Advances in the M-Machine Runtime System

by

Andrew Shultz

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

June 1997

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .... [illegible]
William J. Dally
Professor
Thesis Supervisor

Accepted by .... [illegible]
A. C. Smith
Chairman, Department Committee on Graduate Theses

Advances in the M-Machine Runtime System

by

Andrew Shultz

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This document discusses advances and alterations in an operating system being written for the M-Machine, a multicomputer currently being designed at MIT. The operating system is designed to be lightweight and flexible, able to support a UNIX-like operating system layer interface to higher-level code while exposing machine primitives to user programs in a safe and efficient manner. Alterations to the memory coherence system are presented, as well as the exception system, low-level machine-state accesss, and the initialization sequence.

Thesis Supervisor: William J. Dally
Title: Professor

Acknowledgments

The M-Machine project gave me a chance to go from the small problems offered by classwork to an environment where I could see what real problem were. While the process was sometimes messy, I learned a great deal, and I thank the M-Machine team for the opportunity. I especially thank Yevgeny Gurevich for having patience with my ignorance when I started out, and for leaving the code fairly well commented when he departed. I also want to thank Bill Dally for hiring me onto the project with less than epic qualifications — I hope my work has since proved my worth. And Matthew Reynolds, for turning down the UROP that for me became a way of life — if he'd taken the job, maybe I would have gotten stuck in the basement of the Media Lab. Finally, I should thank Kathy Knobe for the excellent title.

Contents

1 Introduction **9**

2 M-Machine Hardware **11**

2.1 Address Space 12

2.1.1 Pointer Capabilities 12

2.1.2 Hardware Support for Sharing 14

2.2 Multiple Execution 15

2.2.1 V and H Threads 15

2.2.2 Sleeping Threads 15

2.3 Access to Hardware State 16

2.3.1 Events 16

2.3.2 Exceptions 16

2.4 Messaging 17

3 Software Model and Overview **18**

3.1 Multithreading 19

3.2 Invoking the Runtime System 20

3.2.1 Entry Pointers 20

3.2.2 Hardware Invocation 21

3.3 Data Structures 22

3.4 Impact of Machine Size 23

3.5 Major Functional Blocks 24

3.5.1 Memory Management 24

3.5.2 Thread Management 25

4 Access to Machine State: Configspace 26

4.1 Motivation 26

4.2 Thread Management 27

4.3 I/O 28

4.4 Contents of Configspace 28

5 Boot Sequence 30

5.1 Getting the Code onto the Machine 30

5.2 Initializations 31

5.2.1 Giving out the Global Address Space 31

5.3 Handler Setup 32

5.4 The Loader 32

5.4.1 The Simulated Loader 32

5.4.2 Future Loader Work 34

6 Exceptions 35

6.1 Types of Exceptions 35

6.2 Hardware Restrictions 36

6.3 Design of the Exception System 36

6.3.1 Exception Handler 37

6.3.2 Exception Minder 38

6.3.3 Concurrency Issues 38

6.4 Catastrophic Exceptions 39

6.5 Future Exception Handler Work 39

7 Cache Coherence and Reclamation 40

7.1 Overview of Cache-Coherence System 40

7.2 Concerns 42

7.3 The Life of a Page Frame 42

7.4 Modes 45

7.4.1 Normal Operation 45

7.4.2 Panic Operation 45

7.5 Future Development 46

8 Location and Direction **48**

8.1 The M-Machine Runtime System and OS Design 48

8.2 Present Location 49

8.2.1 Left to be Coded 49

8.3 Working on a Moving Hardware Target 50

8.4 Future Work 50

8.4.1 I/O 51

8.4.2 UNIX Library 51

8.4.3 Optimization 51

List of Figures

4-1 Installing an H-Thread 29

5-1 Installing a handler 33

6-1 Exception Handler and Minder Interaction 37

7-1 Software Event Table 41

7-2 Cache-Coherence: the Simple Case 42

7-3 The Life of a Page Frame 43

List of Tables

2.1 Pointers and their Uses 13

Chapter 1

Introduction

The M-Machine is a new parallel computer being developed at the MIT Artificial Intelligence Laboratory in order to test a variety of architectural concepts. The design of the M-Machine hardware is very different from most conventional systems, and so requires a custom runtime system. The runtime system provides the underpinnings necessary to implement a more familiar operating environment, such as UNIX. It also exports some of the novel hardware features for use by user programs in a secure fashion. The runtime system must be flexible, so that all of the machine's features can be exploited, without making future implementation of higher-level system code difficult.

The runtime system is designed not as a single monolithic block, but as a collection of cooperating pieces. While this follows the general trend in OS design, it is not only a convenient way to manage a large and complicated software system, but also necessary for the M-Machine's unique architecture. The architecture of the M-Machine is discussed in chapter 2, with an emphasis on the aspects which most effect the shape of the runtime system.

The runtime system sits directly on top of the M-Machine hardware, and as the hardware has been through a few convulsions before release, the runtime system has had to change at its lowest level to interact with the machine. So while the design of the runtime system described in [5] has changed very little, the implementation has been through several revisions. One of the areas in which hardware changes had the greatest impact was in the boot sequence, which, in any case, was not fully specified before. There is now a clear path from the M-Machine powered up to the M-Machine running user programs. Runtime system access to the machine hardware is discussed in chapter 4, and the boot sequence is discussed in chapter 5.

[5] discussed the low-level memory-coherence manager and protocol at length, but the reclamation portion was left for future work. This piece has been designed and implemented, completing the memory-coherence manager. This new work is described in chapter 7, including a high-level overview of the rest of the cache-coherence manager to place the reclamation in context.

Additional portions of the runtime system which were completed earlier and described in [5] are the physical and virtual memory manager and the thread manager. These have been discussed at length in [5], and will not be covered here except in the most general way. However, there will be some overlap with that document, especially where the new features of the runtime system include revisions to prior work.

A final M-Machine hardware feature which was not serviced by the previous incarnation of the runtime system is exceptions. Exceptions are faults which are not recoverable: for example, while an LTLB miss is an event, an improper store target is an exception. A general discussion of the M-Machine's event/exception model can be found in chapter 2. The exception handler processes exceptions and provides information to the rest of the runtime system about the nature of the error. The constraints and design of the exception handler are discussed in chapter 6.

The final chapter of this work details the present status of the project, and discusses future work to be done, including both necessary work, such as an I/O structure and interface to host computer, and less critical suggestions for future work, such as possibilities for optimizations to the runtime system.

Chapter 2

M-Machine Hardware

In this chapter, the M-Machine hardware is described briefly, with emphasis on the features of the machine which make it unique from an operating system point of view. More detailed discussions of the M-Machine hardware can be found in [1] and [2]. Details of the ISA are available in [3].

The M-Machine is a 2D mesh of nodes connected by a high-speed network. While the initial implementation will have sixteen, the M-Machine is potentially scalable to thousands of nodes. Each node has some RAM and the MAP chip, a large (18mm square) chip specially designed for the M-Machine. The MAP chip has three clusters, a router, and a memory system. Each cluster has an integer unit and a memory unit, and one cluster also has a floating-point unit. The clusters and the memory system communicate with each other through two separate switches: one for intercluster communication, and one for cluster to memory communication.

As well as the three clusters, the MAP chip supports five thread contexts in hardware simultaneously, which may issue instructions on any or all of the clusters. The peak issue rate is therefore seven operations per clock cycle: three integer operations, three memory operations, and a floating point operation on the cluster enabled with an FPU.

The inherent multiple execution of the M-Machine is part of what makes the system a unique environment for its system software. There are two other elements which deserve to be mentioned. The first is the single address space, made possible through the use of guarded pointers. The second is the event model, through which events and exceptions do not interrupt a user thread, but are instead forked to a system thread, which deals with them asynchronously.

2.1 Address Space

The M-Machine has a single global 54-bit address space across all nodes and all threads. Because of this, every node not only has a local translation look-aside buffer (LTLB), but also a global translation look-aside buffer (GTLB), which keeps track of the home nodes of portions of the virtual address space. This GTLB is initialized at boot time, but may be modified on the fly by system software to cope with changing memory needs. Just as there are LTLB misses to cope with unfound local translations, there are also GTLB misses to cope with the global equivalent.

Because there is only a single address space, a given address is a unique identifier, pointing to one particular memory location on the M-Machine. Thus, threads may access data from other threads, even threads on other nodes, if they are given the address where the data is stored. System software provides for coherence between nodes, and the MAP chip hardware provides coherence on one node, without software intervention.

2.1.1 Pointer Capabilities

In order for the M-Machine to have a single global address space, capabilities are used for memory protection. Using capabilities avoids cumbersome access data structures, which have to increase in size as larger memory spaces are used, a problem on a machine which can potentially have hundreds of nodes and a 64 bit address space. Memory words have an additional bit which indicates whether the 64 bit value stored is a pointer, and the top ten bits of a word tagged as a pointer are interpreted as four bits of permission type and six bits of length encoded logarithmically. While system processes may tamper with pointers as they wish, untrusted user processes may only copy them or modify the address portion within the memory segment delimited by the length field. The determination of what privileges a given process has is determined by its IP: there are different pointer types for execute in user mode and execute in system mode. This allows a given thread to switch from running in user mode to running in system mode merely by jumping to a different IP, enabling the use of system functions in user thread slots. This method is discussed in greater detail in chapter 3.

Pointer Type	**Description**
Read	Read only data.
Read/Write	Read/Write data; the standard data pointer.
Execute User	Code to be executed in user mode. Priviledged operations will cause exceptions.
Execute System	Code to be executed in system mode. All operations are acceptable.
Enter User	A pointer which may be jumped to, but not altered. Upon use as a jump target, it transforms into an execute user pointer.
Enter System	A pointer which may be jumped to, but not altered. Upon use as a jump target, it transforms into an execute system pointer.
Execute Message	A pointer which may be sent to be executed upon message receipt.
Key	A pointer which may not be used for execution or data. Because it is a pointer, it may not be forged, and so it may be used as a key.
Physical	A pointer to physical, rather than virtual, memory. It may be used for data or execution in priviledged mode.
Error	An erval pointer, indicating returned by an operation which could not complete normally.
Configspace	A pointer which maps to hardware state on the MAP chip.

Table 2.1: Pointers and their Uses

Since pointers may not be created by user processes, the system needs to maintain only one page table for all threads, making it easy to switch between threads and run different threads simultaneously without worrying about the security of their memory spaces.

Deallocation of a virtual address segment, however, is much more complicated. Since threads may pass pointers among themselves, the deallocation of a segment by one thread does not guarantee that no other thread has a pointer which would allow them to access that segment. Only a global garbage collection could have enough certainty to allow such a segment to be reused. Luckily, with a large address space, reclamation need not be frequent, and it is quite likely that it will be a quicker and easier process to merely reboot the machine. The construction of an efficient garbage collector for the M-Machine could be an interesting future project.

While the M-Maghine has many different pointer types, of specific interest to the runtime system are the entry pointer types, which allow for user threads to indicate system functions to be called, without being able to run the code themselves, and key pointers, which are useful as magic cookies: unforgeable, and convertible by the system into real pointer types. Key pointers are used extensively in the thread management portion of the runtime system, for user threads to identify themselves to the runtime system.

2.1.2 Hardware Support for Sharing

In addition to guarded pointers, the M-Machine has hardware support for sharing memory. Each block of memory (eight words) has two status bits, which may encode invalid, read, read/write, or dirty. If an access is attempted which violates these bits, the memory system does not complete it, but instead uses a hardware mechanism to alert the system software. This software can then satisfy the access by, for instance, fetching data from a remote node.

Another way in which the M-Machine provides support for sharing memory is through atomic test-and-set memory operations. Each memory word has associated with it a sync bit which may be used to perform loads and stores conditional on the status of that bit, setting the bit afterward. The failure mode depends on the specific operation: some generate events for the system software to fix, while others merely indicate failure in a condition register. Again, these instructions are detailed in [3].

2.2 Multiple Execution

The most significant influence on the structure of the runtime system is the multiple execution present in the MAP chip. With three clusters, nine functional units, and six hardware thread slots, there is a great deal going on at once in the M-Machine. The M-Machine cannot afford to halt all user processing for normal system functions, which are handled as interrupts on most machines, because to do so would be to give up the use of the multiple issue hardware which is the focus of the project. So the MAP chip and the system software are made to handle events simultaneously with normal user processes.

2.2.1 V and H Threads

The M-Machine has two distinct levels of parallelism: V-threads and H-threads. V-threads are what reside in the hardware thread slots, and are swapped in and out by the runtime system's thread manager. V-threads are protected from one another, and may not access each other's registers. Of the five hardware thread slots, three are reserved for system threads.

H-threads are subunits of the V-threads, running on individual clusters: thus, a V-thread may consist of one to three H-threads. H-threads are able to access registers of other H-threads in the same V-thread, and H-threads may be spawned off and terminated as desired by a user level program without system software intervention.

During any cycle, the three clusters may be performing operations from different V-threads. Which V-thread may issue on each cluster is chosen fairly by the hardware from the set of threads which are ready to issue.

2.2.2 Sleeping Threads

Threads which are not ready to issue are generally waiting for the result of an earlier operation. When the MAP chip detects that a register is the target of an operation, that register is marked as empty. Operations may not issue, of course, until their operand registers are full. While this is automatically used in order to wait on returning loads and other longer-latency operations, it is also often used by the system software to keep a thread from issuing when it has no work to perform. Since threads which are not ready to issue do not use up execution cycles, the system may in this way wait for events without using up machine resources by spinning in a polling loop. When a register is emptied for this purpose, the thread is said to be sleeping on that register. The M-Machine provides specific instructions for emptying registers, and many of the hardware mechanisms for communicating with system software involve filling specific registers on which the software is sleeping.

2.3 Access to Hardware State

The M-Machine has a large amount of hardware state which is memory mapped using a special pointer type, the configuration space pointer. Through this method, system threads may access machine state such as the cycle counter and the register files of other V-threads. System use of configspace is discussed more extensively in chapter 4. Configspace pointers may never be constructed by user threads, just as pointers in general may not be constructed by user threads, so the hardware is protected from untrusted code.

2.3.1 Events

Events are hardware occurrences which the M-Machine expects to occur as a normal part of running a program, such as LTLB misses or synchronization misses. Events are mapped into the register files of system V-threads. This allows these threads to sleep, remaining ready to execute immediately without taking up any execution cycles.

2.3.2 Exceptions

Exceptions are expected to be fatal to the program which experiences them: occurrences such as pointer permission violations or illegal instructions. One of the system

V-threads is an exception handler, which in the original MAP design had a reduced set of registers, though these registers have since been increased to the full complement. Like events, exceptions are written into registers, awakening the exception handler thread. The exception handler is discussed in chapter 6.

2.4 Messaging

The M-Machine provides instructions which allow atomic message sends from a system level, and a subset of these instructions which may be used at user level. This allows user programs to send messages without system-software intervention on the dispatch side, although system code will always be executed on the receiving side, to ensure that the incoming message queue is cleared. User threads are only permitted to send messages to specific system message handler routines or to virtual addresses. The runtime system uses messages a great deal in the memory-coherence manager, discussed in brief in chapter 7, and more extensively in [5].

Chapter 3

Software Model and Overview

The M-Machine runtime system has a structure which is very different from a traditional operating system, both because of the unique nature of the machine and because this system is being built from the ground up, instead of being a revision of a previous operating system. The runtime system is designed to be lightweight, to tolerate concurrency, and to safely export the capabilities of the M-Machine hardware to user applications.

The M-Machine runtime system supports multiple concurrent user threads running in a single global address space, including memory coherence across multiple nodes. It is this support for concurrency, which includes system threads running side by side with user threads, which makes the M-Machine system so different from more traditional systems. Part of this concurrency is reflected in the ways in which system components communicate, and how user threads communicate with the system. The M-Machine runtime system uses protected function calls and aspects of the hardware to support most communication, but also makes some use of the ability of multiple threads to easily share access to memory segments.

This chapter discusses the issue of concurrent runtime and user threads in greater depth, as well as providing an overview of the methods by which components communicate. It goes on to discuss the impact of the specific organization of the hardware, including some of the changes in the runtime system caused by recent alterations to the hardware. Finally, an overview of the major functions of the system is given, although for greater depth the reader should refer to [5] for components not covered in this document.

3.1 Multithreading

Perhaps the most important difference between the M-Machine runtime system and more traditional operating systems is its support for concurrency. Multiple system threads and multiple user threads may all be issuing instructions simultaneously. This simultaneity includes both interleaving operations from different threads on a particular cluster on a cycle by cycle basis and running operations from different threads on different clusters during the same cycle. Depending on the methods of access, there may even be multiple copies of a particular system function running at once, so that function must be designed to be reentrant. Other system functions, invoked by hardware, are limited to only one copy at a time.

The M-Machine relies on this level of parallelism in the system code because of the high level of parallelism in the hardware, as discussed in the previous chapter. Unless the runtime system can fully make use of the concurrency offered by the hardware, the effort put into hardware design will be lost. Since the system alone does not need all of the resources of the machine for simple fault handling, it therefore must be able to share these resources with user threads so that they do not go to waste.

For instance, a user thread which has faulted may be able to make progress, even while the system is handling the problem. If a load instruction to register ten causes an LTLB fault, the system must be invoked to handle the problem and write a new mapping into the LTLB. While this is occuring, the faulting thread may continue to issue instructions until it absolutely needs the data that was to be placed in register ten. Hardware flags indicate whether a given register has data: registers are emptied when they are targeted, and marked as full when the data arrives. This is true even when the system software has to be invoked to retrieve the data.

The M-Machine calls hardware occurrences of this type events. Events require the intervention of system software, but are expected to be dealt with in a satisfactory manner. LTLB misses, block status misses, GTLB misses, and synchronization misses are all events. In the case of an event, the hardware can be thought to be forking the event into a system thread slot. Actually, what it is doing is awakening an already present but sleeping system thread, but the end result is the same: the user thread and the system thread which handles the event run concurrently.

Another type of hardware occurrence is the exception, which include things such as protection violations. Exceptions are not expected to have clean resolutions, and will probably result in the termination of the program which caused them. However, while the user thread which caused an exception is stopped, the forking model is still relevant, since other user threads may continue to issue concurrently with the exception handler. Exceptions and the exception handler are discussed in depth in chapter 6.

Potentially the M-Machine could have an exception handler, LTLB miss handler, synchronization miss handler, two message handlers, and two user threads running simultaneously.

3.2 Invoking the Runtime System

Obviously the M-Machine runtime system cannot support both a more usual interrupt model of system invocation and the level of rampant concurrency just discussed. There are two methods by which the system may be invoked. The first is to make use of the M-Machine's protected pointers to allow user threads to securely enter system code. The second is the more traditional method of hardware fault response, with, of course, some important differences when implemented on the M-Machine.

3.2.1 Entry Pointers

Entry pointers are the M-Machine's method for allowing user threads to enter system code in a secure manner. An entry pointer may be jumped to, but not changed or used as data addresses. User programs have a table of system entrance points, but the pointers stored in these system entrance points are not valid pointers at all, but magic numbers. When the user program is loaded, the loadedr recognizes this data as magic numbers, and the numbers are replaced with entry pointers from a table in the system. In this manner, the user program may not be compiled with false entry pointers, because false magic numbers will not be recognized by the loader. As an added benefit, when new system calls are added, old programs will still work, but new programs will have access to the new system functions. And moving system functions is trivial, since the patching is performed so late in the process.

There is still one security hole which will have to be patched in the future, which is that system functions in user slots use the user thread's stack. This potentially allows another user thread, which has been passed that stack pointer, to snoop the system's stack. The solution for this is to allocate special system stack spaces, and switch to them on entry to system code. Presently, entry to system code includes a memory barrier instruction, so that user loads and stores must complete before the system code continues, preventing possible returning user loads from fouling up the runtime system. In the future, system code which uses the stack will also spend a few cycles switching to a system stack space.

System functions which are executed in user slots are generally those which are invoked by the deliberate actions of user threads. These included thread operations such as fork and sleep, virtual memory operations alloc and dealloc, and a few stubs to get the attention of the hardware fault handlers, such as the ability to signal the event handler by placing a single word into the hardware event queue.

3.2.2 Hardware Invocation

Most important runtime system procedures are invoked in response to hardware events. These portions of the runtime execute in the system dedicated thread slots and are always present, but use the hardware sleeping mechanism to avoid taking cycles from user threads. The particular registers which they use are not normal registers, however, but hardware state which takes the place of registers in the normal register file.

As a specific example, the LTLB miss handler resides in thread slot five, cluster zero. In that cluster, registers eleven through fifteen are reserved for LTLB miss information, which is automatically written to those registers when a miss occurs. If there is no LTLB miss, these registers are marked as empty. The LTLB miss handler, after setup, waits on register eleven. When an LTLB miss occurs, the code may move forward, and when complete it returns to waiting on that register. The event handler has a similar mechanism, although in its case the special registers are the top of a queue of events. The message handlers sit on incoming message queues in a similar fashion.

Because these handlers may only be invoked in response to hardware, but sometimes may be needed for other tasks (if, for instance, the event handler wants the LTLB handler to map a page of virtual memory to a physical backing), there are ways for other threads to invoke the hardware in order to get at these handlers. In the case of the LTLB handler, this involves creating an LTLB miss to virtual memory locations that is guaranteed never to be mapped. Since virtual addresses cannot be created by user threads, this interface is secure and limited to system functions. The event handler has a software queue of events which user threads can write into, and system functions may through configspace write into the hardware event queue. Since writing one word into the event queue is atomic, a system funciton may safely write a one-word signal event into the hardware queue without worrying that it might be interrupted by a genuine hardware event. This signal event awakens the event handler, which does nothing in response to the signal event, but checks the software queue before again sleeping. Again, writing this signal word into the event queue requires access to special addresses (configspace, discussed in chapter 4) and so can only be used by system functions. These system functions may be invoked in a user thread slot, as described above, and in fact the signal word is only used by user threads.

3.3 Data Structures

With multiple system functions occuring at the same time, data sharing and deadlock among the threads become very serious considerations. Because of this, the

M-Machine runtime system depends on the hardware synchronization operations to provide locking operations. When it isn't safe to use locking accesses to data, the runtime system relies on data structures which are non-locking, yet provide safe access methods.

The M-Machine's locking mechanism uses the synchronizing memory operations. Since they are atomic, this provides a safe lock. However, threads like the event handler cannot wait on user threads, since a user thread might be waiting for the resolution of some event to make progress. In such cases, a non-blocking data structure must be used, such as a queue. With the software event queue, there is a lock to prevent more than one user thread from putting something into the bottom of the queue, but only the event handler takes from the top, and so does not need to bother with a lock. Only user threads place events into the queue, and it is safe for them to compete over locks: the system will still make progress.

In other cases, one thread has access to a data structure despite other locks. The LTLB handler may access memory data structures freely, since potentially the entire machine could be waiting on an LTLB miss to be resolved. Other system threads accessing these structures must acquire a lock. If the other threads receive inconsistent data due to the interference of the LTLB handler, it will cause a later LTLB miss, and the LTLB handler will be able to resolve the problem. Because the LTLB handler is the final authority on LTLB mappings, it can be confident that any problems will be later resolved properly.

3.4 Impact of Machine Size

The size of the M-Machine has some effect on the structure of the runtime system. While the number of nodes makes little difference except with regards to the layout of the global virtual memory space, the parallelism available within the MAP chip does effect the runtime system. The original design for the chip had four clusters, while the present design has only three due to space constraints on the final silicon.

The runtime system requires one H-thread for each handler, which make four in total, and one H-thread per cluster for the exception handler (which will take up a whole V-thread, of course). On the original MAP chip, the four handlers occupied one V-thread, and the exception handler another. With three clusters, the LTLB and message handlers occupy one V-thread, the exception handler another, and the event handler one H-thread of a third. This leaves two H-threads unoccupied, yet not available for user threads. The exception system presently uses one of these for speed, but it could return to using only its own V-thread if a better use were to be found.

If the MAP chip were to grow beyond four clusters or beyond, the runtime system would not experience much increase in efficiency, although additional H-threads would be available.

3.5 Major Functional Blocks

The runtime system can be grouped into two large functional areas: memory management and thread management. These functions are divided among the different event handlers. In general, a handler is a piece of software residing in a particular location on the M-Machine, whereas a manager is the collection of routines for performing a particular function, and may be executed within several different handlers and user threads.

This section presents and overview of these functional blocks. For a more complete description of the particulars of these systems, see [5].

3.5.1 Memory Management

Memory management consists of the physical page frame management routines, the LTLB miss routines, virtual memory management, and cache-coherence.

Physical memory is organized into a giant list of free page frames, maintained within the free page frames themselves. The physical memory manager gives out these page frames as backing only on demand. This means that large segments of virtual memory can be allocated without using up physical memory unless it is actually required. The physical memory management routines reside entirely within the LTLB miss handler, as they are invoked in response to LTLB misses or through the fake miss interface. Since having a difference of opinion about memory mapping would be disastrous, no other handlers have access to the physical memory manager.

Virtual memory, however, may be handed out more freely, and indeed, virtual memory allocation is carried out in user thread slots. Virtual memory is managed using a buddy-list structure, whereby each segment can be split into two segments of half the size. Potentially this could waste almost half of any given segment (for instance, allocating 129 bytes will result in being given a segment of size 256), but this internal fragmentation is not much of a problem given the M-Machine's large virtual address space. And since physical pages are only given out on demand, little physical memory can be wasted.

The cache-coherence manager maintains a coherent picture of memory over multiple nodes. In doing so, it maintains two structures on each node: a list of which blocks it is sharing to other nodes, and their status, and a list of all blocks which it has received from other nodes. Blocks may be read only, read/write, dirty, or invalid, but the cache-coherence manager has a much larger number of states to deal with the messaging between nodes. Cache-coherence is often invoked in response to a block status miss, so some of it resides in the event handler. Other pieces are in the message handlers, to cope with incoming messages. The reclamation portion of the cache-coherence manager is new, and is described in chapter 7, along with a more thorough overview of the rest of the system.

3.5.2 Thread Management

Thread management is carried out almost entirely by the event handler. Threads (by which we mean V-threads) may be forked, exited, scheduled, and signaled, just as in an ordinary thread operating system. The thread manager may swap threads into the three user thread slots. Threads may even be spawned to remote nodes.

Chapter 4

Access to Machine State: Configspace

Most of the M-Machine's hardware state is accessible to system-level programs through a special set of memory addresses: configuration space. Configuration space is set off from the rest of memory through the type field of the pointer. These addresses are accessed using normal loads and stores, although the hardware mechanism involved is different than an ordinary cache interaction. This chapter will discuss briefly the motivation behind configspace, and provide a few specific examples of how the runtime system uses it. For a guide to the use of configspace by software, see [6].

4.1 Motivation

There are three main reasons to have a large amount of hardware state accessible in a uniform manner. The first reason is to support the parallel nature of the machine. The second is to make the design of the hardware more modular and easier to alter as the needs of the project change. The last is in support of the research nature of the machine.

Because of the parallel nature of the machine, system threads need the ability to access state residing in other threads while both are running on the hardware. In a more typical system, one thread would be swapped out and easily accessible to the other through normal memory system accesses, but on the M-Machine, it is more than likely that both are running. The specific example of thread management is discussed in greater length in the next section. In addition to those accesses, the LTLB is accessed through configspace, as well as the I/O and some state to control the running mode of the chip. By providing a simple way for the runtime system to access this state, the M-Machine avoids a potential problem with parallel multithreading.

The M-Machine could have special instructions for accessing the various parts of the hardware, but this would complicate the ISA and use up useful encoding space in the 32 bit instruction. By providing a more general mechanism, the design of the machine did not have to be greatly changed when additional state was needed.

In addition, configspace is used to control the operation mode of the machine while it is running. Tightly coupled mode, in which H-threads are run as though they were one VLIW instruction, may be controlled on a per-thread basis through configspace. Thread preemption thresholds and invalidate on miss mode are two more examples of different ways to run the M-Machine controllable through configspace. In this way, the M-Machine project may test the effects of different modes of parallel computation on the efficiency of our test programs and the runtime system itself.

4.2 Thread Management

Swapping a thread into or out of a hardware V-thread slot illustrates the use of configspace very well. This section will discuss the specific example of installing a V-thread which was previously running and had been swapped out.

The runtime system maintains a thread context data structure for each user thread. That structure stores information such as the parent and children of the thread, as well as signal data. The meat of the thread context, however, is in the three H-thread contexts, which store the hardware state of each H-thread, including register values, instruction pointers, and so on. Figure 4-1 shows the code used by the runtime system for installing an H-thread.

In order to reinstall a V-thread, the runtime system first finds a hardware thread slot which is empty (it may have just created such a slot by evicting a thread). It then installs the three H-threads in the following manner.

After making up a configspace pointer to the base of the H-thread's space, the runtime system first installs the register values for the integer, floating point, and condition registers. Since configspace is memory mapped, it may access these locations as though they were an array in memory, making coding and compilation easy.

It then installs H-thread state such as the priority of the H-thread, the preemption threshold and the register scoreboards. Finally, it places the instruction pointers into the pipeline, which requires some amount of reconstruction because of the way branches are handled in the M-Machine pipeline.

Again, all through this procedure the thread state can be easily treated as an array, or as a more specific data structure in memory if future implementations find this easier. Also, when the thread state changed, it required only simple changes to bring the runtime system up to the latest hardware specifications.

4.3 I/O

I/O is also memory mapped as part of configuration space. The M-Machine I/O system will perform single word or burst transfers over a connection to a host terminal. Again, the memory-mapped design means that special instructions are not necessary. The I/O portion of the runtime has not yet been designed.

4.4 Contents of Configspace

Also in configspace are the various run enable bits for user thread slots, which control the issue of user code. The cycle counter, keeping track of the number of clock cycles since machine restart, is accessible through configspace, as is an event counter for keeping track of hardware events. The hardware event queue can be written through configspace, allowing for the one word signal event previously mentioned. Finally, ECC mode control for the memory system can be accessed through configspace as

```
int tInstallHThread(int ht, struct HContext *hc, int slot) {
    int i;
    int *CP;

    CP = sysSetPtr(CFG_BASE_LT |
   (CFG_LTLC_PER_CLUSTER * ht) |
   (CFG_LT_PER_THREAD * slot));

    for (i = 2; i < 16; i++) {
      CP[i] = hc->int_reg_file[i];
    }

    CP += 16;
    for (i = 1; i < 16; i++) {
      CP[i] = hc->fp_reg_file[i];
    }

    CP += 16;

    for (i = 0; i < 8; i++) {
      CP[i] = hc->cc[i];
    }
    CP += 8;

    CP[3] = hc->priority;
    CP[4] = 0x0;    /* hc->PC */
    CP[5]  = hc->PT;
    CP[7] = 0x0;  /* hc->stall_bit  */
    CP[9] = 1 - (hc->int_empty_scoreboard << 48);
    CP[10] = 1 - (hc->fp_empty_scoreboard << 48);
    CP[11] = 1 - (hc->cc_empty_scoreboard << 48);

    if (isbranch(hc->restartIPvector[0]))
      CP[13] = hc->restartIPvector[0];
    else
      {
       CP[12] = hc->restartIPvector[0];
  CP[12] = hc->restartIPvector[1];
  CP[12] = hc->restartIPvector[2];
if (hc->bp)
  CP[12] = hc->restartIPvector[3];
else
  CP[12] = nextIP(hc->restartIPvector[2]);
      }
    return hc->hardware_mbar_counter;
}
```

Figure 4-1: Installing an H-Thread

well. For a complete list of the contents of configspace and their encodings, see [6].

Chapter 5

Boot Sequence

Taking the M-Machine from its blank starting configuration to running a user program is a complex process, involving several levels of bootstrapping and initialization code. During boot, the runtime system accesses almost all of the software-accessible hardware state in order to set up the system handlers and prepare for user code.

The boot sequence starts from a machine in the reset state. Then a very small piece of code is injected through the M-Machine's diagnostic interface. Since using this interface is extremely slow, only enough code is placed on the machine to begin to use the I/O system through configspace. The I/O system is then used to pull the rest of the system code into memory.

Once the system is resident in memory, the meat of the boot code begins to execute, including handler setup and data structure initialization. The runtime system uses the configuration space ability to write into different V-threads and clusters in order to place system handlers in their proper positions. Finally, the system is ready to load and patch user code.

5.1 Getting the Code onto the Machine

The diagnostic port which is used to perform the beginning code injection has the primary function of reading out all possible hardware state after stopping the machine during execution. In the boot case, however, the diag port is used in reverse to inject a few lines of code into memory and set the machine to begin execution. It does this by placing the code as packets onto the internal switches of the chip, which then perform ordinary interactions with the memory system.

This bootstrap code can be very small: ten instructions to create two pointers: one a configspace pointer to allow reads from the I/O, the other a memory pointer. Then a few more instructions are necessary to form an iterative loop. Once these are set up, the machine may load the rest of the system into memory more quickly. While using the diagnostic port takes hundreds of cycles for each word to be placed in memory, the I/O port takes only a few cycles per word. Because of this, it is well worth having this intermediate stage of loading rather than just more simply using the diag port. After system loading is complete, the real meat of the boot sequence may begin.

5.2 Initializations

Major data structures to be initialized are the local page table (LPT), which is the software backing for the LTLB, and the management structures for virtual and physical memory. The sizes of these structures are set at system compile time, except for the free page frame list, which is stored in the free page frames themselves.

5.2.1 Giving out the Global Address Space

The global address space also needs to be distributed to nodes as part of initialization. At present, the boot sequence determines which node it is running on and then initializes its virtual memory manager with a different segment depending on its node number. It also writes the GTLB with supporting information. While this is certainly adequate for the testing which is going on presently, later versions of the runtime system will need a more flexible mechanism. Most likely the host computer will govern the distribution of the address space, with individual nodes querying it for their piece during boot. This will also allow for memory to be distributed while the machine is running, something which cannot be done under the present system.

5.3 Handler Setup

In order to set up a system handler, the boot code first sets up the proper thread and cluster to fork to the starting handler IP. It then allocates a piece of memory for the handler's stack, and stores the stack address into a particular register. The address of the start of the system's data segment goes into another register, and finally the full and run bits are set for that location, allowing the thread to begin execution. Figure 5-1 shows an example of handler installation.

5.4 The Loader

Loading a user program on the M-Machine is somewhat complicated because of the several different kinds of patching which need to be performed. The more simple type of patching is that pointer bits need to be set on data values which are pointers, since that information is not part of the normal 64 bit data word. The object file format maintains a relocation table which indicates to the loader which data needs to be patched in that way.

The second type of patching is the replacement of system entry point magic numbers with real system entry pointers. Because this replacement is done at load time, the user program may not put false entry pointers into its data segment. Once this patching is done the user program may load these entry pointers from its data segment and use them to access system functions.

5.4.1 The Simulated Loader

The present M-Machine loader is part of the simulator: that is, the runtime system sets up a special trap and the simulator accomplishes the load in between simulated cycles. While this is less than ideal, the amount of time required to actually load a large program into memory, which includes translation from the 32 bit word used by SunOS to the 64 bit M-Machine word (including a change of byte ordering), is far too great to allow simulation in a reasonable amount of time. A prototype loader has

```
/*make a pointer to VT3 Cluster 0*/
instr ialu imm ##(VT3C0Ptr >> 48), intarg0;
instr ialu shoru ##((VT3C0Ptr << 16) >> 48), intarg0;
instr ialu shoru ##((VT3C0Ptr << 32) >> 48), intarg0;
instr ialu shoru ##((VT3C0Ptr << 48) >> 48), intarg0;
instr ialu setptr intarg0, intarg0;

/*first, set up the context pointer:*/
CONSTRUCT_LONG_PTR(_TCP3, itemp0, DStart)
instr memu st i2, itemp0;

/*okay, you're in cluster 0.  first, the preempt:*/
instr ialu imm ##CFG_LT_PRE_COUNTER, itemp0;
instr ialu lea intarg0, itemp0, intarg1;
instr ialu imm #0x40, itemp0
memu st i0, ##(CFG_LT_PRE_THRESHOLD - CFG_LT_PRE_COUNTER), intarg1;
instr memu st itemp0, intarg1;

/*now the initial IP*/
instr ialu imm ##CFG_LT_HFORK_IP, itemp0;
instr ialu lea intarg0, itemp0, intarg1;
LOAD_FAR_LABEL(EVENT_IP, itemp0, DStart)
instr memu st itemp0, intarg1;

/*now set up the stack*/
instr ialu imm #0x2000, intarg0;
FCALL(mem_alloc)

instr ialu imm #(0x2000 - 0x08), intarg1; -- set to last word
instr ialu lea intarg0, intarg1, intarg0; -- in the segment

/*make a pointer to CSW Cluster 0 T3 data transfer*/
instr ialu imm ##(CswT3C0Ptr >> 48), itemp0;
instr ialu shoru ##((CswT3C0Ptr << 16) >> 48), itemp0;
instr ialu shoru ##((CswT3C0Ptr << 32) >> 48), itemp0;
instr ialu shoru ##((CswT3C0Ptr << 48) >> 48), itemp0;
instr ialu setptr itemp0, itemp0;

/*store stack pointer in i2*/
instr ialu imm #(CFG_CSW_PER_REG * 2), itemp1;
instr ialu lea itemp0, itemp1, intarg1;
instr ialu imm #(CFG_CSW_PER_REG * 5), itemp1
memu st intarg0, intarg1;
/*store DStart in i5 and i11*/
instr ialu lea itemp0, itemp1, intarg1;
instr ialu imm #(CFG_CSW_PER_REG * 11), itemp1
memu st DStart, intarg1;
instr ialu lea itemp0, itemp1, intarg1;
instr memu st DStart, intarg1;
```

Figure 5-1: Installing a handler

been constructed to look at what the non-simulated loader would consist of, but it is too slow for use until the hardware is available, taking tens of thousands of simulated cycles to load the simplest programs.

5.4.2 Future Loader Work

In order to complete the boot sequence, the final non-simulated loader will have to be constructed, including a definition of the interface between the runtime system loader and the host computer, accessed through the I/O system.

Chapter 6

Exceptions

Exceptions are incidents in program execution which are genuine errors in the program. Unlike events, which are recoverable faults such as LTLB misses, exceptions are load permission violations, illegal instructions, and similar fatal incidents. Exceptions occur on a per-cluster basis in the M-Machine. In some cases, the exception may be deferred, instead producing a special pointer known as an errval. Errvals propagate through instructions until they cannot be propagated further, at which point they cause an exception. An add instruction which receives an errval as an argument, for instance, passes the errval on as its result. A compare instruction, however, cannot place the errval into its one bit result register, and so must signal an exception. The runtime system is not concerned with errvals, but it is required to handle exceptions when they arise. This chapter discusses the general types of exceptions, the design of the present exception handler, and future work on that handler.

6.1 Types of Exceptions

Exceptions fall into several broad categories. There are permission exceptions, caused when a user mode thread attempts to execute an instruction restricted to system modes only. Another type of exception is attempting to use an errval in an instruction which has no way of propagating it, such as a comparison. Memory permission violations and send permission violations in which pointers of the wrong type (or things which are not pointers at all) are used round out the exceptions, leaving only a few specific exceptions. For a complete listing of all exceptions and their encodings, see [4]

6.2 Hardware Restrictions

A copy of the exception handler runs in each of the three H-threads in the exception V-thread, which is one of the three system V-threads. At the time when this exception handler was deisgned, the resources of the exception handler were greatly reduced. Since then, more registers have become available.

In the older version, when an exception occurs, four integer registers and one floating point register (in clusters which have a floating point unit) are filled with exception information. One general purpose integer register remains. Because of this small set of resources, much of the management of exceptions occurs outside the exception handler itself. Actual exception handler code is small and completely hand-coded to deal with the extraordinary restraints imposed.

6.3 Design of the Exception System

The actual exception system on the M-Machine comes in two parts. The first is the exception handler, which runs in V-thread four and receives exception information when it occurs. The second is the exception minder, which runs in another system thread slot. The exception handler's job has to be simple because of its rather pathetic means: it merely has to wake up the exception minder. The exception minder, with normal hardware resources, has the task of determining what exception has occured and doing something about it.

Communication between the two uses the configspace mechanism discussed in chapter 4. The exception minder sleeps on a register which it has emptied. When the exception handler wishes for the minder to awaken, it writes into that register through configuration space. The exception minder then may read out exception information

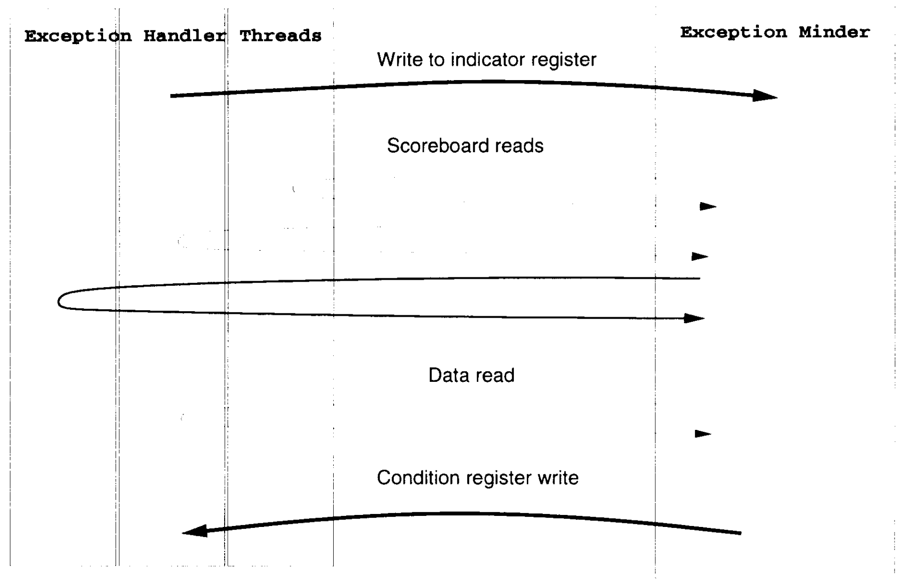


Figure 6-1: Exception Handler and Minder Interaction

through configuration space as well, avoiding possible problems which could occur because of the three distinct copies of the exception handler.

6.3.1 Exception Handler

The exception handler empties the S register and sleeps on it. When an exception occurs, the hardware fills the S register and some subset of the integer, memory, and floating point vector registers depending on which of those three types of exceptions occurred. User threads are also prevented from issuing, since only one exception may be dealt with at a time on any particular cluster. System threads are assumed not to cause exceptions, so they may issue.

The exception handler uses its spare register to construct a pointer to the exception minder's indicator register, and then writes to that address. It then empties a condition register and waits on that. When it is reawakened by a write from the exception minder, it restarts user threads on its cluster by constructing a configspace pointer in its spare register, empties its S register and vector registers, and goes back to sleep.

6.3.2 Exception Minder

Upon awakening, the exception minder first reempties its indicator register. It then uses configspace to read out the integer register scoreboards for the three exception H-threads. With this information, it can determine which of the three exception H-threads is active. By polling, it avoids the potential problem of being awakened by two or even three different exception threads. If it were to rely on the threads to provide information, it would run the risk of exceptions on different clusters overwriting each other.

Once it has determined that an exception has occured in a given cluster, it again uses configspace to read out the vector registers of the exceptions which have occured. The present exception minder merely reports what exceptions have occured, but later versions may easily perform operations such as terminating a thread or writing a substitute value into a faulting register.

Once the exception has been dealt with, the exception minder writes that H-thread's condition register so that it may continue execution. The exception minder spins waiting for the condition register to be empty before writing it, just in case it might have raced far ahead of the exception handler.

6.3.3 Concurrency Issues

The mechanisms for communication between the exception handler threads and the exception minder have been carefully chosen to tolerate their concurrent execution. Because three separate threads may attempt to awaken the exception minder, it polls to determine who is active rather than having that information written into its registers. In addition, by emptying its indicator register early, it guarantees that the worst failure it can experience is to be awakened an additional unnecessary time. Finally, the exception minder makes sure not to write into the condition register of a handler H-thread before it is emptied. It may spin waiting on this condition since the exception handler thread is guaranteed not to be waiting on anything to perform the empty.

6.4 Catastrophic Exceptions

A catastrophic exception is one which occurs in a system handler slot, and can only occur in the case of a gross error in the runtime system software. In the case of such an exception, the cluster experiencing it is halted, and a special hardware flag is set. This flag may be wired to an output pin of the chip to indicate such a failure. We do not expect to recover from a catastrophic exception, and the aftermath of such an event is left to the diagnostic port and the host computer.

6.5 Future Exception Handler Work

The present exception system merely reports what exception has occured and naively restarts the user thread which experienced it. While this is what is desired for hardware testing, in which we often wish to run threads which cause many exceptions, later M-Machine work will require a more sensible process for dealing with exceptions.

In general, a user thread which experiences an exception can be assumed to be likely to produce the wrong result in the future, if it is able to terminate at all. The exception system has no way of patching the typical user failure, so the best thing for the system to do is to terminate the thread and provide as thorough a diagnosis as possible on what went wrong. Threads which are parents or children of the doomed thread may also have to be terminated — they will certainly have to be notified. While this may seem like a vendetta on the part of the runtime system, such threads are likely to need to communicate with the dead thread at some future time, and so must be removed themselves.

Given the present modular nature of the exception system, it could be possible to allow users to insert their own exception routines, which could be executed by the exception minder, forked into a user thread slot, or perhaps spawned off into a spare system H-thread. This would entail some level of trust of the user code, but is potentially useful for research on the M-Machine or in using the M-Machine as a platform for software research.

Chapter 7

Cache Coherence and Reclamation

The M-Machine runtime system provides access to data on remote nodes by caching that data locally. A special set of local page frames is provided for this caching. While the size of this pool of backing page frames can be varied at compile-time, it will certainly be limited, and at some point the runtime system will need to reclaim page frames in order to cache new remote data.

The cache-coherence system was described in [5], but the reclamation procedure was not yet designed at that point.

7.1 Overview of Cache-Coherence System

The cache-coherence system is centered around two large data structures, with messages flowing between nodes to keep them synchronized. These data structures are the coherence directory, which resides on the home node of a piece of data and keeps track of which other nodes have copies of that data, and the software event table, which resides on the remote node and records the status of local cached blocks of remote data, including outstanding block status miss events.

The first attempt to access a remote page of data causes an LTLB miss, and the LTLB miss handler makes a mapping for that remote page to a local physical page frame with all blocks in an invalid state. Upon retrying the faulting reference, and on any future references to new blocks in that page, a block status miss is generated.

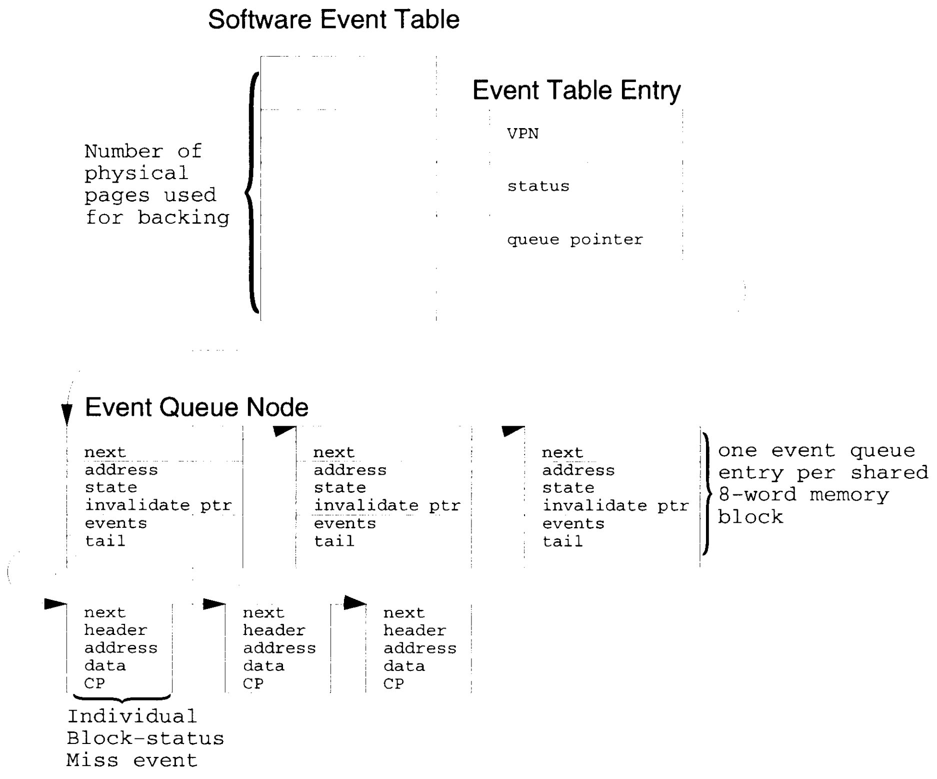


Figure 7-1: Software Event Table

This block status miss is recorded in the software event table.

The block status miss causes the event handler to send a request message to the appropriate remote node, requesting a copy of the data in either read only or read/write permission. The remote node makes note of this in its coherence directory and replies with the data. Upon receipt, the message handler installs the data and removes the outstanding BSM from the software event table.

Clearly, there are many variations on the basic theme, such as the remote node needing to reclaim its data, or the return of a piece of data satisfying several outstanding block status misses, but the simple case gives us enough of an overview to talk about reclamation.

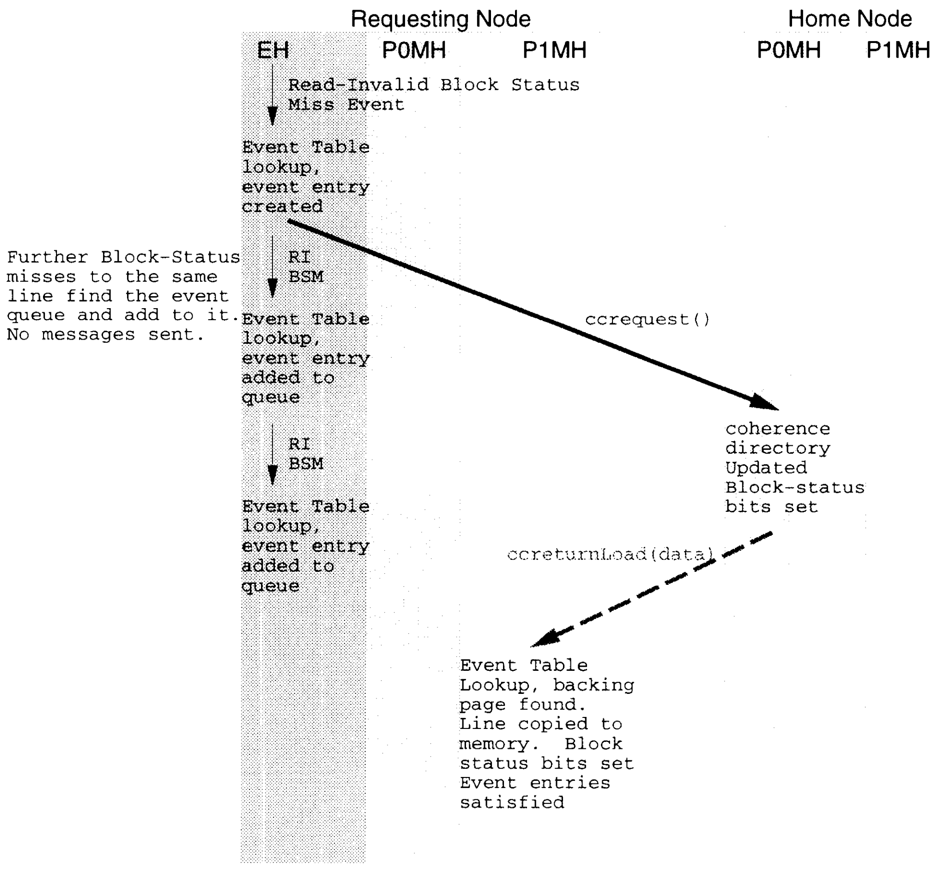

Requesting Node
Home Node
EH
P0MH
P1MH
P0MH
P1MH
Read-Invalid Block Status Miss Event
Event Table lookup, event entry created
Further Block-Status misses to the same line find the event queue and add to it. No messages sent.
RI BSM
Event Table lookup, event entry added to queue
ccrequest()
RI BSM
Event Table lookup, event entry added to queue
coherence directory Updated Block-status bits set
ccreturnLoad(data)
Event Table Lookup, backing page found. Line copied to memory. Block status bits set Event entries satisfied

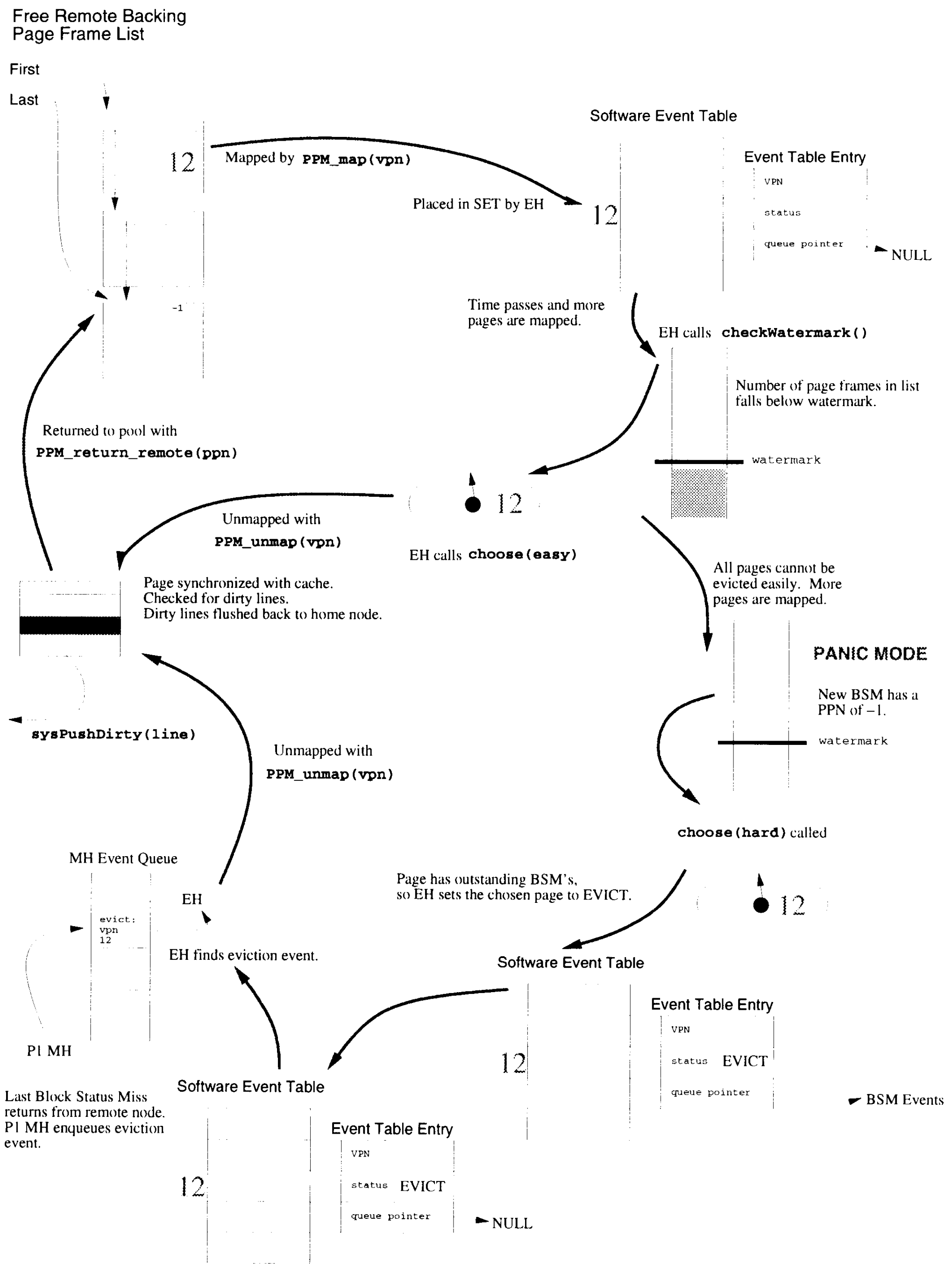
Figure 7-2: Cache-Coherence: the Simple Case

7.2 Concerns

Page reclamation routines ideally would work entirely in the background, reclaiming unused local backing for remote data pages before the runtime system runs out. However, there must also be procedures to deal with the case where the runtime system runs out of backing pages completely. And of course the page reclamation routines must preserve the event handler's guarantee to absorb and process all events.

7.3 The Life of a Page Frame

A remote backing page frame starts its life when it is separated out of the main list of page frames upon system initialization. The software event table is also initialized



Free Remote Backing Page Frame List
First
Last
12
-1
Mapped by PPM_map(vpn)
Placed in SET by EH
Software Event Table
12
Event Table Entry
VPN
status
queue pointer
NULL
Time passes and more pages are mapped.
EH calls checkWatermark()
Number of page frames in list falls below watermark.
watermark
Returned to pool with PPM_return_remote(ppn)
12
EH calls choose(easy)
Unmapped with PPM_unmap(vpn)
Page synchronized with cache. Checked for dirty lines. Dirty lines flushed back to home node.
sysPushDirty(line)
All pages cannot be evicted easily. More pages are mapped.
PANIC MODE
New BSM has a PPN of -1.
watermark
choose(hard) called
Unmapped with PPM_unmap(vpn)
MH Event Queue
evict: vpn 12
P1 MH
EH
EH finds eviction event.
Page has outstanding BSM's, so EH sets the chosen page to EVICT.
12
Software Event Table
12
Event Table Entry
VPN
status EVICT
queue pointer
BSM Events
Last Block Status Miss returns from remote node. P1 MH enqueues eviction event.
Software Event Table
12
Event Table Entry
VPN
status EVICT
queue pointer
NULL

Figure 7-3: The Life of a Page Frame

so that its start and end are the same as the start and end of the list of backing page frames, which are guaranteed to be contiguous by the physical page manager.

Once a new remote virtual address is referenced, the physical page manager maps that virtual page to this backing page frame. The virtual page number is written into the software event table entry corresponding to this page frame.

Eventually many page frames have been mapped to virtual pages, and the event handler detects that the number of remaining page frames is below some water-mark. It then tries to evict pages which are easily evicted. The event handler will invoke a chooser function in order to pick a page to be evicted.

If this page has no BSMs pending to it, its software queue pointer is NULL, and it is easy to evict. If chosen, it will be unmapped by the event handler. Then, it will be flushed from the cache, and each line will be shipped back to the home node in a message if dirty. Finally, this page frame will be returned to the pool of remote backing page frames. The frame does not need to be cleaned because nothing can be read from it without first causing a block status miss event and having new data be installed over the old values.

However, perhaps this page, and all others in the table, have BSMs still waiting to be resolved. In that case, the event handler will not perform any evictions at this time.

If the event handler is unable to evict remote pages from page frames for long enough, the physical page manager will eventually run out of page frames to give out. In that case, a block status miss event will come to the event handler with its physical page number set to 0. The event handler will note that this BSM is unmapped, and go into panic mode. In preparation for panic mode, all user threads must be stopped, so that they do not continue to generate events to pages which have not been mapped or are being evicted.

While panicked, the event handler will attempt to evict pages until it is back below a water-mark, rather than merely evicting one page, which would make the problem likely to immediately reoccur. It will invoke the chooser function to pick a page, first looking for easy pages, which it will evict immediately, and, if those are insufficient, taking pages which have outstanding BSMs as well.

Easy pages will be evicted as they were before. Pages with outstanding BSMs are marked for eviction, and will be evicted once their last BSM comes in. Further BSMs to that page will be recirculated by the event handler.

To come out of panic mode, the event handler must evict enough pages to be below its watermark. In addition, all recirculated events must be dealt with, which may cause some additional mappings and prolong the panic mode.

7.4 Modes

Page reclamation may function in one of three modes. In the first, it is simply not doing anything. In the more interesting modes, it is either evicting pages that are easily evicted, or panicking because there are no remaining backing pages.

7.4.1 Normal Operation

During normal operation, the event handler checks the water-mark whenever a block status miss occurs. If that check returns zero, then there are more than enough backing page frames. If it returns some positive number the event handler will attempt to evict that many pages. It stops evicting pages when it has either evicted the proper number or when it cannot find any more mapped page frames without block status miss events.

7.4.2 Panic Operation

Upon entering panic mode, all user threads are stopped to prevent events from piling up while the event handler is busy. Further events to pages which are set to be evicted, as well as the block status miss event which caused the panic, are placed in the event handler's event recirculation queue.

The event handler checks the watermark in order to determine how many pages it should evict. In this case, it may choose pages which have outstanding BSMs, since any pages which can be easily evicted will have been already at this point. Those pages it sets to evict, and waits for the block status misses to return.

During the panic mode, the event handler deals with other events as normally as possible. User threads will not be generating additional events, but the message handlers may be, so their software queue will need to be attended to to prevent overflow.

When block status misses return, they are dealt with by the P1 message handler, which removes them from the software event table. When it removes the last one from a page which is set to evict, it will enqueue an eviction event for that page frame with the event handler.

Once all (or at least enough) of the evictions that the event handler wished to accomplish have been completed, the event handler deals with its queue of recirculated events, and then resumes normal operation.

7.5 Future Development

There are still a few aspects of the reclamation procedure, and the cache-coherence system as a whole, which might be improved, especially after encountering more vigorous use on the actual M-Machine hardware.

The cache-coherence system presently allocates a large number of software records for block status misses, but does not have a graceful failure mode in case of running out of those records. The system should in the future be provided with a way to deal with the unlikely possibility of an incredibly large number of block status misses overwhelming its storage.

The procedure which the reclamation system uses to choose what pages will be evicted from a local node is an extremely simple system which sweeps the page table in sequence looking for a potential victim. In the future, a clock algorithm might prove more efficient. One potential clock algorithm will sweep forward marking pages as unused until it finds one which is already unused (the one it will choose, unless it has BSMs and the chooser is only looking for easy pages). When it marks pages as unused, it also invalidates them in the hardware LTLB, so that when they come back to software, they will be marked again as used. This sweep will cause extra LTLB misses, but only rarely, and only when evictions are desired.

Finally, the cache-coherence system was implemented when the M-Machine did not have any messaging primitives with a guarantee of in-order delivery. Since that time, this capability has been added to the hardware, and the cache-coherence system could be somewhat simplified by using these message types.

Chapter 8

Location and Direction

This chapter discusses the status of the work on the runtime system: what contributions the runtime system makes to OS design, what is complete, what remains to be done, and what the future holds for this project. There is also a short discussion on the effect that the somewhat unsteady design of the hardware portion of the M-Machine has had on the runtime system project.

8.1 The M-Machine Runtime System and OS Design

The M-Machine runtime system is a unique design for an operating system. This is mostly due to the demands of the M-Machine hardware: its support for multiple threads, its easily accessible hardware state, and its use of pointer capabilities in a single address space. Because of these hardware decisions, the runtime system relies not on a monolithic kernel or even microkernel, but rather on a collection of system level handlers, and even system functions directly invoked in a user thread. The M-Machine runtime system is truely decentralized: there is no one authority to which all other system threads defer on all matters.

Because of this design, the runtime system is able to run concurrently with itself and with user threads, enabling the M-Machine hardware to be constantly avaiable for multiple thread execution. The single address space, guarded pointers, and hardware thread slots enable the operating system to run simultaneously with other threads, and the design of the operating system keeps these hardware features from being wasted.

8.2 Present Location

The handlers described in this document, as well as those described in [5], are completely coded save for a few small portions, and form the bulk of the runtime system. The runtime system is being used for testing, not only of user level programs and the C compiler, but also in order to verify the final hardware design.

8.2.1 Left to be Coded

Although code has been written for almost all of the pieces described in this thesis, there are still a few pieces which will need to be written prior to the arrival of the hardware. These include sections which would be too slow to run fully simulated, such as the loader, and sections which are waiting for final specification of portions of the hardware, such as the middle portion of the boot sequence, during which the final boot code is pulled onto the M-Machine through the I/O port. The interface between the M-Machine and its host computer has not yet been designed, so these particular elements, which involve interaction with the host, will have to wait.

There are also portions of the runtime system which remain in the simulator, mostly functions which require interaction with the outside such as print and file functions. These also await a design for host computer interaction. The home node coherence directory of the cache-coherence system is also still part of the simulator: it was written that way for speed during testing, and while the software event table was moved into simulated code, this data structure has not yet made the transition. Little additional coding should be necessary to move it, since both the simulator and the simulated code are written in C.

8.3 Working on a Moving Hardware Target

The design of the M-Machine runtime system has always had to contend with the fact that the hardware design of the M-Machine has been changing with time. The original design, for instance, had four clusters and six hardware thread slots before it became clear that these resources would not fit on the MAP chip. The configuration space encodings have changed several times over the course of the project as they have been better specified, and events and exceptions have suffered similar alterations.

Even as this document is being completed, the exception V-thread has received a standard set of registers, opening the way for the next exception handler to be fully contained in the exception V-thread and somewhat less complicated than the present design.

But while the changing hardware design has made the runtime system programmers' lives a bit more exciting than is absolutely necessary, proceeding concurrently with the two designs has provided valuable feedback to the hardware team, letting them know more immediately the cost of various hardware decisions. And the runtime programmers? Well, they get another lesson on the value of modularity and good coding style.

8.4 Future Work

In addition to alterations necessary to suit the hardware, there are a few major directions that runtime development will follow in the future. These include the building of libraries for the execution of a greater number of standard programs, and the tightening of existing runtime code.

8.4.1 I/O

Once the interface to the host computer is fully defined, the runtime system will need a library of I/O functions in order to access programs and data stored externally, and to return data to the host storage system. This may include first routing the data internally in the M-Machine to a designated I/O node, or all nodes may be connected to the host machine. The final shape of the loader will also depend on the decisions made about the shape of the I/O system in hardware and software.

8.4.2 UNIX Library

Using the functions already provided by the the runtime system, a UNIX-like layer can be built up, providing a library of standard system calls so that UNIX programs may be easily executed on the M-Machine. In many cases this will involved limiting the more general and powerful runtime system abilities to conform with the older UNIX design. This layer will greatly add to the functionality of the M-Machine as a research multicomputer.

8.4.3 Optimization

The data structures and organization of the runtime system was in many cases chosen for ease of maintainance and design. Future developers may want to replace some portions of the runtime system with more efficient alternatives. The already mentioned change to the cache-coherence system to use in-order messages is one example. Others include the replacement of list data structures in the runtime system with faster structures such as hash tables, and the reimplementation of key sections of code in more efficient C or perhaps in hand-coded assembly.

Bibliography

[1] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. M-Machine architecture v1.0. Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1994.

[2] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee. The M-Machine Multicomputer. A. I. Memo 1532, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, March 1995.

[3] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The MAP instruction set reference manual v1.3. Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February 1995.

[4] Steven W. Keckler. The Architecture of the MAP Exception Logic. Concurrent VLSI Architecture Memo 108, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September 1996.

[5] Yevgeny Gurevich. The M-Machine Operating System. Master's Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, September 1995.

[6] Keith Klayman. Software User's Guide to the Global Configuration Space Controller. Concurrent VLSI Architecture Memo, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1996.