CABLES: THREAD AND MEMORY EXTENSIONS TO SUPPORT A SINGLE SHARED VIRTUAL MEMORY CLUSTER IMAGE

by

Peter Andrew Jamieson

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Cables: Thread and Memory Extensions to Support a Single Shared Virtual Memory Cluster Image

Peter Andrew Jamieson

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2002

This dissertation addresses needed functionality in Shared Virtual Memory (SVM) clusters. Existing APIs for shared memory clusters are not as flexible as APIs supported by todays symmetric multiprocessors and distributed shared memory systems. This makes SVM clusters difficult to use and inhibits application portability across different architectures. This dissertation addresses this problem by providing the standard pthreads interface on top of SVM clusters. Our system, *CableS*, solves issues related to dynamic memory management, thread management, and synchronization. Existing applications that use pthreads can be run on *CableS* with minimal or no modifications. Furthermore, *CableS* introduces minimal overhead in data movement and synchronization operations.

Acknowledgements

Primarily, I would like to thank Angelos Bilas, my supervisor, for two interesting and challenging years. Over time we have had many interesting discussions, and I have advanced from ”Professor Bilas” to ”Angelos”.

Coming in a close second are my parents for their life long tolerance and support. I am here today because of their love, and effort. I probably should also thank my sisters, Patricia and Jen, for being good friends. Finally, my gran, Irene Caron, has always been supportive and loving. I could never have completed this work without such a wonderful family.

Then comes my significant other, Alexis. Not only has she discussed and aided me on technical aspects of my work, but she has provided me with unrestricted friendship and love during this work.

I would also like to thank the many friends I have met. The guys and girls in the lab are invaluable resources and fun to goof off with. Torontula characters, and my many frisbee friends are wonderful, and provide me with that much needed physical outlet when work is stressing me out. To the others throughout the world, you keep me going, because I always know that we will spend happy times together. Matt, Ryan, Paul, Kevin, Rob, Karl, Ajay, Catherine, Alex, Steve, Daphne, Kelly, Ali, Dan, Stella, Nicole, Jeremy, Jamie, Cornell, Mike, Neil, Chuck, Nick, Derek, Patricia, Jen, and Joe thanks for being there.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Recently, there has been progress in building high-performance clusters out of high–end workstations and low-latency, high-bandwidth system area networks (SANs). SANs, used as interconnection networks in these clusters provide memory–to–memory latencies of under 10μs and bandwidth in the order of hundreds of Megabytes, limited mainly by the PCI bus. We expect that in the near future system bandwidth will increase with the wide deployment of next generation PCI buses and network links, although latency will probably remain at current levels.

The shared memory abstraction is being used in an increasing number of application areas. Developers have been writing new applications for the shared address space abstraction and legacy applications are being ported to this abstraction as well. Also, most vendors design both small–scale symmetric multiprocessors (SMPs) and large–scale, hardware cache-coherent distributed shared memory (DSM) systems, thus targeting both scientific and commercial applications.

Given these trends, shared memory clusters are an attractive approach to providing affordable and scalable compute cycles and I/O. For this reason, there has recently been a lot of work on designing efficient shared virtual memory protocols for such clusters [35, 22, 40, 19]. Providing a shared memory programming abstraction on clusters has made it easier to run applications that have been written for more traditional, tightly–coupled multiprocessors (both shared bus and distributed shared memory machines).

Past work [35, 22, 40] has shown that, for a wide range of applications, the performance of shared virtual memory clusters is competitive to more traditional, tightly–coupled multiprocessors. For instance, the authors in [22] find that a 64–processor cluster offers, for most SPLASH-2 applications (after a number of optimizations), at least 50% of the performance offered by a 64–processor SGI Origin2000.

SVM systems as clusters interconnected with low-latency, high-bandwidth system area networks (SANs) can potentially fill in the portability gap between a wide range of architectures supporting the shared memory model. This portability is only possible if each architecture uses the same API. Figure 1.1 depicts a range of architectures that applications can use as their requirements change over time.

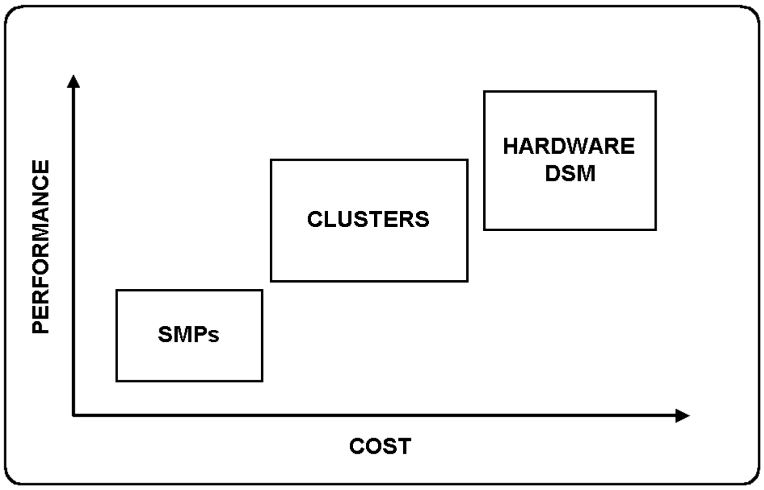


Figure 1.1: The cost performance spectrum in which cluster systems could potentially exist

However, despite the many advantages of clusters over more traditional, tightly–coupled scalable servers, including their competitive performance, their use is not widespread, especially in areas of commercial applications. One of the reasons is that despite the progress on the performance side, it is still a very challenging task to port existing applications or to write new ones for the shared memory programming application programming interfaces (APIs) provided by clusters. Although these APIs provide sufficient primitives to write parallel programs for the shared memory abstraction, they also impose several restrictions.

The main limitations are:

- Processes cannot always be created and destroyed on the fly, during application execution. This is especially true on clusters which use modern SANs. These systems require some form of remote memory mapping. Also, in these systems all nodes/processes need to be present at initialization, for the initial mappings to be performed.

- In most shared memory clusters the synchronization primitives supported are *lock/unlock* and *barrier* primitives. However, more modern APIs support conditional waits.

- Since these systems do not support a shared address space in hardware, in most cases the memory system is often very limited. Programmers may allocate shared memory only during program initialization. Also, placement of data is limited due to restrictions imposed by the SANs, on both the number of regions, as well as the total amount of memory space that can be mapped.

Although these limitations are not very important for large classes of well structured scientific applications, they pose important obstacles for using clusters in areas of applications that exhibit a more dynamic behavior. Therefore, in many areas, traditional shared memory multiprocessors are used because of their ability to support legacy applications with no or very few changes. Also, development of new applications usually occurs on these architectures, since the APIs provided impose fewer restrictions on the programmer. In essence, current clusters that support shared memory provide a very limited single system image to the programmer with respect to processes, memory management and synchronization.

1.2 Objectives

The primary goal of this work, therefore, is to provide a single system image with respect to thread and memory management. This will provide SVM clusters a similar application programmer interface (API) to that of SMPs and tightly–coupled distributed shared memory (DSM) systems. This will allow existing applications to run on clusters with minimal or no changes and will facilitate the task of writing new applications. Furthermore, we would like to minimize introducing additional overhead in applications that have already been tuned for SVM clusters.

In order to overcome these existing limitations and to provide a more complete and single cluster image to the programmer, we design and implement a POSIX *pthreads* interface on top of a cluster. We provide most functions specified by the *pthreads* interface [2] and all functions commonly used by *pthreads* programs. Therefore, existing *pthreads* programs run on our system with minor modifications. These programs can dynamically create and destroy threads on our system, allocate global shared memory throughout execution and use all synchronization primitives specified by the *pthreads* API. More specifically, *CableS* provides:

- **Support for dynamic node and thread management**: *CableS* allows the application to dynamically create and destroy threads at any point during execution.
- **Support for synchronization primitives**: *CableS* supports the conditional wait primitives specified by the *pthreads* interface, as well as locks and barriers.
- **Support for dynamic memory management**: *CableS* addresses a number of issues with respect to memory management. (a) *CableS* provides all necessary mechanisms to support different memory placement policies. Currently, *CableS* supports first touch placement, but can be extended to support other policies as well. (b) *CableS* provides the ability to allocate global, shared memory dynamically at any time during program execution. *CableS* maximizes the amount of the virtual address space that can be used for shared data by using a novel memory registration mechanism. (c) *CableS* incorporates static global variables that were previously not part of the global shared address space.

We should note however, that although *CableS* provides a single system image with respect to thread management, memory management, and synchronization support, it does not include file system and networking support across cluster nodes. Extending *CableS* with file system and networking support is one of the focuses of future work.

1.3 Background

1.3.1 Pthreads

Previous work has proposed a number of APIs for thread libraries. POSIX threads (*pthreads*) [2], Java threads [26] and Microsoft Windows threads are some examples of current APIs. In general, these APIs provide functionality to create, destroy, and administrate threads and synchronization constructs. Specifically, the *pthreads* API has gained popularity on UNIX based systems and is supported by all major operating systems, such as IRIX, Solaris, Linux, AIX, and WindowsNT [1].

The *pthreads* API described within POSIX threads IEEE POSIX 1003.1 [2] consists of:

- Thread Management (Section 16)
- Thread Synchronization (Section 11)
- Thread Specific Data (Section 17)
- Thread Cancellation (Section 18)

In each section, the API syntax and system semantics for the related pthread elements are provided.

1.3.2 Process Virtual Address Space Map on WindowsNT

In WindowsNT process code and data are stored within a private virtual memory address space. The process virtual address space can be broken down into five regions:

1. Code Region: The code is a numerical data representation of instructions for the processor to execute. Code is part of the executable image, and this region is allocated by the OS's program manager.

2. Static Initialized Region: Static initialized data is global data that is visible by all threads. This part is allocated by the OS's program manager as part of the executable image.

3. Static Uninitialized Region: Similar to static initialized data is static non-initialized data region. This data is also part of the executable image called the "bss" region.

4. Heap: The heap is a pool of memory that a process uses dynamically when it needs more room to store data. The heap is managed by the programmer via library calls and system calls. The system calls activates the OS's memory manager which manages the process address space and the mapping of virtual to physical memory.

5. Stack: Stack memory is usually allocated when the process/thread is initialized by the OS. The default stack size for threads in WindowsNT is 1 MB.

1.3.3 Shared Virtual Memory (SVM) Clusters

SVM clusters of high-end workstations and PCs interconnected with low-latency, high-bandwidth SANs are currently used as a platform for providing a cost–effective shared memory abstraction to applications. Since SANs do not employ hardware support for a shared address space, SVM clusters provide the shared address space (SAS) abstraction across nodes in software. A number of protocols have been proposed and used in the past to provide the mechanics behind the SAS [22, 35, 40, 19]. In particular, most SVM protocols support some variation of Release Consistency [15].

SVM systems are a subset of software distributed shared memory (SDSM) systems. The distinguishing feature SVM systems have is that they control application reads and writes through the OS's virtual protection mechanisms. Other types of SDSM system use other methods for controlling reads and writes. For example, Shasta [33] instruments the application code so that processing can be performed when memory is accessed.

Most SVM systems process reads and writes in software by trapping on page fault exceptions. Individual pages within the shared memory region can be read or write protected, and SVM systems enter the page fault handler when protection flags are violated. Within the handler, the system can determine what protocol action to take on shared global memory. For example, when a node reads a piece of data that is owned remotely by another node, the instruction activates the page fault handler, because the page is read protected. The handler then remotely fetches the data and unprotects the data. The node can now continue executing as if the page of data was local.

1.3.4 OpenMP

OpenMP [3] is an API currently proposed for systems that support shared memory abstraction either in hardware or software. The OpenMP standard uses compiler directives to indicate sections of code that should be executed by threads. The two basic parallel constructs are *for loops* and *parallel sections*. *For loops* involve each thread executing iterations of loop. *parallel sections* are blocks of code from which each thread executes an exclusive block of code. The OpenMP API provides locks and barriers as the two synchronization primitives. Finally, there are various function calls to perform simple system and thread administration during program execution.

1.3.5 Single System Image Systems

Traditionally, the OS is responsible for providing a single system image on top of a parallel system. This setup allows the system to manage all physical resources. From the programmer's perspective, any thread in a multi-threaded application can access any resource initialized by another thread in the same application.

1.4 Related Work

Our system, *CableS*, deals with thread and memory management at a low-level. This means that *CableS* deals with OS and Communication topics. In this section, we present the relevant related work.

Dwarkads *et al.* in [14] examine how SVM protocols can be extended to reduce paging in cases where nodes have relatively small physical memories. Our focus is on dealing with limitations of SANs that support direct remote memory access and providing dynamic thread and memory management. Stets *et al.* in [36] discuss home page migration issues in Cashmere. They examine protocol level extensions for migrating protocol pages among nodes. Shasta [33] is an instrumentation–based software shared memory system that was able to support challenging applications using an executable instrumentation mechanism. However, instrumentation–based, fine–grain software shared memory has its own limitations (e.g. depends on processor architecture) and the related issues and solutions can be very different from page–based shared virtual memory systems.

Several attempts have also been made to eliminate registration limits at the NIC level. Chen *et al.* in [11] address some of the limitations in the amount of memory that can be registered and pinned on modern SANs. However, they deal only with the send path and do not address the related issues on the receive side. Basu *et al.* in [6] try to reduce the overhead of dynamically managing registered memory on the NIC to avoid hardware and OS limits both on the send and receive sides. However, the issue of how the application working set size affects the required NIC resources and system performance is still not well understood.

In [34] Scherer *et al.* present a TreadMarks based system that deals with node attaching and detaching. They use the garbage collection mechanism of TreadMarks to move data among nodes. However, the communication layer used does not support direct remote memory operations, and this results in different mechanisms and tradeoffs. Our work attempts to synchronize all resources, including memory, with low level resource migration.

In addition to this research, other related work has focused on the following four directions:

1. To improve the performance of SVM on clusters with SANs. There is a large body of work in this category [35, 25, 22, 40]. Our work relies on knowledge gained in this area and builds upon it by extending the functionality provided by today's clusters.

2. To provide OpenMP implementations for clusters. Relatively little work has been done in this

area. Honghui *et al.* in [27] provide an OpenMP implementation based on TreadMarks. They convert OpenMP directives directly into TreadMark system calls.

3. To provide a *pthreads* interface on hardware shared memory multiprocessors, either shared–bus or distributed shared memory. Most hardware shared memory system and OS vendors provide a *pthreads* interface to applications [29]. In many systems, this is the preferred API for multi-threaded applications due to the portability advantages. Frank Mueller has developed a DSM-threads library for a distributed system. His library is very similar to *pthreads* library and provides threading on systems that use message passing for communicating shared data [30].

4. To provide a single system image on top of clusters. Projects in this area focus on providing a distributed OS that can manage all aspects of a cluster in multitasking environments and not as a platform for scalable computation. Aridor *et al.* in [5] provide a Java Virtual Machine on top of clusters. This work focuses on Java applications and uses the extra layer of the JVM to provide a single cluster image. There have been many other Java based systems that provide elements of a single system image including: [39], [28], [32], and [4]. All of these systems use and modify a Java virtual machine so that sharing can be accomplished with an underlying SVM system, or by message passing. Our work is at a lower layer.

1.5 Overview

The rest of the dissertation is organized as follows: Chapter 2 describes a specific system, *GeNIMA*, and the platform *GeNIMA* runs on. Our work uses *GeNIMA* as the starting system that *CableS* expands on. Chapter 3 describes the design of *CableS* threading and memory API. Chapter 4 shows *CableS* results for a variety of applications. We describe and analyze a series of experiments run on *CableS* to verify correctness, performance, and improved flexibility. Finally, Chapter 5 discusses system limitations, future work, and concludes this work.

Chapter 2

Platform

In this chapter we describe the research platform for this work. An SVM cluster consists of a number of layers (Figure 2.1). In this work we consider clusters built with commodity components. These include the computing units and the system area network (SAN). The SAN provides the communication layer to the system so information can be transfered between computing units. The protocol layer runs on top of the communication layer and implements the shared memory protocol. Finally, the highest layer, the application layer, uses the protocol layer API to execute a parallel algorithm. *CableS* is an extension of an existing SVM system. *CableS* provides a more flexible API to applications, but to understand the design of *CableS* we first need to understand the underlying system and its limitations.

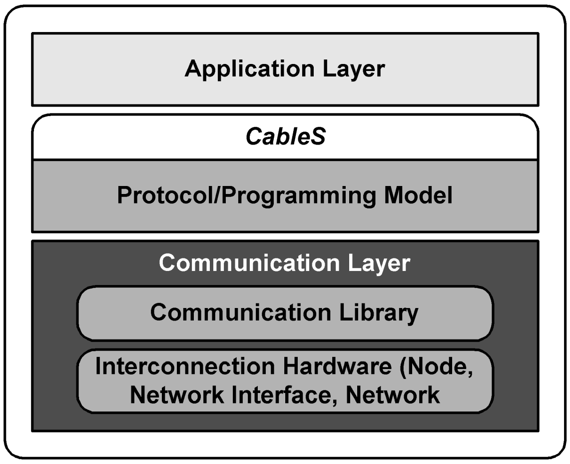


Figure 2.1: System Layers

The cluster we currently use consists of 32 PCs connected together by Myrinet network interface cards (NICs). Each PC consists of a mother board with dual processors and an Ethernet NIC. There are 16 PCs with 400 MHz Pentium dual processors. Each PC is dual boot in either Linux or WindowsNT. Additionally, each PC has at least 512 Mega-bytes (MB) of random access memory (RAM).

The SAN we use is Myrinet [31, 9]. Myrinet is a point to point interconnect. We use a 16–port switch to connect all our nodes in a SAN. Each node is connected to the network with a NIC that is plugged into the PCI bus. The myricom NICs include both memory and a processor are programmable.

2.1 Communication Layer

The communication layer provides specific functionality exploited by SVM clusters. In general, each myrinet NIC is controlled by a control program (firmware) that is loaded directly onto the card. A custom system driver is responsible for providing the necessary extensions for the communication layer.

Our work uses Virtual Memory Mapped Communication (VMMC) [7] as the communication layer. VMMC consists of a custom NIC control program, an OS driver, and a user library. VMMC allows the user to directly access the NIC in a protected manner. Thus, there is no need to trap in the kernel in the common path while exchanging data.

The basic operations provided by VMMC are remote read/write. Nodes can read or write memory through the network without interrupting the remote processor. For the NIC to directly access host memory, it needs to use a physical address. However, users can only specify virtual addresses. Thus, VMMC employs a dynamic mapping scheme [11] that allows the NIC to translate virtual to physical addresses. Users specify which local regions of the virtual address space are accessible to other nodes by using explicit export operations. Similarly, nodes obtain access to remote memory by explicitly importing a region to a local handle. Export and import are one–time operations for each memory region. After the initial import/export operations, the user can directly access remote memory without the kernel or remote processor intervention.

VMMC Operation	Overhead
1-word send (one-way lat)	7.8μs
1-word fetch (round-trip lat)	22μs
4 KByte send (one-way lat)	52μs
4 KByte fetch (round-trip lat)	81μs
Maximum ping-pong bandwidth	125 MBy/s
Maximum fetch bandwidth	125 MBy/s
Notification	18μs
Remote lock acquire	53.8μs
Local lock acquire	12.7μs
Remote lock release	7.4μs

Table 2.1: Basic VMMC costs.
All send and fetch operations are assumed to be synchronous. These costs do not include contention in any part of the system.

In addition to remote reads and writes, VMMC provides a few other useful tools for SVM clusters. When a machine would like to interrupt another machine, a special notification buffer can be used. This buffer is a virtual memory region that must be exported and imported. Any information remotely written to the buffer tells the NIC to interrupt one of the local processors and execute a notification interrupt handler.

Finally, the VMMC library also provides a spawning mechanism, through which a node may remotely start a process on another node. Once the process is started, the original node has no control over the spawned executing process.

Table 2.1 shows the cost of basic VMMC operations on our cluster. Noticeably, VMMC provides a one way, end–to–end latency of around 8μs, which is currently among the best performing systems using a Myrinet interconnect [13]. The lock values are for a NIC implementation of locks and are not currently used in our system.

2.2 *GeNIMA*

GeNIMA is an SVM protocol that provides a SAS abstraction on top of VMMC [23]. *GeNIMA* is based on home–based lazy release consistency [17]. *GeNIMA* [22] is a page-level shared virtual memory protocol that has been developed to take advantage of modern SAN properties.

2.2.1 Application Structure for *GeNIMA*

The API provided to user applications is the parallel M4 macros. These macros are used for both hardware and software distributed shared memory systems. Figure 2.2 shows a simple program written with M4 macros. With this configuration, the programmer only have to deal with high level parallelism problems since the communication and memory sharing has been abstracted into *GeNIMA*'s library. Essentially, the programmer writes the application to deal with correct synchronization of data, and possibly, the application can be optimized so that pages of data are placed on the nodes that will use the data on these pages the most. We call this optimized data placement.

```
/* Header Files to GeNIMA */
#include "genima.h"

/* Declare Global variables/structures as pointers */
var *global_vars;


/* this would be the functions that the program would execute */
work_functions() { }

main () {
        /* Initialise System */
        MAIN_INITENV();

        /* Allocate memory for global variables/structures (must be done here)*/
        global_vars = GLOBAL_ALLOC(size);
        /* must also initialize locks and barriers here */

        /* Create all the threads that will run */
        CREATE(work_function);

        /* call the work procedure */
        work_function();

        /* wait for all threads to end */
        WAIT_FOR_END(number of threads);

        /* end call for the system */
        MAIN_END();
}
```

Figure 2.2: The M4/GeNIMA version of an application.

GeNIMA is built on top of VMMC and takes advantage of the remote memory access capabilities provided by VMMC. For this reason, during the initialization phase, *GeNIMA* needs to establish import/export memory mappings among all the system's nodes. It is also very important that processes on different nodes reserve their virtual address memory regions for the executable image and global memory at exactly the same addresses. In particular, *GeNIMA* performs the following steps during application execution:

1. Start the process on each node within the system.

2. Prepare a region of virtual memory for allocation. This is the shared memory region (Figure 2.2 Stage 1).

3. The parallel application now allocates *all* shared memory that will be used, initializes *all* synchronization constructs and specifies on which node data must be placed (Figure 2.2 Stage 2). If the programmer does not specify data location, then the system will automatically place data using a round robin page placement policy.

4. Once all data is allocated, the system builds a page table for the shared memory and establishes any additional protocol data structures needed for implementing the protocol (Figure 2.2 Stage 3).

5. Each node exports all virtual shared memory regions that it owns and each node imports virtual shared memory regions that other nodes own (Figure 2.2 Stage 3). Ownership of pages has already been determined either by the programmer or the round robin page placement policy.

6. All additional shared management information is imported and exported so that protocol information can be communicated without needing to interrupt the processors during parallel execution (Figure 2.2 Stage 3).

7. Each node creates one thread per processor that will execute the parallel code (Figure 2.2 Stage 3).

8. The system now starts executing the parallel code (Figure 2.2 Stage 4).

9. The system completes execution by waiting for all threads to complete computation (Figure 2.2 Stage 5).

GeNIMA provides an environment in which applications, like scientific applications, that exhibit static initialization patterns can execute. The main features of this platform are the following: (i) The user level communication library, VMMC, allows the system to avoid any interrupts on shared data updates. (ii) Communication of shared data is asynchronous. (iii) All memory and synchronization constructs are initialized at the beginning of execution. (iv) All threads are started together.

Chapter 3

Design

As described previously, SVM systems have a number of limitations. This makes writing new applications and porting applications over to an SVM cluster a challenging task. *CableS* attempts to alleviate many of the threading and memory restrictions by providing a flexible thread and memory management API.

The two main components of *CableS* are:

1. **Threading management extensions:** We adopt and implement the *pthreads* API (POSIX threads IEEE POSIX 1003.1 [2]), which includes thread management, synchronization, and additional POSIX thread related functionality.

2. **Memory system extensions:** We provide extensions to support dynamic memory management.

CableS is an environment in which the underlying data sharing protocol is preserved. However, a dynamic environment with respect to node management, thread management, and memory system management is added above the data-sharing component.

This chapter first introduces the basic mechanisms and structures (3.1) needed to implement distributed management. The design of the thread API (3.2) and memory system extensions (3.3) are then described individually. Finally, we show the updated programmer's template and explain how programs can be ported to *CableS* (3.4).

3.1 Basic Mechanisms and Structures

3.1.1 Administration Requests

The first mechanism is used to handle requests between remote nodes by adding to VMMC's notification primitive. During application execution, *CableS* needs to remotely invoke various operations; for example, when a remote thread must be created. VMMC's notification interrupts the processor when data is written to an exported buffer. In particular, *CableS* creates and exports a buffer so that every other node in the system can remotely invocate actions by sending a packet. The packet consists of a type that describes the desired action and additional information for the arguments of the desired request. This action is called an *administration request*. Administration requests interrupt one of the remote node's processors, and therefore, this is expensive since the remote node must now share one of its processors until the administration request is complete.

With this mechanism in place, a request can be made to the node on which the remote thread should be created. Of course, additional information is needed to describe the request. Figure 3.1(a) shows the basic packet structure for administration requests. Figure 3.1(b) shows a possible remote thread create packet.

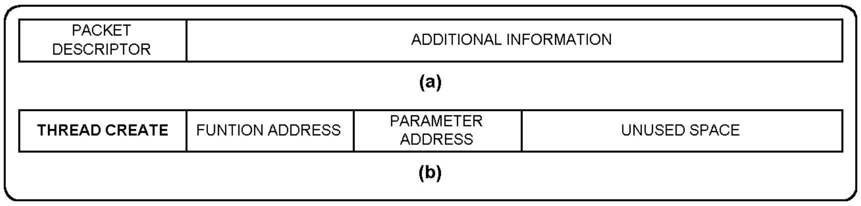


Figure 3.1: Admin Notification Packets
(a) The packet format for administration requests. (b) A sample packet for remote thread create.

3.1.2 Communication Acknowledgments

Some administration requests require actions by both the requesting and servicing nodes. *CableS* uses acknowledgements to indicate when specified actions have been completed. Communication acknowledgments can be done either by an administration request or a flag. Administration requests are preferable in instances where a node must wait a long time for the acknowledgment but can do additional work in the meantime. Flags, which nodes spin on (check flag in a loop until flag is set), are better for quick messages where nodes either are forced to wait for an acknowledgment, or the acknowledgment is expected to be sent quickly. *CableS* uses both types of acknowledgements, but the majority are flags.

3.1.3 Application Control Block

CableS is a system that is dynamic throughout execution. This means that all nodes can initialize resources at any time. Therefore, all nodes need to communicate with the master node to obtain information about resources and global state. The master node is the first node that runs the application. *CableS* uses per application global state, called the application control block (ACB) to store and communicate resource information.

The ACB currently resides on the master node, and all other nodes can retrieve a current version of part or all the global state by remote reads. The ACB is written to in three possible ways by a non-master node: (i) A node uses an administrative request to the master node, and the master node locally updates the requested part of the ACB. The master node is responsible for synchronizing the updates. (ii) A region of the ACB is guaranteed to be exclusively written to by one node. The exclusive node is responsible for synchronizing any updates. (iii) System locks are used. This ensures that any node, after obtaining a lock, is allowed to update part of the ACB. Figure 3.2 shows the three types of ACB writes. These writes are used to guarantee that the data in the ACB is properly updated.

This single point of ACB ownership greatly simplifies the task of maintaining global state. However, this may burden the master node with a high amount of network requests. To reduce remote reads, nodes maintain a local version of the ACB so that remote reads are only performed when the node needs to check if system state has changed. System state only changes for uninitialized resources; changes to initialized resources are broadcast directly by the node that is modifying the resource. Another way to

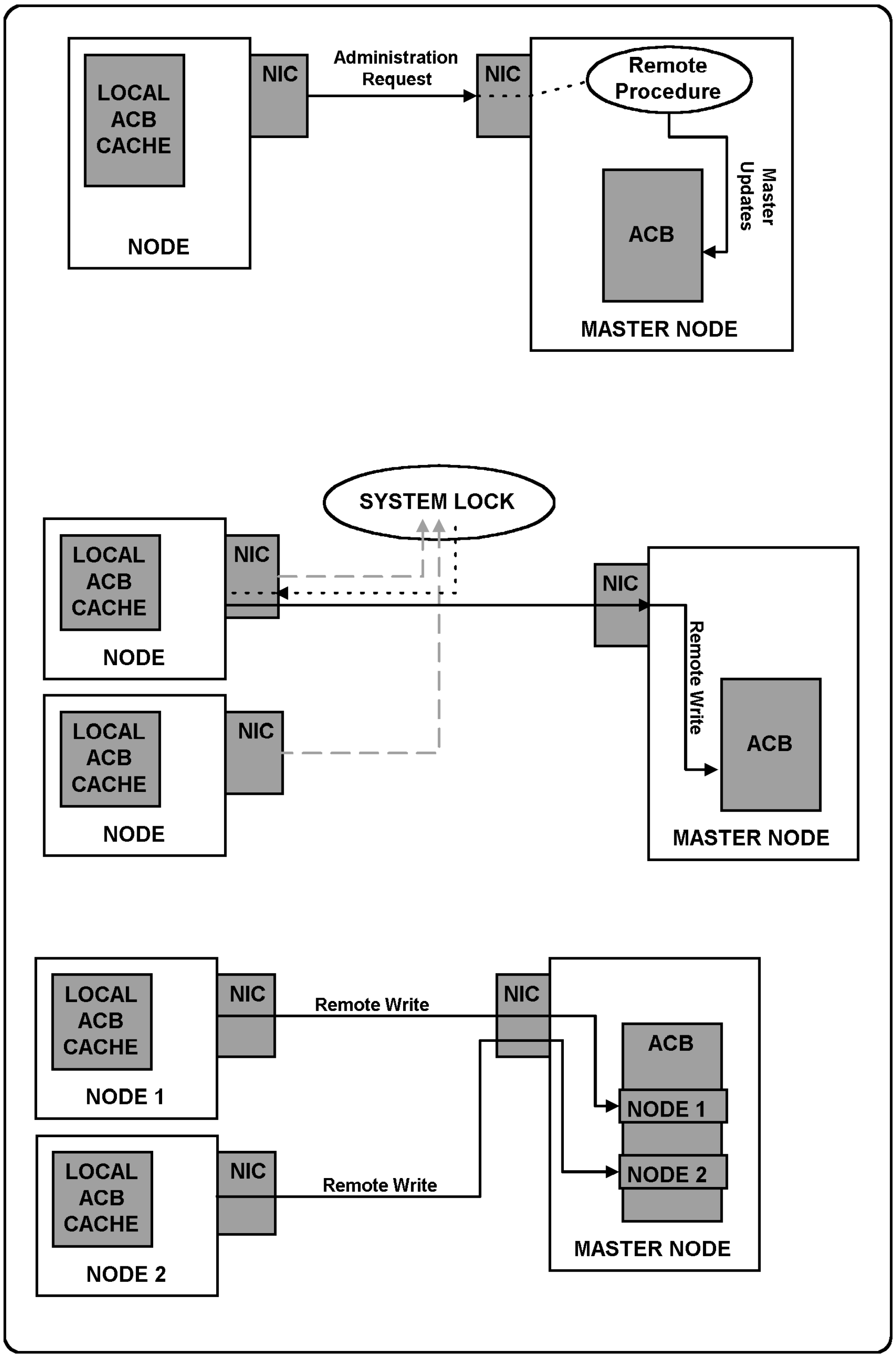


Figure 3.2: Application Control Block (ACB) Access:
Figure shows the three types of writes, used on the Application Control Block (ACB), that guarantee safe writes.

reduce remote reads is by distributing information in the ACB to other nodes and using the master node as a redirection point. In this case, the ACB is examined once to determine where the information is, and all future reads and writes are done on the redirected node and not the master. The central point of ACB ownership was also chosen because in it, only small portions of the ACB needs to be read or written. These operations only occur at initialization and termination library calls.

3.2 Pthreads Implementation

Our *pthreads* API design for *CableS* first describes the implementation of thread management, which includes thread cancellation. We then describe thread synchronization, and finally, we detail the additional thread functionality such as thread cancellation, resource attributes, and system configuration functions.

3.2.1 Thread Management

Thread management includes functionality that controls a thread's operation. These operations consist of the creation and destruction of a thread, as well as ordered waits while other threads complete execution (join synchronization).

3.2.1.1 Thread Creation

Each application starts on a single node. Additional nodes are attached dynamically from a pre–defined pool, as the application needs to create more threads. Figure 3.3 shows the algorithm used to introduce new nodes into the system. Node attachment uses VMMC's process spawning mechanism to start a process and uses administration requests to get all nodes in the systems to connect with the new node. *CableS* currently only allows one node to be attached at a time.

On *CableS* threads can be created in three ways. (i) Local thread creation: This uses the local OS's thread create call. The system then updates the ACB to reflect that a local thread has been created.

```
/* This is done by any node that wants to add a new node to the system */
int node_attach(NODE node_id)
{
        /* make a request from the master node if a new node can be spawned */
        master_node_request(new_node_id);

        /* broadcast to each node that a new node as been added to the system */
        PACKET admin_request_packet = {NEW_NODE, new_node_id);

        while(admin_request(master_node, &admin_request_packet) ==
                NODE_PRESENTLY_INITIALIZING) {}

        /* setup the communication buffers */
        setup_new_connections(new_node_id);
}

/* This is what the master node does to determine if a node attach can be done.
This function is called through an Administration Request */
void master_node_process_request()
{
        local_node_lock();

        if (there is a node that has just been attached and is still initializing)
        {
                acknowledge(NODE_PRESENTLY_INITIALIZING)
        }
        else
        {
                /* find a new node id */
                acknowledge(new_node_id);
        }

        local_node_unlock();
}
```

Figure 3.3: Code representation of node attachment

The ACB needs to store information about the thread's id, the node the thread is executing on and the number of threads now executing on this node. Local OS thread creation is reasonably expensive. Some systems start a pool of threads that are dispatched work to avoid the expensive local creation/destruction calls. *CableS* currently does not support thread pooling, but this functionality could be added. (ii) Remote thread creation: An administration request is sent to an attached node. The request contains information about the thread's arguments so that the remote node can create a thread as specified. It is the responsibility of the remote node to update the ACB to reflect the state of the new thread. Remote thread creation is completed with a flag acknowledgment that includes information on whether the create was successful or not. (iii) The third type of thread creation is a combination of node attachment and remote thread creation. This type of thread creation is used when the system determines that a new node should be attached to the system.

Figure 3.4 shows *CableS* current approach to thread creation. Each new thread executes on its own processor until the system is full, at which point threads are allocated to the node with the least number of threads. In the case when nodes have equal number of threads an arbitrary node, other than the master node, is chosen. The master node is rarely chosen since it already has additional system responsibilities.

3.2.1.2 Thread Destruction

Thread destruction involves terminating a threads execution and cleaning up the ACB. Threads can terminate either by completing their tasks, or by being prematurely killed. *CableS* supports both types of destruction, but does not currently support the pthread_kill function on WindowsNT, which instantly kills a thread via an asynchronous signal.

The two types of thread destruction supported on WindowsNT are:

- *Thread Exit:* When a thread completes execution, or explicitly calls *pthread_exit()* the API specifies that the thread exits. *CableS* needs to update ACB to reflect that this thread no longer exists, and the local OS's thread exit function is called. At this point any threads that are waiting for this

```
int pthread_create(void **function, void *args)
{
        local_node_lock();

        /* LOCAL CREATE */
        if (there is local thread room)
        {
                create_thread_local_OS_call(function);
        }
        /* REMOTE CREATE */
        else
        {
                get_ACB_for_system_thread_info();

                /* CREATE A THREAD ON AN EMPTY PROCESSOR */
                if(there is a node with less threads than processors)
                {
                        PACKET = admin_request_packet = {REMOTE_THREAD, function, args);
                        admin_request(remote_node, &admin_request_packet);
                }
                /* CREATE A NODE + A THREAD */
                else if (there is a reserved node in the system that hasn't been attached)
                {
                        node_attach(uninitialised_remote_node);
                        PACKET = admin_request_packet = {REMOTE_THREAD, function, args);
                        admin_request(uninitialised_remote_node, &admin_request_packet);
                }
                /* CREATE A THREAD */
                else
                {
                        NODE create_node = find_a_node_with_minimum_threads();

                        PACKET = admin_request_packet = {REMOTE_THREAD, function, args);
                        admin_request(create_node, &admin_request_packet);
                }
        }

        local_node_unlock();
}
```

Figure 3.4: Code representation of thread creation

particular thread to exit are resumed.

- *Thread Cancellation:* Thread canceling involves destroying a thread at specific points of execution. When thread A wishes to cancel thread B, the cancel function is called and sets a cancel flag for thread B in the ACB. At specified points of execution, thread B checks its cancel flag and if set, then it explicitly calls *pthread_exit*. Specified cancel points can be called directly by the user or called in library calls outlined in the POSIX *pthreads* standard.

3.2.1.3 Thread Join

Thread joining is an operation where if thread A joins on thread B, then thread A will wait until thread B exits. *CableS* implements a join by using conditional waits (explained in thread synchronization). Basically, thread A waits on a condition variable. Thread B will signal the waiting thread when thread B calls *pthread_exit()*.

3.2.2 Thread Synchronization

Currently, in most shared memory clusters the synchronization primitives supported are *lock/unlock* and *barrier*. However, the *pthreads* API supports *mutexes* and *conditional waits*. We describe *CableS* current support for all of these constructs.

3.2.2.1 Mutexes

Mutexes are *pthreads* name for locks, and on *CableS* mutexes are implemented as SVM locks. *CableS* wraps the original lock primitives of the underlying SVM system. However, the POSIX *mutex* specification states that initialization error checks must be done. *CableS* verifies a *mutexes* existence through the ACB.

There is also *pthread_trymutex()* call that attempts to lock, but instead of blocking until the lock is attained, simply continues execution and indicates whether the lock was attained or not. *CableS* implements *pthread_trymutex()* with a flag check in the ACB.

The only additional problem *CableS* has in implementing *mutexes* is that in a dynamic system, not all nodes are known up front, and therefore, locks that belong to non–active nodes cannot be used. Static SVM systems usually distribute the locks in a round robin fashion to distribute the work load. *CableS* keeps a pool of available locks that can be attained from active nodes. When a node is attached, the nodes locks are added to the pool. This solves the problem, and locks within the pool are still round robin allocated to spread the work.

Finally, *CableS* has the ability to migrate locks and uses this functionality to move locks when a node leaves the system. Migration of a lock resource is achieved by allocating a new lock resource on an active remote node and updating the ACB. The mutex entry in the ACB is redirected to the newly created lock resource.

3.2.2.2 Conditional Waits

Conditional waits are a synchronization construct in which thread(s) block on a conditional variable and go into a suspended state. They remain in a suspended state until another thread signals the conditional variable. There are two types of signals: (i) A simple signal by a thread means that a thread will resume. Which thread is resumed is determined by the system and is currently determined by a first-in-first-out (FIFO) queue. (ii) A broadcast by a thread means that all threads currently suspended on the condition variable will resume.

CableS, implements conditional waits by spinning on a flag. Although implementations that use spinning consume processor cycles they are most common in parallel systems to reduce wake–up latency.

When a thread calls a conditional wait, *CableS* sets a local flag indicating that this thread is spinning (suspended). When another thread performs a signal it fetches flags until it finds a waiting thread (round-robin search). The signaling thread then turns off the flag so that the waiting thread can resume. When a thread broadcasts on a condition, the flag search is bypassed, and the thread directly turns off all flags

```
/* Global Thread Count */
int threads_in_barrier = NUMBER_OF_THREADS;

barrier()
{
	pthread_mutex_lock(); // Lock so the global variable can be changed

	threads_in_barrier --;

	if (threads_in_barrier == 1) // for last thread to enter barrier
	{
		threads_in_barrier = NUMBER_OF_THREADS;
		pthread_cond_broadcast(); // wake up everyone
	}
	else // any other thread except the last
	{
		pthread_cond_wait(); // wait for everyone else
	}

	pthread_mutex_unlock();
}
```

Figure 3.5: A simple barrier implementation using pthread mutex and a condition variable

associated with the condition variable.

3.2.2.3 Barriers

Global (barrier) synchronization can be implemented in *pthreads* with mutexes and conditions as illustrated in figure 3.5. However, to support legacy parallel applications efficiently we extend the *pthreads* synchronization to include a barrier operation (*pthread_barrier()*). This operation causes a global stop point among all application threads executing at the time when the barrier is performed.

The only complication with barriers is that the underlying barrier implementations, based on static initialization, is aware of the number of threads in the system. The number of threads is needed so that the barrier knows how many threads to wait for. *CableS* modifies the underlying SVM barrier call to dynamically determine the present number of threads in the system and synchronize based on this current system state.

3.2.3 Additional Thread Functionality

There are a number of minor API function specifications in *pthreads*, which provide additional control and flexibility to the programmer.

3.2.3.1 Private Thread Data

Private thread data is a way in which a thread can have data that is exclusive to itself. This is essentially local data, and *CableS* can easily implement this functionality by providing a local structure in non-shared memory that a thread can exclusively access.

3.2.3.2 Resource Attributes

Synchronization constructs and threads are defined within *pthreads* as resources that have particular attributes. The *pthreads* API provides a variety of functions to create, destroy, and modify a resources attributes. *CableS* supports resource attributes and maintains all information within ACB.

3.3 Memory System Extensions

Pthreads rely on the 'C' library's malloc/free operations for memory management. These functions are implemented in SMPs and hardware DSM systems on top of system calls that manage physical memory and application's virtual to physical mappings. However, todays SVM clusters do not provide the same level of support for global memory management. To understand what *CableS* memory system extends, we first examine memory system limitations in current state-of-the-art clusters. Then we describe the solutions proposed in *CableS* and how the new memory system works.

3.3.1 Current System Limitations

Modern SANs support direct remote memory operations, and these operations are the basis of *CableS* shared memory approach. To avoid interrupting remote processors, the NICs need to register memory regions ahead of time so that the NIC can pin down the virtual memory region in physical memory. Pinning of the data is required so that the DMA can proceed with only the NIC and not the local processors. In VMMC's case, this is the importing/exporting stage. Current SVM systems on clusters interconnected with SANs take advantage of the non interrupting remote operations to reduce overhead of propagating as well as obtaining updates of shared data. They, however, perform all registration at initialization time since the systems know the size and node location of all data. This results in a number of limitations:

(i) All shared memory has to be allocated at initialization time, since all memory registration operations happen at initialization. Thus, all physical memory in each node that will be used to back globally shared pages has to be allocated and committed during an initialization phase. It is difficult to estimate up front how much memory each application will require, especially for applications that exhibit dynamic behavior.

(ii) In most SANs today [12, 16, 13] there is a limit (a) on the number of memory regions and (b) on the total amount of memory that can be registered on the NIC. Most SANs can register up to a few thousand regions and a few hundred megabytes of memory. These limitations are related to the available translation table space on each NIC. On the other hand, applications use large amounts of global shared data that can easily exceed a few hundred megabytes. Moreover, many shared memory applications exhibit access patterns to memory that result in a working set which consists of non-contiguous shared pages. Each page in the working set of each process should be placed, whenever possible, for performance reasons, on the node where the process runs.

The registration limitations conflict with this requirement. There are three possible solutions:

(a) The primary copies of each page in the working set will be placed in a way that does not violate the registration limitations to accommodate large problem sizes. In this case, pages in the working set of a process will have their primary copies in remote nodes resulting in excessive network traffic and performance degradation.

(b) Another alternative is to place the primary copies of pages in the working set on the node where the process runs. In this case the registration limitations may be violated since there will be a large number of non-contiguous memory regions that have to be registered.

(c) Register many non-contiguous regions in one operation, including the gaps between regions. This solution results in registering essentially all the shared address space which is not feasible due to the the total amount of memory that can be registered.

(iii) In most SVM systems today, global static variables are not included in the shared address space. These are global variables that are declared statically in the user program. In the threads programming model, these variables are visible to all threads. The compiler/linker automatically allocates these variables to a designated part of the virtual address space. Since this part of the address space is not currently under the control of the shared memory protocol, static global variables are not shared across nodes. This imposes additional challenges in the process of porting existing, shared memory applications to clusters.

3.3.2 Effects on System API

These limitations impact the API provided to programmers in many ways. To support a global shared address space on a cluster without hardware support, most systems today perform the following steps:

(i) Start and initialize all the nodes that will be used to run the application at the same time. This step simplifies the task of providing a shared address space. Because all nodes are present at initialization, they can all perform, at the same time, all necessary steps of creating the shared portion of the virtual address space. However, this is a major issue for running applications where the number of nodes that will be used is not known up front.

(ii) Allocate a region of the virtual address space on each node. This step is usually fairly simple, since it involves using a system call to reserve part of the application's virtual address space. In most systems the maximum amount of memory that will be used for shared data is allocated up front, but the resources used in this case are not as critical since they are mostly page–table space in the operating system and do not usually pose a problem on today's systems.

(iii) Determine which node will maintain the primary copy of each portion of the shared memory. This step is necessary on system area networks that use direct remote operations. In most clusters today, which use SANs that support these operations, this decision is not changed during program execution, which means that it is not possible to migrate shared data pages between nodes.

(iv) Establish communication–layer mappings between the primary copy of this region and the same region on all other nodes. Establishing these mappings is done by filling in the necessary information in the NIC page table. Although this step is performed once during initialization, dynamically changing page placement (i.e. migrating pages) during program execution would require unmapping and re-mapping remote pages on the fly.

(v) Provide the application on each node with a convenient interface to the shared address space. This includes any restrictions on the usage of the shared address space. The two main restrictions imposed by most SVM clusters today are: (a) Applications cannot use static global variables since these variables are placed in special part of the virtual address space, over which the SVM protocol does not have control. (b) Applications cannot allocate and deallocate shared memory after thread creation. For instance, the SPLASH-2 applications do not require either of these conditions making it easy to port them on today's clusters. However, most other applications written for hardware shared memory systems require both conditions making it a challenge to port these applications onto SVM clusters.

3.3.3 Solutions

CableS changes the memory system so that many of these limitations are eliminated. *CableS* introduces a new way to deal with the shared memory region, which leads to a memory system that can do memory migration. Currently, *CableS* uses first touch placement policy. *CableS* can also allocate memory at any point of execution, and global static variables can be shared across nodes.

3.3.3.1 Shared Memory Region

CableS is an SVM system in which nodes can be added at any time. This means that shared memory either must have static characteristics that new nodes already know, or that the information can be communicated. *CableS* employs both of these concepts to achieve a shared memory that can migrate pages.

CableS deals with two types of shared memory on a particular node. A page is either primary copy and is owned by the node, or the page is owned by another remote node. These two types of shared memory are dealt with differently, since a remote page must be fetched and a local page must be initialized to be fetched by other nodes.

The first thing each node does to set up shared memory is to reserve a contiguous set of virtual addresses, and the region is physically allocated in the OS. This set includes all possible pages that may be used by the application. The virtual addresses must be the same across all nodes so that shared pointers reference the same memory. Each node reserves an additional contiguous set of virtual addresses that are mapped to the first set; we call this a double mapping. This is done so that the first set of addresses can be used by the user application, and the second set of addresses can be used by the protocol. For example, if the user application write protects a page, the protocol can still write to the page since the protocol's virtual address is not protected. This is needed so that the protocol can update a remote page while still protecting that page from local threads.

Once the shared memory address range has been initialized, each node then builds a larger contiguous virtual address region for primary copies. This region is backed by physical memory, is fixed in physical memory, and is exported to all other nodes in the system. When a primary copy is allocated to a node, then an address of the primary copy region is reserved, and from now on this pages data will be stored in this region. The page must also be virtually mapped into the proper address where the user application specifies the page location. Finally, page information is stored in the ACB so that remote nodes can determine what address to remotely read/write the primary copy of the page.

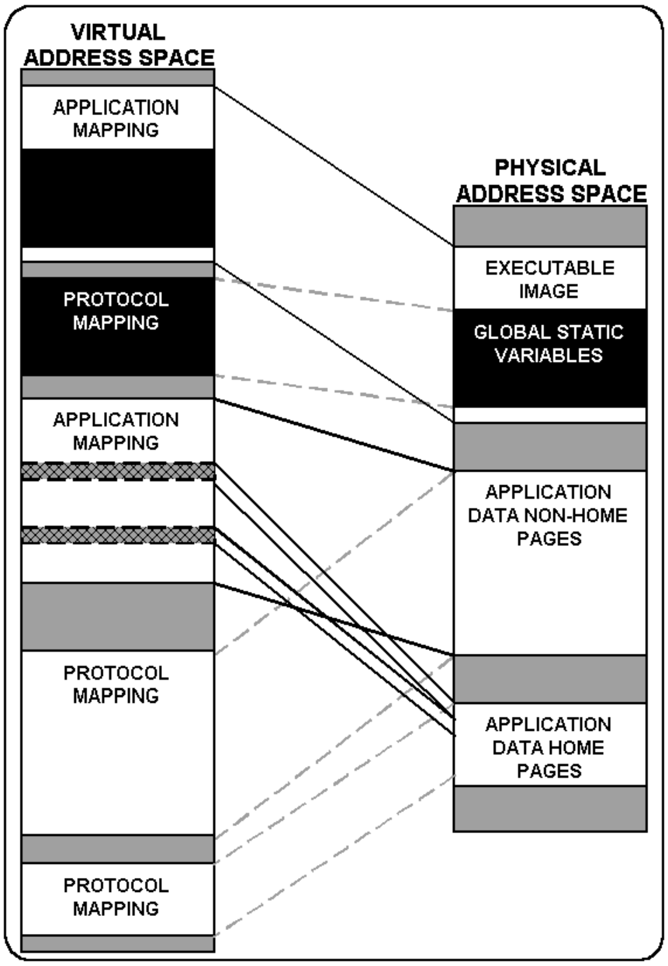


Figure 3.6: Illustration of how all the regions are mapped in Virtual and Physical l memory.

Implementation Details: *CableS* is restricted by the local OS (WindowsNT) and can only map specific sized virtual regions. Pages are 4 K-Bytes, and virtual memory regions are 64 K-Bytes. Therefore, *CableS* operates with primary copies of pages in groupings of 16 pages called *segments*. Figure 3.6 shows a potential memory layout on a node. The 16 page granularity reduces the amount of information needed to be recorded about page location and also speeds up mapping. However, the granularity increase will have an effect on performance since in many cases, pages will be improperly placed and a type of granularity based false–sharing will occur.

3.3.3.2 First Touch Policy

With the above shared memory design, *CableS* has the ability to migrate pages. Currently, *CableS* places pages based on the first touch policy. The first touch policy greatly simplifies handling the SVM protocol since there is no protocol information about the page at initialization. Also, on the first touch, there is no need for copying data since no data has been written, and this reduces communication needs.

Implementing a first touch policy requires that the system delays binding of virtual addresses until the region is first read or written. *CableS* maintains information about each memory segment that is allocated in a global directory. This information is entered when the segment is allocated. During execution, when a node touches the segment, it uses the global directory to identify whether or not the segment has been touched by anyone else. If it has already been touched, then the segment is registered from the ACB and is mapped to the corresponding region on the home node. If this is the first touch to the region, then the node becomes the home by updating the ACB and by appropriately mapping the physical pages to its shared virtual address space so that the application can use it.

Decisions about segment migration are done within the page fault handler, but within a distributed system there is potential for the same segment to be simultaneously touched by multiple threads. Synchronization of the global information and ordering simultaneous accesses to a newly allocated region is facilitated through system locks.

3.3.3.3 Any Time Allocation

Once segment migration is supported through *CableS* implementation of the first touch policy, the remaining functional addition to shared memory is how to support any time allocation. *CableS* uses a simple approach to the problem by allocating memory in shared memory in a sequential order. *CableS* starts allocating regions by incrementing an allocation pointer by the size of the requested allocation. This allocation pointer is kept within the ACB and is synchronized by a system lock. Therefore, *CableS* currently does not support full memory management, and this is a point of future work.

3.3.3.4 Global Static Variables

Global static variables are global C variables that are not explicitly allocated by the programmer, but instead, these variables are allocated by the OS's program loader. Figure 3.6 shows exactly where global static variables are located in memory. Currently, SVM systems and their related APIs, such as M4, treat global static variables as local pointers, which are later allocated explicitly by the programmer. Explicit allocation greatly simplifies management of these variables, since the system can allocate them in designated areas of the shared address space [1]; however, this imposes a burden on the programmer, since they need to allocate each global variable explicitly.

CableS uses a type quantifier to allocate these global variables in a special area within the executable image. At application initialization, *CableS* makes the first node in the system the primary copy owner for this region and establishes all necessary mappings to other nodes as they are attached to the application. Thus, static global variables of arbitrary types can be shared among system nodes.

Implementation Details: To facilitate this functionality within WindowsNT, there are two OS specific problems: (i) How to determine global static variable addresses within the executable image? (ii) How to make a second mapping of these variables for the protocol? WindowsNT provides a linker directive *#pragma data_seg(NAME)* which when included in the application causes all global variables to be allocated in an exclusive region. *CableS* examines the map file (file describing the virtual addresses of variables and functions) produced by the application and extracts the virtual address range of *NAME*. With the virtual address range, the region can almost be included within the SVM protocol. To fully integrate the virtual region, the master node still needs a second virtual mapping. WindowsNT currently does not support any mechanism to map virtual memory addresses to a second virtual address range. *CableS* gets around this problem by extending the VMMC driver firmware to support a double mapping call. Essentially, the driver has been extended as a kernel patch. With these two work arounds in place,

[1] This assumes that the designated part of the virtual address space for static variables is the same in all processes. This is true in most systems today (or can be easily enforced).

```
/* Header Files to CableS */
#include "CableS.h"

/* Declare Global variables/structures */
GLOBAL_DATA var *global_vars;
GLOBAL_DATA var other_global_vars;

/* this would be the functions that the program would execute */
work_functions() {
        /* pthread calls, pthread initializations and memory allocation calls */
}

main () {
        /* Initialise System */
        pthread_start();

        /* pthread calls, pthread initializations and memory allocation calls */

        /* end call for the system */
        pthread_end();
}
```

Figure 3.7: The current programming model for programs written for CableS

CableS can now use global static variables.

3.4 Summary

CableS provides a shared memory programming model that is very similar to a *pthreads* programming model for tightly–coupled shared memory multiprocessors, such as SMPs and hardware DSMs. Figure 3.7 shows an example of a *CableS* program.

To convert a *pthreads* program for *CableS* the following modifications are required:

1. Add the *pthread_start* and *pthread_end* library calls to initialize the system.
2. Prefix all static variables with a *GLOBAL* identifier.
3. Compile/Link with the *CableS* library.

With these changes in place, *CableS* can execute the *pthreads* application.

Chapter 4

Experiments and Results

In this chapter we evaluate *CableS* in two ways: In terms of programs it can run and the overhead it introduces. Section 4.1 describes the range of applications that have been ported and executed on *CableS*, and how these applications show that the system can run a new range of applications. Section 4.2 presents performance aspects of *CableS*. This includes micro-benchmarks we collected for individual management components of the *CableS* library (Section 4.2.1) as well as an experiment that studies the overhead introduced by *CableS*.

4.1 Pthreads Functionality

4.1.1 Pthreads

We first use the three simple conversion steps described in Chapter 3 to convert 3 publicly available *pthreads* programs for *CableS*. The programs are called: (i) Prime number (PN), which, as indicated by its name, computes all prime numbers in a user specified range. (ii) Producer–consumer (PC), a producer–consumer program which runs with two threads. (iii) Pipe (PIPE), which creates a threaded pipeline where each stage consists of a calculation.

Table 4.1 shows the *pthreads* programs which were run on *CableS* and the *pthreads* calls each of the programs makes, along with the average execution time of each function. We use this table to show

PROGRAM	C	J	L	Co	Ca	K	G	Cr	Lo	Un	Wa	Si	Br	Sp
PN	•	•	•	•	•		•	2254	23	2	6154	-	1	15677
PC	•	•	•	•			•	1.1	0.05	0.005	17	0.042	-	-
PIPE	•		•	•			•	1008	52	3	527	12	-	11249

Table 4.1: Shows pthread programs with their respective pthread function calls and execution times (in ms) for the basic API operations.
LEGEND: **C** = pthread_create, **J** = pthread_join, **L** = mutexes, **Co** = conditions, **Ca** = thread cancel, **K** = thread specific information, **G** = program uses static global variables **Cr** = create, **Lo** = mutex locks, **Un** = mutex unlock, **Wa** = condition wait, **Si** = condition signal, **Br** = condition broadcast, **Sp** = spawn time.

the actual costs of *CableS* operations during program execution (including any induced contention). PC only uses two threads, and therefore, runs on only one node. Performance-wise, PC shows the approximate cost of local API operations. PN and PIPE provide an indication of the average execution time of remote operations in *CableS*. We see that local operations are about three orders of magnitude faster than remote operations (this will be studied in more detail with micro-benchmarks). With respect to synchronization operations, conditional waits and mutex lock operations include the cost of communication and application wait time. Conditional signals and broadcasts are much faster than waits and mutexes since they involve sending only small messages to activate threads in remote nodes, and there is no blocking.

These applications are relatively simple *pthreads* programs, but the fact that these applications run on *CableS* shows that the system has gone beyond its original static limitations. To convert these programs into static, runnable applications on *GeNIMA* would require that the applications be completely restructured.

4.1.2 OpenMP Applications

Next, we use OdinMP [10] to compile 3 *SPLASH-2* applications written for OpenMP. These OpenMP applications are FFT, LU, and OCEAN.

OdinMP compiles an OpenMP program into a *pthreads* program. Each OpenMP application, when converted into *pthreads* code, uses thread management, mutex locking/unlocking, and conditional waits.

Barriers, within OdinMP, are implemented via mutexes and conditional waits. We have used *GeNIMA* barriers to improve system performance.

PROGRAM	4 processors	8 processors	16 processors
FFT	1.61	2.05	2.44
LU	3.17	3.71	7.10
OCEAN	1.33	1.43	1.92

Table 4.2: Speedups for the 3 *SPLASH-2* OpenMP programs.

Table 4.2 shows the speedups of the 3 applications. Performance evaluation of these applications is challenging since:

- OdinMP introduces conversion overhead for OpenMP management.
- Data is not configured for the first touch migration policy. When data is initialized in a warmup phase this is considered the first touch, and data is stored exclusively on the master-node. This means all pages are allocated on one node and the communication costs are high.
- These applications are written SMP-type shared memory architectures and are not optimized for DSM (especially software) systems so the speedups are not indicative of the actual performance that can be obtained on cluster systems.

Based on these challenges we have not chosen to compare *SPLASH-2* OpenMP programs with the M4 versions of the applications. *CableS*, however, now has the ability to run OpenMP programs. OpenMP programs and the OpenMP standard are developed for DSM systems and SMPs. For this reason, *CableS* memory extensions are key to executing OpenMP applications on an SVM cluster. Other cluster systems can also execute OpenMP programs [27, 34], but they deal with the memory issues differently. They solve the existence of global static variables by using their custom compiler to correctly allocate all needed memory. We have not compared our results with their results since we do not currently have common applications to compare.

4.1.3 SPLASH-2 Applications

To include the applications that run on *GeNIMA*, *CableS* provides a *pthreads* implementation of the M4 macros. *CableS* implementation slightly differs from *GeNIMA* since *CableS* does not have each node initializing data. Instead, *CableS* starts with the master node, which performs initialization, and then the master node starts the threads on each of the newly introduced nodes. *CableS* M4 macro implementation also uses barriers, but not barriers implemented with *pthreads* mutexes and conditional waits. Performance is better with the original *GeNIMA* barriers as will be demonstrated in Section 4.2.1.

With this implementation, *SPLASH-2* applications can now be executed on *CableS* as well as any application written with the M4 macros.

4.2 Performance Analysis

We now examine the performance of *CableS* by looking both at micro-benchmarks and actual applications. We use micro-benchmarking to examine the overheard of each of the individual system functions involved in the *pthreads* API and the memory system. We also use the *SPLASH-2* applications to compare the original SVM system, *GeNIMA*, with *CableS* to study the overhead introduced by *CableS*.

4.2.1 Micro-benchmarks

We begin examining the performance of *CableS* by examining each piece of the design. Specifically, we provide and analyze execution times for all of the *pthreads* functional units. Then we examine the costs associated with first touch migration, and finally, we provide a comparison between *GeNIMA*'s original barrier implementation and an implementation using *pthreads* mutexes and conditional waits.

Table 4.3 shows the results from our micro-benchmarking. We obtain these numbers on 2 and 4 node systems. For these tests there is no contention in system resources and there is no shared memory protocol activity (no application shared data is used). We run experiments multiple times and average costs over all executions.

CableS Mechanism	Total	Local *CableS*	Remote *CableS*	Local OS	Communication
attach node	3690 ms	1 ms	1978 ms	523 ms	1188 ms
local thread create	766 μs	140 μs	-	626 μs	-
remote thread create	819 μs	110 μs	40 μs	-	47 μs
local mutex lock (first time)	33 μs	10 μs	-	-	23 μs
local mutex lock	4 μs	4 μs	-	-	-
remote mutex lock (first time)	122 μs	15 μs	35 μs	-	72 μs
remote mutex lock	101 μs	16 μs	35 μs	-	50 μs
mutex unlock	6 μs	6 μs	-	-	-
conditional wait	30 μs	5 μs	-	-	15 μs
conditional signal	100 μs	14 μs	-	2 μs	85 μs
conditional broadcast	110 μs	7 μs	-	2 μs	101 μs
GeNIMA barrier	70 μs	-	-	-	65 μs
pthreads barrier	13 ms	-	-	-	-
segment migration on ACB owner (first time)	159 μs	92 μs	-	67 μs	-
segment owner detect on ACB owner	1 μs	1 μs	-	-	-
segment migration (first time)	252 μs	95 μs	-	65 μs	92 μs
segment owner detect (first time)	23 μs	1 μs	-	-	22 μs
segment owner detect	1 μs	1 μs	-	-	-
administration request	20 μs	2 μs	-	-	18 μs

Table 4.3: CableS execution times for the basic events. For node attach the remote OS time is 2031 ms and for remote create the remote OS time is 622 μs.

Node attaching is the most expensive system operation since a new node needs to perform all initialization with other nodes in the system. This time will increase as more nodes are introduced since more import/export links need to be established. Some elements of node attaching are done in parallel, and the breakdowns will not exactly add up to the perceived total. Additionally, the communication time includes the time for importing nodes, which potentially includes waiting time since a buffer can not be imported until the other node has exported it.

Unlike the remote mutex cost, local mutex cost refers to the instance when the mutex was last locked/unlocked by another local thread and there is no communication involved. The first time mutex cost refers to the case where the mutex is acquired for the first time. At that time the acquirer needs to perform additional bookkeeping.

Condition wait and signal involves mostly local processing and ACB direct read/write operations to update and retrieve condition information. These overheads are relatively low and depend only on direct remote operation costs, so they are not expected to vary much with the number of nodes in the system. On the other hand, the current implementation of condition broadcast depends on the number of nodes waiting on the condition and involves processing for each node in the system and communication (one remote write) for each node waiting on the condition.

Segment migration determines if a segment has an owner and taking ownership of the page if it does not. These two actions can be taken by any node, but the segment state is maintained on one node so there is remote and local migration based on the need for this state information. Segment migration costs slightly more in the remote case since information needs to be read and written from and to the ACB owner node. Owner detection is the additional page fault processing cost when the segment does not need to actually migrate but needs to be detected. This processing is usually small but depends on whether the segment information is locally cached.

We also include execution values for two types of barriers. *GeNIMA* barriers are implemented in the original SVM system as native operations. The *pthreads* barrier is implemented using *pthreads* primitives: a mutex, a condition variable, and a shared variable. Since each synchronization variable is handled by a single node, this node becomes a centralization point. The difference in performance is due to the point–to–point nature of synchronization used in the *pthreads* version.

Each barrier was used in a for loop consisting of 1000 iterations. The first and last barrier were excluded from the results as well as the loop overhead. Table 4.3 clearly shows that the original *GeNIMA* barrier implementation is the best performer. *GeNIMA*'s barrier implementation is better because the barrier synchronizes with all other nodes. Barriers which use mutexes as the first stage of synchronization cause a serial point of entrance in which only one node can enter at a time, and therefore, these forms of synchronization cause unnecessary thread blocking.

4.2.2 *GeNIMA* vs *CableS*

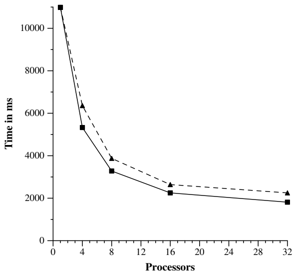
The next set of experiments performed on *CableS* provides insight on how *CableS* differs from the original system, *GeNIMA*. To run the *SPLASH-2* applications on *CableS* we have provided an implementation of the M4 macros.

In *CableS* we use the *pthreads_barrier()* call we introduced, as opposed to a mutex-based implementation of barriers. This choice is made for fairness reasons. Specific knowledge provided by the SPLASH-2 applications about global synchronization and exploited in the original SVM system should be exploited in *CableS* as well. Since *pthreads* was not designed for parallel applications that frequently use global synchronization we provide this call for the purpose of this comparison.

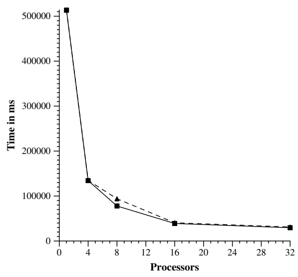
Each application was run on both *GeNIMA* and *CableS*. Both systems used not only the same barrier implementation, but the same data placement optimizations. The M4 macros provide a page placement macro which we convert into memory touches in *CableS*. *CableS*, however, needs to modify the *SPLASH-2* code slightly so that three initialization phases are strictly followed: (i) the code allocates data first, (i) then data is placed, and finally, (iii) the data is initialized. The original applications meld these phases together, and if these applications are run without modification on *CableS* the initialization instructions are considered first touch, which falsely places pages on the Master node.

The applications we use are: FFT, LU, OCEAN, RADIX, WATER-SPATIAL, WATER-SPAT-FL, RAYTRACE, and VOLREND. These applications have been used in a number of recent studies. We use the versions from [22]. Their characteristics and behavior have been studied in [37, 38, 24]. FFT, LU, OCEAN share a common characteristic in that they are optimized to be single-writer applications; a given word of data is written only by the processor to which it is assigned. Given appropriate data structures they are single-writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are almost all local. The applications have different inherent and induced communication patterns [37, 20], which affect their performance and the impact on SMP nodes. Water [37] and Radix [8, 18], exhibit more challenging data access patters for shared memory systems. These access patterns may result in false sharing depending on the level of sharing granularity. The problem sizes we use are: FFT-m22, LU-n4096, OCEAN-n514, RADIX-(n16777216, m33554432), WATER-32768 molecules, VOLREND-head, and RAYTRACE-car.512.env.

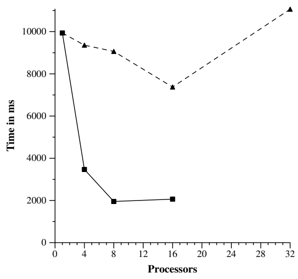
Figure 4.1 shows the execution times of each application in both system configurations for 1, 4,



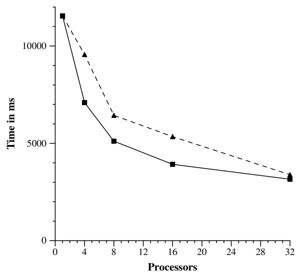
(a) FFT



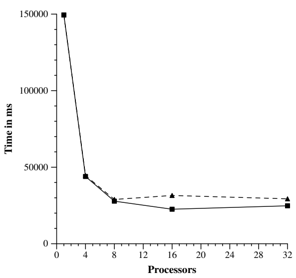
(b) LU



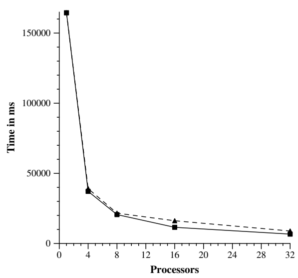
(c) OCEAN



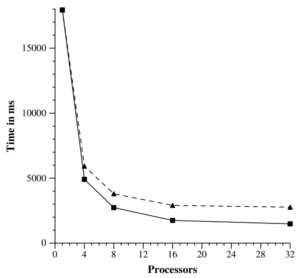
(d) RADIX



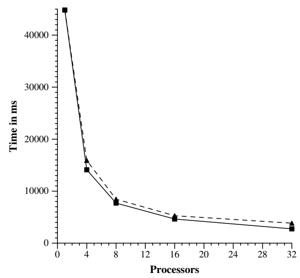
(e) WATER-SPATIAL



(f) WATER-SPAT-FL



(g) VOLREND



(h) RAYTRACE

Figure 4.1: SPLASH-2 M4 vs M4-pthread executions with 1, 4, 8, 16, and 32 processors. Solid line represents the M4 execution times, and dashed line represents the M4-pthread executions times

8, 16, and 32 processors. We see that for five out of the eight applications, FFT, LU, RAYTRACE, WATER-SPATIAL, and WATER-SPAT-FL, the overhead of *CableS* is within 25% of the original system in the 32–processor configuration.

The other three applications, OCEAN, RADIX, and VOLREND exhibit different behavior under the two systems. The large difference in OCEAN is due to a protocol optimization in the the original system which is not supported by *CableS*. From the execution traces the original system could not execute OCEAN with 32 processors, because of memory registration limits. *CableS* with its memory extensions was able to run OCEAN on 32 processors. RADIX and VOLREND are more interesting: Since *CableS* relies on re-mapping of virtual memory segments to dynamically allocate homes, home allocation is restricted due to WindowsNT limitations to a granularity of 64 KByte segments as opposed to the 4 KByte page size. In these applications, unlike the first set of applications, the large mapping

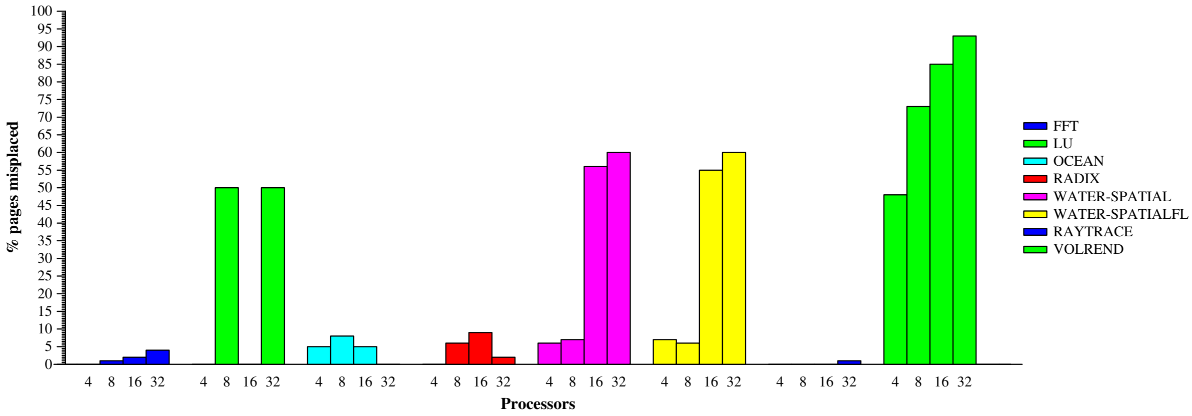


Figure 4.2: SPLASH-2 applications with their percentage of page misplacements for 4,8,16, and 32 processors

granularity results in improper page placement for many pages and in high protocol and communication costs.

Figure 4.2 shows the percentage of pages misplaced in *CableS* compared to the page–placement in the original system. FFT, OCEAN, RADIX, and RAYTRACE exhibit less than 10% of misplaced pages with small performance impact. We define a misplaced page as a page that *CableS* places on a different home node when compared against *GeNIMA*. For instance, in FFT misplaced pages cause approximately 2000 additional page faults per node. LU, WATER-SPATIAL, WATER-SPAT-FL, and VOLREND exhibit a large number of misplaced pages. This is not a problem in the two versions of WATER and LU due to the large computation to communication ratio (LU) and the infrequent synchronization (LU and WATER). LU exhibits a high percentage of misplaced pages at 8 and 32 processors. However, the application suffers only 200 additional read page faults which adds 50ms–100ms to the execution time. Thus the performance of the parallel section is almost identical between the two configurations. Similarly, WATER-SPAT-FL is not affected by the misplaced pages. In VOLREND the misplaced pages result in high performance degradation. For example, with 32 processors the application shows a speedup of 12.09 on the original system, as opposed to only 6.49 on *CableS*.

Overall, we see that *CableS* introduces additional overhead in applications tuned for the shared memory abstraction only when it results in improper data placement due to the 64KByte–granularity mapping restrictions in WindowsNT. The impact of a misplaced page is not easily quantifiable and depends on the application behavior.

Chapter 5

Conclusions

5.1 Limitations and Future Work

The main limitation of this work is that even though we provide a single system image with respect to memory and thread management, networking and files are still managed separately by each node. This means that applications written for clusters will need to consider how to deal with files and network resources within the distributed system. For example, how should the programmer deal with a socket connection created on one node that another node wishes to use? Supporting the full single system image will further extend the range of applications that can run on a cluster. There are also aspects of the memory system that need to be studied and improved. In [21] we describe some of the memory system limitations, and to what degree *CableS* has solved these problems.

The current solution for global static variables does not address issues related to library reuse. Basically, we would like to avoid annotating every variable. This means that we need some mechanism to extract the virtual address range without using our current solution. The problem, however, is if we can include all global static variables then how do we deal with global variables in libc and other libraries. These variables can be internal OS variables and have not been designed with global cluster distribution in mind. Two solutions are: (i) extend *CableS* to share the complete address space, or (ii) share all variables in the executable image. Both of these solutions will work, but careful consideration must be taken into determining how other libraries (like libc) will interact.

With the dynamic system, there may be new avenues of research into SVM cluster performance with respect to automatic memory placement. Migration of memory to threads, and threads to memory has been examined in a variety of systems. This avenue should be studied for a system like *CableS* to see if the system supports existing schemes, or provides new optimizations. The most interesting problems in memory placement come from the unknown behavior of persistent applications (applications that are supposed to execute for a very long time). *CableS* provides most of the infrastructure to investigate placement policies. However, two things are still needed. Firstly, SVM benchmark applications are needed which would provide defined points where data reorganization is beneficial to the application. Secondly, the migration scheme needs to be closely analyzed with respect to the underlying SVM protocol, and there remain questions on how and when can data be efficiently migrated during run-time.

CableS is a first step towards enabling a wider range of new applications to run unmodified on clusters. Thus, we now have the potential to examine commercial workloads, such as the Apache world-wide-web server and Kaffe Java Virtual Machine that are usually run on small–scale SMPs.

5.2 Conclusion

In this work we design and implement a system that provides a single system image for SVM clusters with modern SANs. Our system supports the *pthreads* API and within this API, provides dynamic thread and memory management as well as all synchronization primitives. Our memory management system deals with limitations of modern SANs that support direct remote memory operations. We show that this system is able to support *pthreads* applications written for more tightly–coupled, hardware shared memory multiprocessors. We use a wide suite of programs to demonstrate the viability of our approach to make clusters easier to use in new areas of applications.

Our results show that existing applications can run on top of *CableS* with few or no modifications and applications tuned for performance on shared memory systems incur additional overhead only when the 64-KByte granularity of mapping physical to virtual memory in WindowsNT results in improper data placement.

Bibliography

[1] http://sources.redhat.com/cygwin/.

[2] International standard ISO/IEC 9945-1: 1996 (E) IEEE std 1003.1, 1996 edition (Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995) Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language] .

[3] www.openmp.org.

[4] ANTONIU, G., BOUGE, L., HATCHER, P., MACBETH, M., MCGUIGAN, K., AND NAMYST, R. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001.

[5] ARIDOR, Y., FACTOR, M., TEPERMAN, A., EILAM, T., AND SCHUSTER, A. A high performance cluster JVM presenting a pure single system image. In *ACM Java Grande 2000 Conference* (2000).

[6] BASU, A., WELSH, M., AND VON EICKEN, T. Incorporating memory management into user-level network interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[7] BILAS, A., LIAO, C., AND SINGH, J. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the The 26th International Symposium on Computer Architecture* (Atlanta, Georgia, May 1999).

[8] BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures* (July 1991), pp. 3–16.

[9] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W. Myrinet: A gigabit-per-second local area network. *IEEE Micro 15*, 1 (Feb. 1995), 29–36.

[10] BRUNSCHEN, C., AND BRORSSON, M. OdinMP/CCp - a portable implementation of OpenMP for c. *The 1st European Workshop on OpenMP* (1999).

[11] CHEN, Y., BILAS, A., DAMIANAKIS, S., DUBNICKI, C., AND LI, K. UTLB: A mechanism for address translation on network interfaces. In *Proceedings of the Eighth International Conference Architectural Support for Programming Languages and Operating Systems ASPLOS* (San Jose, CA, Oct. 1998), pp. 193–203.

[12] COHEN, D., FINN, G. G., FELDERMAN, R., AND DESCHON, A. The use of message-based multicomputer components to construct gigabit networks. *ACM Computer Communication Review 23*, 3 (July 1993), 32–44.

[13] DUBNICKI, C., BILAS, A., CHEN, Y., DAMIANAKIS, S., AND LI, K. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects* (Aug. 1997).

[14] DWARKADAS, S., HARDAVELLAS, N., KONTOTHANASSIS, L., NIKHIL, R., AND STETS, R. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of the Second Merged Symp. IPPS/SPDP 1999)* (1999).

[15] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)* (May 1990), pp. 15–26.

[16] GILLETT, R., COLLINS, M., AND PIMM, D. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96* (Feb. 1996).

[17] HILL, M., ZHOU, Y., SCHOINAS, I., IFTODE, L., LI, B. R. T. K., SINGH, J. P., AND WOOD, D. A. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. Tech. Rep. TR-535-96, Department of Computer Science, Princeton University, December 1996, 10 Pages.

[18] HOLT, C., SINGH, J. P., AND HENNESSY, J. Architectural and application bottlenecks in scalable DSM multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (May 1996).

[19] IFTODE, L., DUBNICKI, C., FELTEN, E. W., AND LI, K. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture* (Feb. 1996).

[20] IFTODE, L., SINGH, J. P., AND LI, K. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)* (May 1996).

[21] JAMIESON, P., AND BILAS, A. CableS : Thread control and memory system extensions for shared virtual memory clusters. In *To be presented at the 8th International Symposium on High-Performance Computer Architecture* (2002).

[22] JIANG, D., O'KELLEY, B., YU, X., BILAS, A., AND SINGH, J. P. Application scaling under shared virtual memory on a cluster of SMPs. In *The 13th ACM International Conference on Supercomputing (ICS'99)* (June 1999).

[23] JIANG, D., O'KELLEY, B., YU, X., KUMAR, S., BILAS, A., AND SINGH, J. P. Application scaling under shared virtual memory on a cluster of SMPs. In *Proceedings of the 1999 international conference on Supercomputing* (Rhodes, Greece, June 1999).

[24] JIANG, D., SHAN, H., AND SINGH, J. P. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming* (June 1997).

[25] KELEHER, P., COX, A., DWARKADAS, S., AND ZWAENEPOEL, W. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference* (Jan. 1994), pp. 115–132.

[26] LINDHOLM, T., AND YELLIN, F. The java virtual machine specification, second edition, 1999.

[27] LU, H., HU, Y. C., AND ZWAENEPOEL, W. OpenMP on networks of workstations. In *Proceedings Supercomputing* (1998).

[28] MA, M., WANG, C., AND LAU, F. Jessica: Java-enabled single-system-image computing architecture, 1998.

[29] MUELLER, F. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference* (Jan. 1993), pp. 29–41.

[30] MUELLER, F. Distributed shared-memory threads: Dsm threads. *Workshop on Run-Time systems for Parallel Programming* (April 1997), 31–40.

[31] MYRICOM INC. The Myrinet on-line documentation. http://www.myri.com:80/scs/documentation, 1996.

[32] PHILIPPSEN, M., AND ZENGER, M. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience 9*, 11 (1997), 1225–1242.

[33] SCALES, D., AND GHARACHORLOO, K. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles* (Oct. 1997).

[34] SCHERER, A., LU, H., GROSS, T., AND ZWAENEPOEL, W. Transparent adaptive parallelism on nows using OpenMP. In *Principles Practice of Parallel Programming* (1999), pp. 96–106.

[35] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)* (Oct. 1997).

[36] STETS, R., DWARKADAS, S., KONTOTHANASSIS, L., , RENCUZOGULLARI, U., AND SCOTT, M. L. The effect of network total order, broadcast, and remote-write capability on network-based shared memory computing. In *The 6th IEEE Symposium on High-Performance Computer Architecture* (January 2000).

[37] WOO, S., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)* (May 1995).

[38] WOO, S. C., SINGH, J. P., AND HENNESSY, J. L. The performance advantages of integrating message-passing in cache-coherent multiprocessors. In *Proceedings of Architectural Support For Programming Languages and Operating Systems* (1994).

[39] YU, W., AND COX, A. Java/DSM: A platform for heterogeneous computing, 1997.

[40] ZHOU, Y., IFTODE, L., AND LI, K. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium* (Oct. 1996).