, $7, $4          // check if current entry matches

    bne $5, $0, deadlock_bypass_found
    addiu $6, $6, 40
    j deadlock_bypass_loop

deadlock_bypass_found:
    and $7, $7, $3          // apply mask
    swsw $7, 0($6)
    li $3, 1
    jr $31

deadlock_bypass_notfound:
    li $3, 0
    jr $31


// OCREQ handler subroutine.

// three args are passed in

// $2 has the address we are requesting
// $9 has the owner tile that we are doing the requesting for
// $11 has the request number (1 for cough-up, 0 for invalidate)

// $13 is the OCREQ overflow vector (global variable)
// $14 is the d-tile one-hot number (global constant)
// $18 is the OCREQ table begin address (global constant)
// $20 is "1" (global constant)

// $3, $5, $9 and $10 must not be written, as a caller needs those.
// $30 is a linkage pointer one stack-frame away, so must not be killed

// $4, $6, $7, $8 are destroyed

ocreq_begin:
    // the OCREQ consists of a message to an I-tile, asking the
    // I-tile to interrupt a U-tile.  Locally, we store what we want
    // to tell the owner when the owner calls back.  The OCREQ table
    // is first collated by tile #, and each tile gets 2^7
    // consecutive bytes (32 words).  Each of these words is a
    // particular OCREQ.

    // since tiles 3, 7, 11, 15 are actually D-tiles, the OCREQ tile
    // space corresponding to them does not contain actual OCREQs.
    // instead, they contain the number of outstanding requests
    // contained in the table for the preceding 3 tiles.  Thus, the
    // table is set up like so:

    // 0x000 [OCREQs to tile 0] - 32 words
    // 0x080 [OCREQs to tile 1] - 32 words
    // 0x100 [OCREQs to tile 2] - 32 words
    // 0x180 - number of requests to tile 0
    // 0x184 - number of requests to tile 1
    // 0x188 - number of requests to tile 2
    // 0x18C - 18C through 1FC are blank (29 words)

    // 0x200.... tile 4, 5, 6... etc

    sll $6, $9, 7           // shift target tile# left by 7
                    // to get an OCREQ begin address

    addu $4, $6, $18        // add the master OCREQ offset to
                    // get the beginning of OCREQs
                    // for this particular tile

    swlw $8, 0($4)          // number of requests is the
                    // first word in the list.
    bne $8, $0, ocreq_not1st

    sll $6, $9, 2               // $6 = tile # times 2
    la $7, hot_pending_request_table
    addu $6, $7, $6             // $6 = address in hot_table
    swlw $7, 0($6)

    bne $7, $0, ocreq_1st
.global ocreq_hot_begin ocreq_hot_begin:
    // add to hot-pending-request, and send hot msg to interrupt ctrl
    // if request is an invalidate, add it to the hot_pending_request_table.
    // otherwise, store 0x10000000 to the location.

    addu $7, $2, $11        // $7 = address and request
    li $4, 0x10000000
    swsw $4, 0($6)

    andi $4, $7, 1
    bne $4, $0, ocreq_hot_begin_flush

    swsw $7, 0($6)          // store request
ocreq_hot_begin_flush:
    rlm $8, $9, 3, 0x60
    ori $8, $8, 3           // $8 = I-tile handling this U-tile (same
                    // row, col = 3)

    li $6, TILE_NUM
    bne $6, $8, ocreq_send_hot

    // add to hot-map
    andi $6, $9, 0x3
    sll $6, $20, $6
    or $28, $6, $28

    // add to hot_table
    rlm $6, $9, 4, 0x30     // strip column of user-tile
    ori $6, $6, TILE_ROW        // $9 = uCOL.uCOL.dROW.dROW.0.0

    la $8, hot_request_table    // store request in hot_table
    addu $8, $8, $6

    swsw $7, 0($8)

    j ocreq_done

.global ocreq_send_hot ocreq_send_hot:
    li $6, (3 << 24)
    or $cgno, $8, $6        // dynamic message to I-tile,
    addiu $cgno, $0, DI_HOT_OP  // send hot-request to I-tile
    rlm $6, $9, 2, 0x0C     // strip column of user-tile
    ori $6, $6, (TILE_ROW >> 2) // $9 = uCOL.uCOL.dROW.dROW
    addu $cgno, $6, $0      // send U-tile col & D-tile row
    addu $cgno, $7, $0      // send actual request

    j ocreq_done

ocreq_1st:
    // number of requests increases from zero to 1.  We must contact
    // the interrupt tile when this happens.  If the # of requests
    // increases, say, 11 to 12, then what that means is that the
    // phone-call has been made, and we do not need to make another,
    // its just that multiple requests will be told to the U-tile
    // when the U-tile calls back.

    rlm $7, $9, 3, 0x60     // shift column over by 3 to get
                    // it into proper coord form
    ori $7, $7, 3           // this is the I-tile handling
                    // this particular U-tile (same
                    // row, col=3)

    // if the I-tile is the same, physically, as the D-tile, then do
    // not send a message.  This is to avoid a deadlock in the GDN
    // caused by $cgno being dependent on $cgni!

    li $6, TILE_NUM
    bne $7, $6, ocreq_send_request

    rlm $7, $9, 3, 0x18     // shift up the U-tile number
                    // and mask only the column,
                    // times eight
    sllv $7, $14, $7        // shift dirtile's number by
                    // this amount

    or $16, $7, $16         // and mask in the new request

    j ocreq_not1st

ocreq_send_request:


    // the request to the I-tile is two words; one a GDN header, and
    // a data word.  The GDN header says "1 byte long, sending to the
    // I-tile".  The data word is the current D-tile.  What the GDN
    // message says is "D wishes to speak to U."  The message will
    // be of the form UUDDDD, where UU is a two-bit indicator of the
    // U-tiles column, and DDDD is the one-hot address of the
    // current D-tile this code is running on.

    sll $6, $20, 25         // $6 = 2 << 24
    or $cgno, $7, $6        // dynamic message to I-tile,
                    // size 2

    addiu $cgno, $0, DI_LIST_OP // 0x300 is the opcode

    sll $7, $9, 4           // shift up the U-tile number
    or $cgno, $14, $7       // and swap in the dirtiles
                    // number, which is stored in
                    // $14... request sent
ocreq_not1st:

    // at this point, $4 still contains the beginning offset of the
    // U-tiles OCREQ.  We must add to that the correct number of
    // words so that we end up at the first blank spot.  The number
    // of requests (times four) is stored in $8.

    addiu $8, $8, 4         // increase number-of-reqs by
                    // four to save some instrs in
                    // word addressing land.
    swsw $8, 0($4)          // and write this number back to
                    // main RAM.

    addu $4, $4, $8         // first open OCREQ slot
    addu $7, $2, $11        // address and request - note
                    // that since the address ends
                    // in 1, the request is now 3
                    // for an invalidate and 2 for
                    // a flush
    swsw $7, 0($4)          // store request.

    // here, we do bounds checking.
    li $6, 120          // max number of requests, times
                    // four.

    bne+ $8, $6, ocreq_done     // this branch is almost always
                    // taken, even if the compiler
                    // doesn't think so

    // we just wrote in the last request into the table.  Oh dear.
    // Set the stall condition.  If any OCREQ set (for any one tile)
    // is filled, then we no longer take OCREQs for ANY tile.  This
    // is a bit overcautious, but given that OCREQs are ideally
    // passed off kinda quickly, this should be reasonable ($$$ test
    // this performance)

    // $13 contains one bit for each tile that has its OCREQ table
    // full.  This one-hot address scheme is used because some
    // OCREQs communications (not this one, but say an invalidate
    // flurry!) could possibly set a lot of tiles to go from 30->31.

    sllv $8, $20, $9        // 1 << stalling tile number
    or $13, $8, $13         // set that bit

ocreq_done:
    jr $31

// Main routine begins here.
```

```
begin:

        mtsri   SW_FREEZE, 1
        la      $4, swcode
        mtsr    SW_PC, $4
        mtsri   SW_FREEZE, 0

    j init_begin
init_done:

    PASS(56)

.global mainloop mainloop:
    jal static_network_begin

    jal gdn_handler_begin
    jal gdn_handler_begin
    jal gdn_handler_begin

    sll $2, $16, 1      // ignore bit31 of UD-map
    or $2, $2, $28

    mfsr $3, MDN_BUF
    andi $3, $3, 1

    // skip IRQ handling if msg is on MDN output. otherwise,
    // our MDN interrupt may not be immediately sent to the
    // tile, and the tile may prematurely receive
    // GDN system messages.

    bne $3, $0, skip_irq
    // if there are pending IRQs, handle them

    mfsr $3, GDN_BUF
    andi $3, $3, 0x1F
    addiu $3, $3, -10
    bgez $3, skip_irq

    jnel $2, $0, pending_irq_begin
skip_irq:
    // SCAFFOLD testing
//  .set noat
//      mfsr $1, CYCLE_LO       // grab cycle count
//  PASS_REG($1)
//  .set at
    // end SCAFFOLD

    // SCAFFOLD deadlock detector avoided because it does an lw()
    //j mainloop
    // end SCAFFOLD

    jal deadlock_detector_begin

    j mainloop

    DONE(1)

.end begin

reg_1: .word 0

reg_2: .word 0

reg_3: .word 0

reg_4: .word 0

reg_5: .word 0

reg_6: .word 0

reg_7: .word 0

reg_8: .word 0

reg_9: .word 0

reg_10: .word 0

reg_11: .word 0

reg_12: .word 0

reg_13: .word 0

reg_14: .word 0

reg_15: .word 0
```

```
pending_addresses_remaining: .word 0

int_status_0: .word 0

int_status_1: .word 0

int_status_2: .word 0

        .swtext
        .align 3

swcode:

    nop         route $cEi->$csti
    nop         route $cEi->$csti
    nop         route $cEi->$csti
        j swcode                route $cEi->$csti

ocreq_table: .rept 512 .word 0 .endr

hot_pending_request_table: .rept 64 .word 0 .endr

hot_request_table: .rept 48 .word 0 .endr

pending_request_table: .rept 512 .word 0 .endr

pending_address_table: .rept 1024 .word 0 .endr

static_input_buffer: .rept STATIC_BUFFER_SIZE .word 0 .endr
```

A.4.2 User-Tile Code

The following code is loaded onto user-tiles, and contains the handlers for MDN and Event Counter interrupts. After the necessary initialization has been performed, the code jumps to the actual application via the "jal program_begin" instruction at the end of the "begin" routine.

```
// user tile.  Part of a shared-memory implementation.
// Levente Jakab
// 5/11/04 18.51
// Modified by Satish Ramaswamy
// 5/11/05

#define DEADLOCK_TRIGGER_CYCLES 19000

#define OP_UI_DONE  0x000 #define OP_UI_CALL  0x100 #define
OP_UD_CALL  0x200 #define OP_UD_DONE  0x400 #define OP_EC
0x600 #define OP_IU_HOT   0x080 #define OP_UD_HOT_DONE  0xE00

//#include "libints.h"
#include "module_test.h"

    .text
    .align 2

    .global begin
.ent begin

.extern gdn_user_flag .extern gdn_user_msg


///// PERFORMANCE DATA ////
.extern mdn_cycles .extern ec_cycles .extern _profile_state
///////////////////////////

// interrupt 3 handler
.global mdn_interrupt mdn_interrupt: j mdn_code .global mdn_code
mdn_code:

    // an external interrupt signals a long chain of transactions,
    // initiated by a directory tile.  First, a D-tile sends the
    // U-tile an interrupt, and the U-tile ends up here.  It needs
    // to figure out just who did that, so it sends an MDN message
    // to the I-tile in charge.  The I-tile sends back a list of
    // dirtiles that wish to talk to it, and also a possible spoof
    // flag, meaning that a data word just sent to the tile was
    // false, and further action must be taken.

    // At this point, the U-tile talks to each of the D-tiles via
    // the MDN, getting a list of addresses that must either be
    // flushed or invalidated.  It does so.

    // Finally, the U-tile sends to the I-tile a message saying that
    // its task is complete.  The I-tile now knows it is free to
    // interrupt the U-tile again.
// store some regs. DO NOT CHANGE $9.

isw $2, %lo(reg3_2)($0)
isw $3, %lo(reg3_3)($0)
isw $4, %lo(reg3_4)($0)
isw $5, %lo(reg3_5)($0)
isw $6, %lo(reg3_6)($0)
isw $7, %lo(reg3_7)($0)
isw $8, %lo(reg3_8)($0)
isw $9, %lo(reg3_9)($0)
isw $10,%lo(reg3_10)($0)
isw $11,%lo(reg3_11)($0)

isw $31, %lo(reg3_31)($0)

/////////// PERFORMANCE DATA ///////////////
mfsr $2, CYCLE_LO
isw $2, %lo(mdn_cycle_tmp)($0)
////////////////////////////////////////////

or $10, $0, $0

// temporarily remember last interrupt in $9
// DO NOT CHANGE $9 AFTER THIS LINE
ilw $9, %lo(int_state)($0)

// store the new interrupt state
isw $0, %lo(int_state)($0)

// first, send a message to the interrupt controller.  The
// message will be of length 1.
li $cgno, ((1 << 24) | INT_NUM)

// send along the tile's number and an opcode
li $cgno, (OP_UI_CALL | TILE_NUM)

// now, wait for the I-tile to send back the list of
// requests.  When it arrives, grab the requests relevant only
// to this tile, by rotating and masking appropriately.

// initially word stored in $7 by gdn_sys_receive
jal gdn_sys_receive

// DO NOT MODIFY $2!
```

```
    rrm $2, $7, TILE_ROTATE, 0xFF

    // read OP, and check if HOT request
    li $8, 0x10000000
    and $8, $8, $7          // $8 = 1 if normal interrupt
                    // $8 = 0 if hot interrupt
                    // DO NOT MODIFY $8!

    bne $8, $0, mdn_spoof_check

    // must be a hot-request, so store all the requests in 'hot_request_table'

    isw $7, %lo(hot_request_table_4)($0) // initial word already read into $7
    or $4, $cgni, $0
    isw $4, %lo(hot_request_table_3)($0)
    or $4, $cgni, $0
    isw $4, %lo(hot_request_table_2)($0)
    or $4, $cgni, $0
    isw $4, %lo(hot_request_table_1)($0)

.global mdn_spoof_check mdn_spoof_check:
    mfsr $4, GDN_BUF
    rrm $4, $4, 5, 0x1      // check 1st el in GDN_BUF
    beq $4, $0, mdn_check_memop
    // there is still a word on the GDN. Check if it is spoof msg,
    // vs. user msg
    or $3, $cgni, $0
    srl $4, $3, 12
    subu $4, $4, 0xF
    bnez $4, mdn_spoof_check_valid
    // this is a user msg
    or $4, $cgni, $0
    sw $4, gdn_user_msg($0)
    li $4, 1
    sw $4, gdn_user_flag($0)
    j mdn_spoof_check

.global mdn_spoof_check_valid mdn_spoof_check_valid:
    li $10, 1           // signal that spoof occurred

mdn_check_memop:
    // also check if EC interrupt is pending, so that we may
    // extract its address and take care not to invalidate it
    mfsr $7, EX_BITS
    andi $7, $7, 0x40
    or $3, $10, $7

    beq $3, $0, mdn_no_spoof

    // a spoof has occurred.  The last lw instruction was patently
    // false.  Therefore, invalidate the cache line in question
    // and back up the program counter to the lw.  We may use any
    // register except $2 and $8.

    // First, we must grab the program counter at which the interrupt
    // occurred.

    mfsr $3, EX_PC

    // now, we know that the interrupt occurred precisely 2 cycles
    // after the offending instruction.  So back up the program
    // counter by one.  We'll back it up another in a sec.

    // BUGFIX interrupts occurs 3 cycles later!

    //addiu $3, $3, -8
.global mdn_not_memop mdn_not_memop:

    // back up one instruction. BUGFIX: (backup 2 instructions)
    addiu $3, $3, -4
    ilw $4, 0($3)           // load the instruction there

    // now, we must make sure that the instruction is either a lw or
    // a sw.

    // bits 31-29 are either 010 or 001.  Thus, on 1xx, 000, or 011,
    // forget about it.
    rrm $5, $4, 31, 0x1     // grab the high bit
    bne $5, $0, mdn_not_memop   // exit on 1xx

    // high bit must be a zero.
    rrm $5, $4, 29, 0x3     // grab bits 30-29
    beq $5, $0, mdn_not_memop   // exit on 000
    xori $5, $5, 0x3        // flip low bits
    beq $5, $0, mdn_not_memop   // exit on 011

    // high bits must be either 010 or 001.  Next bits must be either
    // 000, 010, or 100.  Not 110, and certainly not xx1.
    rrm $5, $4, 26, 0x1     // check for xx1
    bne $5, $0, mdn_not_memop   // exit on xx1
    // must be xx0
    rrm $5, $4, 27, 0x3
    xori $5, $5, 0x3        // check for 110
    beq $5, $0, mdn_not_memop   // exit on 110

    // so what we really have here is an honest to goodness memory op
    // in $4.  Extract the address, thrashing regs $5, $6,and $7.
    // (and $31)
    jal grab_address_and_data
    mfsr $7, EX_BITS
    andi $7, $7, 0x40
    beq $7, $0, mdn_not_memop_inv
.global mdn_not_memop_shaddr mdn_not_memop_shaddr:
    // if event_counter interrupt is pending after this MDN int,
    // we must write the address to "shared_address" so as not to
    // prematurely invalidate.
    srl $7, $6, 5
    sll $7, $7, 3
    isw $7, %lo(shared_address)($0)

    beq $10, $0, mdn_no_spoof

mdn_not_memop_inv:
    // we have the address, so invalidate the cache line.
    ainv $6, 0

    // and set the program counter back to the offending lw/sw,
    // where the cache will miss and the tile will go back to being
    // in stall, once we return from handling the interrupt.
    mtsr EX_PC, $3

//BUGFIX: incorrectly subtracted 10 decimal
    andi $2, $2, 0xEF       // knock off the bit-4 mask.

//BUGFIX: check if event_counter interrupt is pending on a write
//        to the spoofed data. If pending, reset the event counter,
//        so that a false exclusive request doesn't occur.

    mfec $4, EC_WRITE_OVER_READ
    srl $4, $4, 16
    sll $4, $4, 2
    bne $3, $4, mdn_no_spoof

    mtec EC_WRITE_OVER_READ, $0
```

```
mdn_no_spoof:
    beq $8, $0, mdn_hot_request

    li $3, ((1 << 24) | 3)      // load the current dirtile
                    // to deal with.  Start at tile
                    // 3.  Add on a header meaning
                    // an MDN message of length 1.
mdn_ud_outer_loop:

    // see if we actually need to talk to this dirtile by masking
    // in the lowest bit.  If not, then bypass.
    andi $4, $2, 1
    beq $4, $0, mdn_ud_ocreq_noneed

    or $cgno, $3, $0        // send the header

    // and send the data word
    li $cgno, (OP_UD_CALL | TILE_NUM)

    // now wait for the dirtile's response.
    jal gdn_sys_receive
    or $4, $7, $0           // $4 now contains the number
                    // of requests we are expecting.
                    // This is our main iterator.
.global mdn_ud_ocreq_loop mdn_ud_ocreq_loop:

    or $5, $cgni, $0        // load an OCREQ.  This contains
                    // in bits 0 an opcode, and in
                    // bits 31-2 an address.

    andi $6, $5, 0x1        // grab the opcode.  The opcode
                    // is 10 for invalidate and 01
                    // for flush.

    srl $8, $5, 5           // clear lower 2 bits
    sll $8, $8, 3

    // $2 has the list of dirtiles we need to talk to
    // $3 has the current dirtile we are talking to
    // $4 has the number of the request for the current dirtile
    // $5 has the address
    // $6 has the requests's opcode

    // if the opcode is 10, we want an invalidate, and if it is
    // 01 we want a flush.

    ilw $7, %lo(shared_address)($0)

    beq $6, $0, mdn_ud_ocreq_invalidate

    // must be a flush.  We can flush with extra address low-bits
    // as the cache does not care.  This op coughs up the line and
    // writes it to main RAM.  It also invalidates it because
    // sometimes the line must be patched up.
.global mdn_ud_flush mdn_ud_flush:
    //beq $7, $8, mdn_ud_flush_conflict
    aflinv $5, 0

//mdn_ud_flush_conflict:
    //afl $5, 0

    // we have handled the flush.
    j mdn_ud_ocreq_handled

mdn_ud_ocreq_invalidate:

    // must be an invalidate.  Therefore, we invalidate the cache
    // line pointed to.
    beq $7, $8, mdn_ud_ocreq_invalidate_conflict
    ainv $5, 0

mdn_ud_ocreq_invalidate_conflict:

mdn_ud_ocreq_handled:

    // we've taken care of the current OCREQ, whether it be
    // invalidate or flush.

    addiu $4, $4, -1        // subtract one more request
                    // and loop if there are more
                    // coming.
    bne $4, $0, mdn_ud_ocreq_loop

mdn_ud_ocreq_ack:

    // acknowledge, to the dir tile, what we have just accomplished
    or $cgno, $3, $0        // send the header

    li $cgno, (OP_UD_DONE | TILE_NUM)
```

```
.global mdn_ud_ocreq_noneed mdn_ud_ocreq_noneed:
    srl $2, $2, 1           // shift right the list of
                    // dirtiles needing to talk.
    addiu $3, $3, 32        // and increment the current
                    // dirtile number.


    // if there are more dirtiles wanting to talk, then loop
    bne $2, $0, mdn_ud_outer_loop

    j mdn_ui_done

.global mdn_hot_request mdn_hot_request:
    ori $11, $0, 12

    li $3, ((1 << 24) | 0x83)   // load the current dirtile
                    // to deal with.  Start at tile
                    // 15.  Add on a header meaning
                    // an MDN message of length 1.

.global mdn_hot_ocreq_loop mdn_hot_ocreq_loop:
    addiu $3, $3, -32       // decrement dir-tile

    ilw $5, %lo(hot_request_table_1)($11) // copy first message to $5,

mdn_hot_ocreq_loop1:
    beq $5, $0, mdn_hot_ud_ocreq_noneed

    andi $6, $5, 0x1        // grab the opcode.  The opcode
                    // is 10 for invalidate and 01
                    // for flush.

    srl $8, $5, 5           // clear lower 2 bits
    sll $8, $8, 3

    // $3 has the current dirtile we are talking to
    // $5 has the address
    // $6 has the requests's opcode

    // if the opcode is 10, we want an invalidate, and if it is
    // 01 we want a flush.

    ilw $7, %lo(shared_address)($0)
    beq $6, $0, mdn_hot_ud_ocreq_invalidate

    // must be a flush.  We can flush with extra address low-bits
    // as the cache does not care.  This op coughs up the line and
    // writes it to main RAM.  It also invalidates it because
    // sometimes the line must be patched up.
mdn_hot_ud_flush:
    //beq $7, $8, mdn_hot_ud_flush_conflict
    aflinv $5, 0

//mdn_hot_ud_flush_conflict:
    //afl $5, 0

    // we have handled the flush.
    j mdn_hot_ud_ocreq_ack

mdn_hot_ud_ocreq_invalidate:

    // must be an invalidate.  Therefore, we invalidate the cache
    // line pointed to.
    beq $7, $8, mdn_hot_ud_ocreq_invalidate_conflict
    ainv $5, 0
    j mdn_hot_ud_ocreq_ack

mdn_hot_ud_ocreq_invalidate_conflict:

mdn_hot_ud_ocreq_ack:

    // acknowledge, to the dir tile, what we have just accomplished
    or $cgno, $3, $0        // send the header

    li $cgno, (OP_UD_HOT_DONE | TILE_NUM)

mdn_hot_ud_ocreq_noneed:
    addiu $11, $11, -4
    andi $2, $3, 0x60
    bne $2, $0, mdn_hot_ocreq_loop

.global mdn_ui_done mdn_ui_done:
    // we are done, so send along a final ack to the I-tile handling
    // us, letting it know that we are about to exit interrupt mode.

    // The message will be of length 1.
    li $cgno, ((1 << 24) | INT_NUM)
    // send along the tile's number and an opcode
    li $cgno, (OP_UI_DONE | TILE_NUM)

    isw $9, %lo(int_state)($0)

    ////////// PERFORMANCE DATA ////////
    ilw $2, %lo(mdn_cycle_tmp)($0)
    mfsr $3, CYCLE_LO

    subu $3, $3, $2             // $3 = # cycles spent in this int.
    lw $2, mdn_cycles($0)
    addu $2, $2, $3
    lw $4, _profile_state($0)
    beqz $4, mdn_restore_regs
    sw $2, mdn_cycles($0)

    blez $9, mdn_restore_regs
    // we were in EC interrupt beforehand, so adjust the ec_stall_cycles
    // so that we don't doublecount. $3 = mdn cycles on this interrupt.
    ilw $2, %lo(ec_cycle_tmp)($0)
    addu $2, $2, $3
    isw $2, %lo(ec_cycle_tmp)($0)
    ////////////////////////////////////


mdn_restore_regs:
    // restore regs
    ilw $2, %lo(reg3_2)($0)
    ilw $3, %lo(reg3_3)($0)
    ilw $4, %lo(reg3_4)($0)
    ilw $5, %lo(reg3_5)($0)
    ilw $6, %lo(reg3_6)($0)
    ilw $7, %lo(reg3_7)($0)
    ilw $8, %lo(reg3_8)($0)
    ilw $9, %lo(reg3_9)($0)
    ilw $10, %lo(reg3_10)($0)
    ilw $11, %lo(reg3_11)($0)
    ilw $31, %lo(reg3_31)($0)

eret

//$7 returns system GDN message
.global gdn_sys_receive gdn_sys_receive:
  or $7, $cgni, $0
  srl $8, $7, 12
  addiu $8, $8, -0xF
  bnez $8, gdn_sys_receive_end

  // must be user message. therefore, read in next word and store.
  or $7, $cgni, $0
  sw $7, gdn_user_msg($0)

  li $7, 1
  sw $7,gdn_user_flag($0)

  j gdn_sys_receive
gdn_sys_receive_end:
  jr $31


// interrupt 6 handler
.global sw_event_counter_interrupt sw_event_counter_interrupt: j
event_counter_code event_counter_code: .global hi hi:
    // an event-counter event records the cache line going from
    // 'clean' to 'dirty', meaning that an sw has caused main memory
    // to go out of date.  Since the event counter takes 6 cycles to
    // interrupt, and there are no consecutive sw calls, we must
    // check for a total of three sw ops in the instructions
    // following the one that trapped.

    // squirrel away registers $2-6, 31 as we will be destroying
    // them.

    // SCAFFOLD these two stored previously under abnormal
    // circumstances.

    intoff
    nop
    nop
    nop

    isw $2, %lo(reg6_2)($0)
    isw $3, %lo(reg6_3)($0)
    isw $4, %lo(reg6_4)($0)
    isw $5, %lo(reg6_5)($0)
    isw $6, %lo(reg6_6)($0)
    isw $7, %lo(reg6_7)($0)
    isw $8, %lo(reg6_8)($0)
    isw $9, %lo(reg6_9)($0)
    isw $31, %lo(reg6_31)($0)
```

```
mfsr $2, EX_PC
isw $2, %lo(reg6_eret)($0)

/////////// PERFORMANCE DATA ///////////////
mfsr $2, CYCLE_LO
isw $2, %lo(ec_cycle_tmp)($0)
////////////////////////////////////////////

li $2, 1
isw $2, %lo(int_state)($0)

// grab the event counter
// SCAFFOLD - uncomment.  User has responsibility of writing
// 0xFFFF into $3, and address of triggering instruction into
// $2. WE DO IT ANYWAYS.

//li $3, 0xFFFF
//addiu $2, $31, -0xC

mfec $3, EC_WRITE_OVER_READ

// SCAFFOLD - uncomment
srl $2, $3, 16          // program counter of triggering
                // instruction.  Two bits are an
                // off-by-word-versus-byte.
sll $2, $2, 2

mtec EC_WRITE_OVER_READ, $0 // reset event counter

addiu $3, $3, 1         // add one to number of events
                // to get number of extra sw's
andi $3, $3, 0xFFFF     // we look for...
                // 0, -1 or -2 (negative
                // because event counter counts
                // backwards)

// the event that triggered (PC=$4) must be an sw, by definition

// grab the instruction from imem - $3 contains the number of
// sw's detected, $2 the address of the triggering pc, $4 is the
// instruction found there.  We're really trying to minimise
// register use because we are in interrupt, so we have to store
// and restore them, which is overhead.
ilw $4, 0($2)
```

```
// now, $4 has the instruction.  This subroutine will return in
// $6 the address and in $5 the data.  $4 and $7 are destroyed,
// as is $31.
jal grab_address_and_data


// store shared address
srl $7, $6, 5
sll $7, $7, 3
isw $7, %lo(shared_address)($0)

// now we have the address in $6 and the data in $5. check to
// see if the address corresponds to shared memory. if not, we
// can safely exit.

li $7, 0x6000000
and $4, $6, $7
bne $4, $7, event_counter_done

// let's make an MDN header to the appropriate directory.  Bits
// 6-5 of the address are the row of the correct dirtile, and
// the column is "11".  The bits are put into slots 6-5.
andi $4, $6, 0x60       // bits correspond to slots - no shifting!
ori $4, $4, 0x3         // and mask in

// SCAFFOLD length should be 3
li $7, (4 << 24)        // message length is 3.


// SCAFFOLD should be $cmno
or $cgno, $7, $4        // header length and target,
                // sent!

// SCAFFOLD this is not needed
ori $cgno, $0, 0x600

// send the tile number (self addressed stamped envelope, an
// MDN header for a return of length 1)

// SCAFFOLD MDN
li $7, (1 << 24)
ori $cgno, $7, TILE_NUM

// and now send the address and data
```

```
    // SCAFFOLD MDN
    or $cgno, $6, $0

    // SCAFFOLD MDN
    or $cgno, $5, $0

    // send has finished, so turn interrupts on
    inton

.global event_counter_wait_first event_counter_wait_first:

    // check to see if there is anything on the MDN.  This is a
    // non-blocking wait that allows interrupts to fire.
        mfsr $4, MDN_BUF        // pending incoming data
    rrm $4, $4, 5, 0x7      // knock off "out" and other crap

        beq $4, $0, event_counter_wait_first

.global event_counter_wait_first_done
event_counter_wait_first_done:
    or $4, $cmni, $0

    // wait for an acknowledgement, and possibly invalidate or
    // flush.  Message 0 is none, 1 is flush, 2 is invalidate.
    beq $4, $0, event_counter_none_first

    andi $4, $4, 1          // grab low bit

    beq $4, $0, event_counter_inv_first
//BUGFIX - NEEDS TO INVALIDATED
.global event_counter_flush event_counter_flush:
    aflinv $6, 0

    j event_counter_none_first

event_counter_inv_first:

    ainv $6, 0

event_counter_none_first:

    // hey, that might have been it.
    beq $3, $0, event_counter_done
```

```
.global event_counter_extra event_counter_extra:
    // there must have been multiple stores... start rooting around
    // the instructions immediately following the program counter.

    // First, grab the LAST possible instruction (the one that was
    // interrupted) - we need to check only what come before it.
    ilw $3, %lo(reg6_eret)

    // immediately following THIS store, cannot be another store, so
    // discount that instruction and move on.
    //addiu $2, $2, 4

event_counter_more_addresses:
    addiu $2, $2, 4         // next instruction

    ilw $4, 0($2)           // grab it.

    // an SW looks like the following.  In bits 31-29 are 001.  In
    // bits 28-26 are 000, 010, or 100.  But not 110.  Do a set of
    // tests.
    rrm $5, $4, 29, 0x7     // grab high 3 bits.
    xori $5, $5, 0x1        // compare with "001"
    bne $5, $0, event_counter_more_addresses

    // looks like the top 3 bits are "001".  Now, is bit 26 a zero?
    rrm $5, $4, 26, 0x1     // 1 or 0
    bne $5, $0, event_counter_more_addresses

    // okay, 26 is a zero.  Good.  Now are 28 and 27 both 1, because
    // that is bad.
    rrm $5, $4, 27, 0x3     // bits 27 and 26
    xori $5, $5, 0x3        // compare with "11"
    beq $5, $0, event_counter_more_addresses

    // $4 is now known to be a store.  Do exactly as with the
    // first word.
    jal grab_address_and_data

    // now we have the address in $6 and the data in $5.

    // let's make an MDN header to the appropriate directory.  Bits
    // 6-5 of the address are the row of the correct dirtile, and
    // the column is "11".  The bits are put into slots 6-5.
    andi $4, $6, 0x60       // bits correspond to slots - no shifting!
    ori $4, $4, 0x3
    li $7, (4 << 24)        // message length is 3.

    intoff
    nop
    nop
    nop

    // SCAFFOLD should be $cmno
    or $cgno, $7, $4        // header length and target,
                    // sent!
    // SCAFFOLD this is not needed
    ori $cgno, $0, 0x600

    // send the tile number (self addressed stamped envelope, an
    // MDN header for a return of length 1)

    // SCAFFOLD MDN
    li $7, (1 << 24)
    ori $cgno, $7, TILE_NUM

    // and now send the address and data

    // SCAFFOLD MDN
    or $cgno, $6, $0

    // SCAFFOLD MDN
    or $cgno, $5, $0

    inton

event_counter_wait_next:

    // check to see if there is anything on the MDN.  This is a
    // non-blocking wait that allows interrupts to fire.
        mfsr $4, MDN_BUF        // pending incoming data
    rrm $4, $4, 5, 0x7      // knock off "out" and other crap

        beq $4, $0, event_counter_wait_next

    or $4, $cmni, $0

    // wait for an acknowledgement, and possibly invalidate or
    // flush.  Message 0 is none, 1 is flush, 2 is invalidate.
    beq $4, $0, event_counter_none_next
    andi $4, $4, 1          // grab low bit

    beq $4, $0, event_counter_inv_next

// BUGFIX - NEEDS TO BE INVALIDATED
.global event_counter_wait_next_flush
event_counter_wait_next_flush:
    aflinv $6, 0

    j event_counter_none_next

event_counter_inv_next:

    ainv $6, 0

event_counter_none_next:

    // and since we just found an SW, the next one cannot be one...
    // we want to skip that exhaustive test (8 instructions,
    // including some nasty branches!) as much as possible.
    //addiu $2, $2, 4

    // see if we're done yet (if we have hit the instruction that
    // got interrupted, we must be done)
    bne $2, $3, event_counter_more_addresses

event_counter_done:
    intoff          // turn interrupts off to avoid race conditions
                // with MDN disrupting performance data.
    nop
    nop
    nop
    li $2, -1
    isw $2, %lo(int_state)($0)

    ////////// PERFORMANCE DATA ////////
    ilw $2, %lo(ec_cycle_tmp)($0)
    mfsr $3, CYCLE_LO
    subu $3, $3, $2
    lw $2, ec_cycles($0)
    addu $2, $2, $3
    lw $3, _profile_state($0)
    beqz $3, event_counter_done_restore_regs
    sw $2, ec_cycles($0)
    ////////////////////////////////////

event_counter_done_restore_regs:
    // clear shared address
    isw $0, %lo(shared_address)($0)

    // Restore the regs we so conscienciously put away.
    ilw $2, %lo(reg6_2)($0)
    ilw $3, %lo(reg6_3)($0)
    ilw $4, %lo(reg6_4)($0)
    ilw $5, %lo(reg6_5)($0)
    ilw $6, %lo(reg6_6)($0)
    ilw $7, %lo(reg6_7)($0)
    ilw $8, %lo(reg6_8)($0)
    ilw $9, %lo(reg6_9)($0)

    // --------> WE NEED AN INTOFF HERE <-------

    ilw $31, %lo(reg6_eret)($0)
    mtsr EX_PC, $31

    ilw $31, %lo(reg6_31)($0)

    // and we're done.
    eret

// this routine takes an instruction that is an sw and extracts an
// address from it.  The instruction is in register $4.  Registers $5
// $6, $7 are used as temporaries.  $4 is destroyed as well.

grab_address_and_data:

    // First, extract the register number from $4.  The instruction
    // bits are as follows: OOOOOOAAAAADDDDDXXXXXXXXXXXXXXXX
    // O = opcode.  Bits 31-26.  Already verified as store.
    // A = address.  Bits 25-21.  This is the reg number we seek.
    // D = data.  Bits 20-16.  What is written.  Data reg.
    // X = offset.  Bits 15-0.  Dealt with later.

    // rotate right by 18 bits and mask by 0b11111000, leaving the
    // reg number multiplied by 8.  rrm is hella nifty.

    rrm $5, $4, 18, 0xF8

    // what we do here is grab the address of this big table.  We
    // add to that eight times the register number in the
    // instruction.  We jump to that address!  (Eight because each
    // part of the switch is two instructions, thus 8 bytes.)  This
    // gets the value stored in the register... either it is still
    // stored in the reg, or it is fetched from instruction memory.
    // Very fast.

    // The result goes into $5.  Note that the four ports ($24-27)
    // are blanked out - this is because attempting to re-read a
    // port would be disastrous at best.  Thus, things break but at
    // least don't hang.  Note to user: don't sw directly off a
    // network.  Also, $1 is reserved for assembler, so that also
    // may break, though it is supported, as at best it returns
    // trash.

// BUGFIX: must differentiate interrupt 3 & 6 registers

    la $7, grab_address_regtable_3  // address.
    ilw $6, %lo(int_state)($0)
    beq $6, $0, grab_address_and_data_skip
    la $7, grab_address_regtable_6

grab_address_and_data_skip:

    addu $7, $5, $7          // add in the reg's offset.

    jr $7

grab_address_regtable_3:
    or $5, $0, $0            // $0
    j grab_address_regdone
.set noat
    or $5, $0, $1            // $1 - reserved
.set at
    j grab_address_regdone
    ilw $5, %lo(reg3_2)($0)      // $2 - stored
    j grab_address_regdone
    ilw $5, %lo(reg3_3)($0)      // $3 - stored
    j grab_address_regdone
    ilw $5, %lo(reg3_4)($0)      // $4 - stored
    j grab_address_regdone
    ilw $5, %lo(reg3_5)($0)      // $5 - stored
    j grab_address_regdone
    ilw $5, %lo(reg3_6)($0)      // $6 - stored
    j grab_address_regdone
```

```
ilw $5, %lo(reg3_7)($0)     // $7 - stored
j grab_address_regdone
ilw $5, %lo(reg3_8)($0)     // $8 - stored
j grab_address_regdone
ilw $5, %lo(reg3_9)($0)     // $9 - stored
j grab_address_regdone
or $5, $0, $10          // $10
j grab_address_regdone
or $5, $0, $11          // $11
j grab_address_regdone
or $5, $0, $12          // $12
j grab_address_regdone
or $5, $0, $13          // $13
j grab_address_regdone
or $5, $0, $14          // $14
j grab_address_regdone
or $5, $0, $15          // $15
j grab_address_regdone
or $5, $0, $16          // $16
j grab_address_regdone
or $5, $0, $17          // $17
j grab_address_regdone
or $5, $0, $18          // $18
j grab_address_regdone
or $5, $0, $19          // $19
j grab_address_regdone
or $5, $0, $20          // $20
j grab_address_regdone
or $5, $0, $21          // $21
j grab_address_regdone
or $5, $0, $22          // $22
j grab_address_regdone
or $5, $0, $23          // $23
j grab_address_regdone
or $5, $0, $0           // $24 - csti
j grab_address_regdone
or $5, $0, $0           // $25 - cgni
j grab_address_regdone
or $5, $0, $0           // $26 - csti2
j grab_address_regdone
or $5, $0, $0           // $27 - cmni
j grab_address_regdone
or $5, $0, $28          // $28
j grab_address_regdone
or $5, $0, $29          // $29
```

```
    j grab_address_regdone
    or $5, $0, $30          // $30
    j grab_address_regdone
    ilw $5, %lo(reg3_31)($0)    // $31 - stored
    j grab_address_regdone

grab_address_regtable_6:
    or $5, $0, $0           // $0
    j grab_address_regdone
.set noat
    or $5, $0, $1           // $1 - reserved
.set at
    j grab_address_regdone
    ilw $5, %lo(reg6_2)($0)     // $2 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_3)($0)     // $3 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_4)($0)     // $4 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_5)($0)     // $5 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_6)($0)     // $6 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_7)($0)     // $7 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_8)($0)     // $8 - stored
    j grab_address_regdone
    ilw $5, %lo(reg6_9)($0)     // $9 - stored
    j grab_address_regdone
    or $5, $0, $10          // $10
    j grab_address_regdone
    or $5, $0, $11          // $11
    j grab_address_regdone
    or $5, $0, $12          // $12
    j grab_address_regdone
    or $5, $0, $13          // $13
    j grab_address_regdone
    or $5, $0, $14          // $14
    j grab_address_regdone
    or $5, $0, $15          // $15
    j grab_address_regdone
    or $5, $0, $16          // $16
    j grab_address_regdone
    or $5, $0, $17          // $17
    j grab_address_regdone
    or $5, $0, $18          // $18
    j grab_address_regdone
    or $5, $0, $19          // $19
    j grab_address_regdone
    or $5, $0, $20          // $20
    j grab_address_regdone
    or $5, $0, $21          // $21
    j grab_address_regdone
    or $5, $0, $22          // $22
    j grab_address_regdone
    or $5, $0, $23          // $23
    j grab_address_regdone
    or $5, $0, $0           // $24 - csti
    j grab_address_regdone
    or $5, $0, $0           // $25 - cgni
    j grab_address_regdone
    or $5, $0, $0           // $26 - csti2
    j grab_address_regdone
    or $5, $0, $0           // $27 - cmni
    j grab_address_regdone
    or $5, $0, $28          // $28
    j grab_address_regdone
    or $5, $0, $29          // $29
    j grab_address_regdone
    or $5, $0, $30          // $30
    j grab_address_regdone
    ilw $5, %lo(reg6_31)($0)    // $31 - stored
    j grab_address_regdone

grab_address_regdone:

    // add offset, which sits in $4 and needs to be sign extended.
    // $$$
    // is there a more efficient way of doing this?
    sll $6, $4, 16
    sra $6, $6, 16          // sign extended
    addu $6, $5, $6         // now $6 has the full address.

    // same thing, except grab the data now.  Rotate by 13, and
    // data ends up in $5

    rrm $5, $4, 13, 0xF8

    // what we do here is grab the address of this big table.  We
    // add to that eight times the register number in the
    // instruction.  We jump to that address!  (Eight because each
    // part of the switch is two instructions, thus 8 bytes.)  This
    // gets the value stored in the register... either it is still
    // stored in the reg, or it is fetched from instruction memory.
    // Very fast.

    // The result goes into $5.  Note that the four ports ($24-27)
    // are blanked out - this is because attempting to re-read a
    // port would be disastrous at best.  Thus, things break but at
    // least don't hang.  Note to user: don't sw directly off a
    // network.  Also, $1 is reserved for assembler, so that also
    // may break, though it is supported, as at best it returns
    // trash.

// BUGFIX: must differentiate interrupt 3 & 6 registers

    la $7, grab_data_regtable_3 // address.
    ilw $4, %lo(int_state)($0)
    beq $4, $0, grab_address_regdone_skip
    la $7, grab_data_regtable_6

grab_address_regdone_skip:
    addu $7, $5, $7         // add in the reg's offset.

    jr $7


grab_data_regtable_3:
    or $5, $0, $0           // $0
    j grab_data_regdone
.set noat
    or $5, $0, $1           // $1 - reserved
.set at
    j grab_data_regdone
    ilw $5, %lo(reg3_2)($0)     // $2 - stored
    j grab_data_regdone
    ilw $5, %lo(reg3_3)($0)     // $3 - stored
    j grab_data_regdone
    ilw $5, %lo(reg3_4)($0)     // $4 - stored
    j grab_data_regdone
    ilw $5, %lo(reg3_5)($0)     // $5 - stored
    j grab_data_regdone
    ilw $5, %lo(reg3_6)($0)     // $6 - stored
    j grab_data_regdone
    ilw $5, %lo(reg3_7)($0)     // $7 - stored
```

```
j grab_data_regdone
ilw $5, %lo(reg3_8)($0)     // $8 - stored
j grab_data_regdone
ilw $5, %lo(reg3_9)($0)     // $9 - stored
j grab_data_regdone
or $5, $0, $10          // $10
j grab_data_regdone
or $5, $0, $11          // $11
j grab_data_regdone
or $5, $0, $12          // $12
j grab_data_regdone
or $5, $0, $13          // $13
j grab_data_regdone
or $5, $0, $14          // $14
j grab_data_regdone
or $5, $0, $15          // $15
j grab_data_regdone
or $5, $0, $16          // $16
j grab_data_regdone
or $5, $0, $17          // $17
j grab_data_regdone
or $5, $0, $18          // $18
j grab_data_regdone
or $5, $0, $19          // $19
j grab_data_regdone
or $5, $0, $20          // $20
j grab_data_regdone
or $5, $0, $21          // $21
j grab_data_regdone
or $5, $0, $22          // $22
j grab_data_regdone
or $5, $0, $23          // $23
j grab_data_regdone
or $5, $0, $0           // $24 - csti
j grab_data_regdone
or $5, $0, $0           // $25 - cgni
j grab_data_regdone
or $5, $0, $0           // $26 - csti2
j grab_data_regdone
or $5, $0, $0           // $27 - cmni
j grab_data_regdone
or $5, $0, $28          // $28
j grab_data_regdone
or $5, $0, $29          // $29
j grab_data_regdone
```

```
    or $5, $0, $30          // $30
    j grab_data_regdone
    ilw $5, %lo(reg3_31)($0)    // $31 - stored
    j grab_data_regdone


grab_data_regtable_6:
    or $5, $0, $0           // $0
    j grab_data_regdone
.set noat
    or $5, $0, $1           // $1 - reserved
.set at
    j grab_data_regdone
    ilw $5, %lo(reg6_2)($0)     // $2 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_3)($0)     // $3 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_4)($0)     // $4 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_5)($0)     // $5 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_6)($0)     // $6 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_7)($0)     // $7 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_8)($0)     // $8 - stored
    j grab_data_regdone
    ilw $5, %lo(reg6_9)($0)     // $9 - stored
    j grab_data_regdone
    or $5, $0, $10          // $10
    j grab_data_regdone
    or $5, $0, $11          // $11
    j grab_data_regdone
    or $5, $0, $12          // $12
    j grab_data_regdone
    or $5, $0, $13          // $13
    j grab_data_regdone
    or $5, $0, $14          // $14
    j grab_data_regdone
    or $5, $0, $15          // $15
    j grab_data_regdone
    or $5, $0, $16          // $16
    j grab_data_regdone
    or $5, $0, $17          // $17
    j grab_data_regdone
    or $5, $0, $18          // $18
    j grab_data_regdone
    or $5, $0, $19          // $19
    j grab_data_regdone
    or $5, $0, $20          // $20
    j grab_data_regdone
    or $5, $0, $21          // $21
    j grab_data_regdone
    or $5, $0, $22          // $22
    j grab_data_regdone
    or $5, $0, $23          // $23
    j grab_data_regdone
    or $5, $0, $0           // $24 - csti
    j grab_data_regdone
    or $5, $0, $0           // $25 - cgni
    j grab_data_regdone
    or $5, $0, $0           // $26 - csti2
    j grab_data_regdone
    or $5, $0, $0           // $27 - cmni
    j grab_data_regdone
    or $5, $0, $28          // $28
    j grab_data_regdone
    or $5, $0, $29          // $29
    j grab_data_regdone
    or $5, $0, $30          // $30
    j grab_data_regdone
    ilw $5, %lo(reg6_31)($0)    // $31 - stored
    j grab_data_regdone

grab_data_regdone:

    jr $31


// main routine.
.global begin begin:

    // set up the event counter event to trigger interrupt 6
    aui $6, $0, 0x409

    // seed the lfsr
    li $3, 0xF
    mtec EC_LFSR, $3

    mtsr EVENT_CFG, $6
    li $6, kEVENT_CFG2_WRITE_OVER_READ_MASK

    mtsr EVENT_CFG2, $6
    mtec EC_WRITE_OVER_READ, $0

    // load the CACHE_STALL counter
    li $6, 0xffffffff
    mtec EC_CACHE_STALLS, $6

    // load vector of interrupt 3 handler and store it
    // in instruction memory.
    ilw $3, %lo(mdn_interrupt)($0)
    isw $3, (3 << 4)($0)

    // and the interrupt 6 handler...
    ilw $3, %lo(sw_event_counter_interrupt)($0)
    isw $3, (6 << 4)($0)

    // enable int 3 (MDN) and int 6 (event counter)
    mtsri EX_MASK, ((1 << 3) + (1 << 6))

    // initial interrupt state is -1
    li $3, -1
    isw $3, %lo(int_state)($0)

    isw $0, %lo(d_flush)($0)
    isw $0, %lo(d_inv)($0)

    // interrupts on
    inton

    // setup deadlock handler
    //li $4, 100000
    //li $5, 0
    //j setup_deadlock_handler

    // wait for directory tile & ctrl tile to start up
    li $2, 160000

wait_for_sys:
    addiu $2, $2, -1
    bne $2, $0, wait_for_sys

    jal program_begin
```

Bibliography

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

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

[3] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: a scalable cache coherence scheme. In *International Symposium on Computer Architecture*, April 1994.

[4] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*, chapter 1.2. Morgan Kaufmann Publishers, 2003.

[5] Levente Jakab. A shared memory system using the raw processor architecture. Master's thesis, M.I.T., 2004.

[6] James Psota. rmpi: a message passing library for raw. May 2004.

[7] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared memory. Technical report, Stanford University.

[8] Michael Bedford Taylor. Comprehensive specification of the raw processor. Technical report, M.I.T., 2003.

[9] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological con-

siderations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

6964-49