CableS : Thread Control and Memory Management Extensions for Shared Virtual Memory Clusters

Peter Jamieson and Angelos Bilas

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{jamieson,bilas}@eecg.toronto.edu

Abstract

Clusters of high-end workstations and PCs are currently used in many application domains to perform large-scale computations or as scalable servers for I/O bound tasks. Although clusters have many advantages, their applicability in emerging areas of applications has been limited. One of the main reasons for this is the fact that clusters do not provide a single system image and thus are hard to program. In this work we address this problem by providing a single cluster image with respect to thread and memory management. We implement our system, CableS *(***C***luster en***able***d thread***S***), on a 32-processor cluster interconnected with a low-latency, high-bandwidth system area network and conduct an early exploration of the costs involved in providing the extra functionality. We demonstrate the versatility of* CableS *with a wide range of applications and show that clusters can be used to support applications that have been written for more expensive tightly–coupled systems, with very little effort on the programmer side: (a) We run legacy* pthreads *applications without any major modifications. (b) We use a public domain OpenMP compiler (OdinMP [8]) to translate OpenMP programs to* pthreads *and execute them on our system, with no or few modifications to the translated* pthreads *source code. (c) We provide an implementation of the M4 macros for our* pthreads *system and run the SPLASH-2 applications. We also show that the overhead introduced by the extra functionality of* CableS *affects the parallel section of applications that have been tuned for the shared memory abstraction only in cases where the data placement is affected by operating system (WindowsNT) limitations in virtual memory mappings granularity.*

1. Introduction and Background

The shared memory abstraction is used in an increasing number of application areas. Most vendors are designing both small–scale symmetric multiprocessors (SMPs) and large–scale, hardware cache-coherent distributed shared memory (DSM) systems, targeting both scientific and commercial applications. However, there is still a large gap in the configuration space for affordable and scalable shared memory architectures, as shown in Figure 1. Shared memory clusters are an attractive approach for filling in the gap and providing affordable and scalable compute cycles and I/O.

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 provide memory–to–memory latencies of under 10μs and bandwidth in the order of hundreds of MBytes/s, limited mainly by the PCI bus. For instance, the cluster we are developing at the University of Toronto uses Myrinet as the interconnection network and currently provides one–way, memory–to–memory latency of about 7.8μs and bandwidth of about 125MBytes/s. Similar clusters are being built at many other research institutions. Recent work has also targeted the design of efficient shared virtual memory (SVM) protocols for such clusters [29, 20, 32, 17]. These protocols take advantage of features provided by SANs, such as low–latencies for short messages and direct remote memory operations with no remote processor intervention [15, 11, 10], to improve system performance and scalability [20].

Despite the many advantages of clusters, their use is not widespread. One of the main reasons is that despite the progress on the performance side, it still is a very challenging task to port existing applications or to write new ones for the shared memory programming APIs provided by clus-

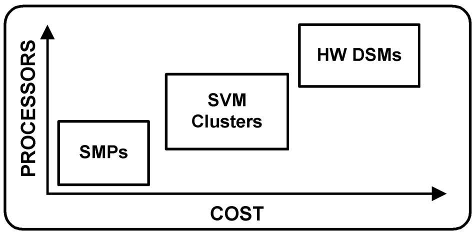


Figure 1. The architectural space for shared memory systems. Shared memory clusters may be able to fill in a gap in the cost-performance range and provide application portability across architectures that covers the full spectrum.

ters. Many shared memory clusters are written according to M4–macros rules (Figure 2). Although these APIs provide sufficient primitives to write parallel programs, they also impose several restrictions: (i) Processes cannot always be created and destroyed on the fly during application execution. This is especially true on clusters that use modern SANs with support for direct remote memory operations. In these systems all nodes/processes need to be present at initialization to perform the initial mappings. (ii) Programmers allocate shared memory only during program initialization and should not free memory until the end of execution. For instance, these rules are followed by the SPLASH-2 applications that are usually used for evaluating shared memory systems. Also, placement of primary copies of shared pages is limited due to restrictions imposed by SANs on both the number of regions as well as the total amount of memory space that can be mapped. (iii) In most shared memory clusters the synchronization primitives supported are *lock/unlock* and *barrier* primitives. However, more modern APIs support conditional waits as well as other primitives. These, and other limitations are not very important for large classes of scientific applications that are well structured. However, they pose important obstacles for using clusters in areas of applications that exhibit a more dynamic behavior, such as commercially–oriented applications. In essence, current clusters that support shared memory provide a very limited single system image to the programmer with respect to process, memory management, and synchronization.

The goal of this work is to overcome the above limitations for existing and new applications written for the shared memory model. To achieve this we provide a more complete and functional single cluster image to the programmer by designing and implementing a *pthreads* interface on top of our cluster. We also perform a preliminary evaluation of the costs associated with the additional system functionality. Our system, *CableS* (**C**luster en**able**d

```
/* 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. The programming template for many SVM systems. At Stage 1: all global variables are declared as pointers. Stage 2: global variables are allocated between the initialization sequence. Stage 3: threads are created.

thread**S**), allows existing *pthreads* programs to run on our system with minor modifications. Programs can dynamically create and destroy threads, allocate global shared memory throughout execution, and use synchronization primitives specified by the *pthreads* API. More specifically, our system provides support for:

Dynamic memory management: *CableS* addresses a number of issues with respect to memory management. (a) It provides all necessary mechanisms to support different memory placement policies. Currently, *CableS* implements first touch placement, but can be extended to support others as well. (b) It provides the ability to allocate global, shared memory dynamically at any time during program execution. (c) It deals with static global variables in a transparent way.

Dynamic node and thread management: *CableS* allows the application to dynamically create threads at any point during execution. Currently, new threads are allocated to nodes with a simple, round–robin policy. When threads exceed a maximum number, a new node is attached to the application. On the fly, the system performs all the necessary initialization to support the *pthreads* API.

Modern synchronization primitives: *CableS* supports the conditional wait primitives.

The main limitation of *CableS* is that, although it provides a single system image with respect to thread management, memory management, and synchronization support, it does not yet include file system and networking support across cluster nodes. The general issue here is that operating system (OS) state is still not shared across nodes. However, this is beyond the scope of this work and we do not examine this further.

We demonstrate the viability of our approach and the versatility of our system by using a wide range of applications: (a) We run existing *pthreads* applications with minor modifications. (b) We use a public–domain OpenMP compiler, OdinMP [8], that translates OpenMP programs to *pthreads* programs for shared memory multiprocessors and run the translated OpenMP programs on our system. OdinMP [8] is designed for shared memory multiprocessors that support *pthreads*. Our system supports the OpenMP programs with no modifications to the OpenMP source and minor modifications to the *pthreads* sources. (c) We provide an implementation of the M4 macros for *pthreads* and we run some SPLASH-2 applications. We also show that the overhead introduced by the extra functionality affects the parallel section of applications that have been tuned for the shared memory abstraction *only* in cases where data is improperly placed due to OS limitations in virtual memory mappings granularity. In the SPLASH-2 applications most overhead is introduced during application initialization and termination.

The rest of the paper is organized as follows. Section 2 describes the design of *CableS*. Section 3 presents our experimental results. Section 4 presents related work and Section 5 discusses our high level conclusions.

2. System Design

CableS is a system built upon an existing state-of-the-art, tuned SVM system, *GeNIMA*, which provides the basic shared memory protocol. *CableS* supports a full *pthreads* (POSIX Threads IEEE POSIX 1003.1 [1]) API, which enables legacy shared memory applications written for traditional, tightly coupled, hardware shared memory systems to run on shared memory clusters. Within the *pthreads* API, *CableS* addresses the following issues: (i) Dynamic global memory management. (ii) Dynamic thread management. (iii) Support for modern synchronization primitives. Our main contribution is our memory extensions to support a transparent dynamic memory management.

2.1. Memory Subsystem

We deal with memory management issues at both the communication and SVM levels. First, we explain how the communication layer is coupled with the SVM layer. Secondly, we describe what limitations currently exist at the communication level, and how these limitations effect the system API. Finally, we describe how *CableS* addresses these limitations.

Nodes in modern clusters are usually interconnected with low–latency, high–bandwidth SANs that support user–level access to network resources [15, 10, 7]. By allowing users to directly access the network without OS intervention, these systems dramatically reduce latencies compared to traditional TCP/IP–based local area networks. Moreover, to further reduce latencies, SANs usually support direct remote memory operations: Reads and writes to remote memory are performed without remote processor intervention. This mechanism provides fast access to remote memory within a cluster. SVM systems on clusters interconnected with SANs take advantage of these features to reduce the overhead associated with propagation and updating of shared data [29, 20].

In these mechanisms, a node maps one or more regions of remote memory to the local network interface card (NIC) and then performs direct operations on these regions without requiring OS or processor intervention on the remote side. This mapping operation is called registration and usually requires work at both the sending as well as the receiving NIC.

2.1.1 Current SAN Limitations

Due to hardware resource limits (e.g., memory on the NIC) SANs [15, 6, 14, 11], incur a number of limitations:

One limitation is the number of memory regions that can be registered on the NIC (usually a few thousand). Usual solutions in SVM protocols to reducing the number of regions are: (a) To group shared pages in regions and map them in one operation. In this case, pages in the working set of a process may have their primary copies (homes) in remote nodes resulting in excessive network traffic and performance degradation. (b) To place the primary copies of pages in the working set on the node where the process runs. In this way the registration limitations may be violated since there will be a large number of non-contiguous memory regions that have to be registered. (c) To register the many, non-contiguous regions in one operation, including the gaps between regions. However, this results in registering essentially all the shared address space. This is not feasible due to the total amount of memory which can be registered. None of these solutions is satisfactory.

Another limitation is the total amount of memory that can be registered on the NIC (usually a few hundred MBytes). The only solution to this is dynamic management of registered memory [9, 4], which introduces additional costs but may allow for larger amounts of remote memory to be used for direct operations. Although we are exploring this alternative at the NIC level, this direction is beyond the scope of this work.

The amount of memory that can be pinned due to OS limitations, where a pinned page means that the page will never be swapped out of main memory. This is a fundamental limit in current OS design that cannot be overcome in SVM systems.

2.1.2 Current Limitations on SVM APIs

The above limitations inflict a number of constraints on SVM systems with respect to memory management:

Allocation and deallocation of global shared memory is limited. Many systems today allocate all global shared memory at initialization and deallocate it at program termination. Furthermore, static memory management requires all participating nodes to be present at application startup time. This simplifies significantly the task of providing a shared address space. Since 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. Thus, resource requirements in memory and nodes need to be known up front, which is not always possible with applications that exhibit dynamic behavior, and in addition, resources may be overall, poorly utilized.

The amount of process virtual memory that can be allocated to global shared data is constrained. In many cases, although processes have available virtual address space, and the cluster has enough physical memory to efficiently support large problem sizes, the virtual memory cannot be used due to the above SAN limitations. This is going to be especially true as 64–bit processors are used in commodity clusters. Moreover, many shared memory applications exhibit access patterns to memory that result in a working set which consists of non-contiguous shared pages, further complicating registration issues.

There is no dynamic assignment of primary shared page copies to nodes (home placement and migration). The complex and expensive registration phase results usually in static management of the primary copies (homes) of shared pages. Thus, SVM systems, which take advantage of remote DMA (direct memory access) operations, do not usually provide dynamic and on–demand memory placement.

In the threads programming model, global shared variables are visible to all threads; however, this is not true in most SVM systems. Global static variables are not usually included in the shared address space. The compiler/linker automatically allocates these variables to a designated part of the virtual address space that is not part of the global address space. This imposes additional challenges in the process of porting existing shared memory applications to clusters.

So far, most of these issues have been dealt with by avoiding the problems. For instance, the SPLASH-2 applications have been written in a way that avoids all dynamic memory management issues. However, this is not (or should not be) true for most other shared memory applications, such as *pthreads* applications. The result is inflexible systems that are not easy to program. Tables 1 and 2 summarize the SAN limitations and constraints they impose on SVM systems.

SAN Limitations	Affects base SVM	Affects *CableS*
Number of registered regions	Yes	No
Total amount of registered memory	Yes	Yes
Total amount of pinned memory	Yes	Yes

Table 1. SAN limitations and constraints.

SVM Limitations	Addressed by base SVM	Addressed by *CableS*
Dynamic shared memory allocation and deallocation	No	Yes
Amount of virtual memory used for shared data	No	Partially (NIC)
Dynamic page placement	No	Yes
Global static shared variables	No	Yes

Table 2. SAN and SVM limitations and constraints.

2.1.3 Proposed Solution

Table 2 shows the issues that *CableS* deals with. *CableS* addresses the issues associated with the number of exported regions in SANs and with most SVM memory management limitations.

Reducing the number of registered regions: *CableS* uses double virtual mappings [19] for home pages. Initially, one contiguous part of the physical address space in each node is used to hold the primary copies of shared pages that will be allocated to this node. This part of the physical address space is always pinned, since it will be accessed remotely by other nodes (Fig. 3). The primary copies are mapped twice to the virtual address space of the process. One mapping is to a contiguous part of the virtual address space and is used only by the protocol to register the home pages with one operation, avoiding the registration limitations mentioned above. The second mapping is used by the application to access the shared data. For this mapping, the home pages are divided in groups of fixed size (in the current system 64 KBytes) and are mapped to arbitrary locations in the virtual address space of the process. It is important to note that these locations are not necessarily contiguous.

Dynamic allocation and deallocation: As the application requires more shared memory, it first allocates a region in the global virtual address space. Then, it determines which node will hold the primary copies of these pages according to some placement policy (currently first touch). When a home page is touched: (a) The home node extends the home pages section and registers the additional pages with the NIC. Then, it maps the virtual memory region to the newly allocated home pages (Fig. 3). As the pri-

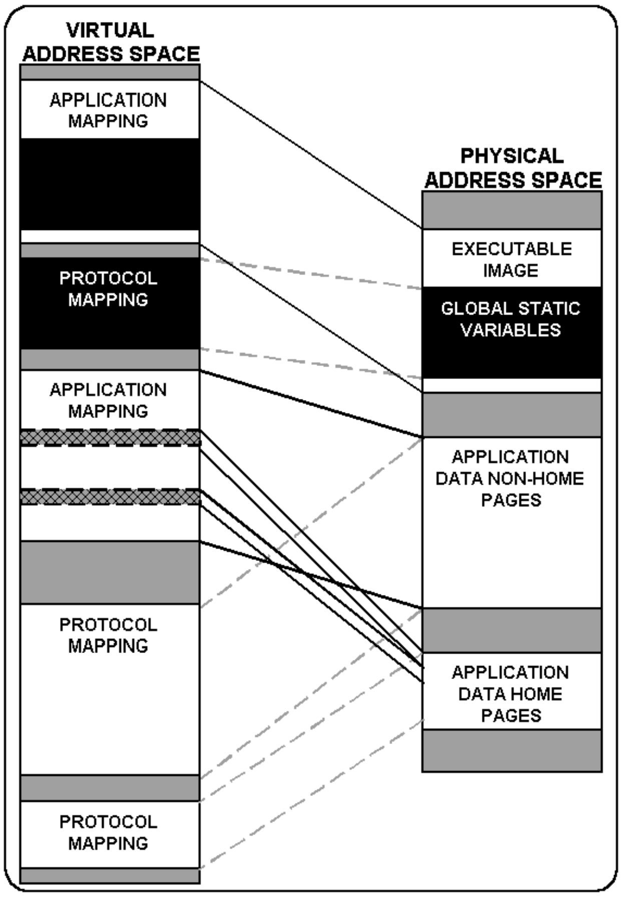


Figure 3. The virtual memory map for the application and protocol regions.

mary copies of shared pages are placed in different nodes, the home pages portion of the physical address space is mapped to non-contiguous regions of the shared virtual address space in the home node. (b) Every other node in the system, registers the newly allocated virtual memory region with the NIC so that each node can fetch updates from the primary copies and rely on the OS to allocate arbitrary physical frames for these pages. The contiguous portion of the virtual address space that is exported is currently attached and exported as a single region. It is up to the communication layer to dynamically handle this region of registered virtual memory without statically reserving physical memory and NIC resources [9, 4]. **Dynamic placement and migration:** Implementing a dynamic placement policy requires that the system delays binding of virtual addresses until later in program execution. For instance, implementing a first touch policy, requires delaying binding until it is first read or written. *CableS* maintains information about each memory segment allocated in the global directory. During execution, when a node touches the segment, it uses the global directory to identify if the segment has been touched by anyone else. If it has, then the segment is registered with the NIC and is mapped to the corresponding region on the home node (Fig. 3). If this is the first touch to the region, then the node becomes the home by updating the global information and by appropriately mapping the physical pages to its shared virtual address space so that the application can use it. Synchronization of the global information and ordering simultaneous accesses to a newly allocated region is facilitated through system locks. Table 2 mentions this feature as fully supported, but although we provide all necessary mechanisms for page migration, we do not yet provide a policy.

Amount of available virtual memory: The amount of virtual memory that can be used for shared data depends on the number of regions and on the total amount of memory the NIC can register and pin. *CableS* partially addresses this by taking advantage of the double mapping. Instead of exporting non-contiguous pages in the application map (Fig. 3), we export the single contiguous protocol mapping of the home pages (Fig. 3). Although, the total amount of memory that can be registered and pinned is still limited by the NIC, our approach allows certain applications, e.g OCEAN, to run larger problem sizes.

Global static variables: *CableS* deals with global static variables in a transparent way. It uses a type quantifier *GLOBAL* in WindowsNT:

```
#define GLOBAL _declspec(allocate("GLOBAL_DATA"))
```

to allocate these global variables in a special area within the executable image (Fig. 3) [1]. At application initialization, the first node in the system becomes the primary copy for this region. All necessary mappings are established to other nodes as they are attached to the application. Thus, static global variables of arbitrary types can be shared among system nodes. This approach can be used in other operating systems. For example, in Linux the *__attribute__ ((section ("GLOBAL_DATA")))* has similar functionality.

Finally, *CableS* does not attempt to deal with the restrictions on the amount of memory that can be registered and pinned because these are issues that are better dealt at the NIC level (Table 1). However, this work is beyond the scope of this paper, which focuses at SVM library level issues.

2.2. Thread Management

In a distributed environment, threads of execution need to be started and administered on remote systems. For this purpose, *CableS* needs to maintain and manage global state that stores location and resource information about each thread in an application. *CableS* uses per application global

[1]Making this region part of the shared address space in NT is not straight forward, since the system does not seem to allow remapping of this area in the process virtual address space. For this reason we extend the VMMC driver to provide the necessary supporting functionality.

state, called the application control block (ACB). This state is updated by all nodes in the system via direct remote operations as well as notification handlers. *CableS* maintains the most up to date system information on the first node where the application starts (master node). To ensure consistency of the ACBs, updates are performed either by the master node through remote handler invocations or by node update regions in which the system guarantees that the node is the exclusive writer.

The thread management component of the *pthreads* library is hinged around thread creation. Thread creation in *CableS* involves one of three possible cases: (i) Create a thread on the local node. Local thread creation is equivalent to a call to the local OS to create a thread. (ii) Create a thread on a remote node that is not used by this application. This operation is called attaching a remote node to the application. When *CableS* needs to attach a new node to the application, the master node M creates a remote process on the new node N. Node N, starts executing the initialization sequence and performs all necessary mappings for the global shared memory that is already allocated on M. N then retrieves global state information from M including shared memory mappings and sends an initialization acknowledgment back to M. M broadcasts to all other nodes in the system that N exists and that they can establish their mappings with N. At the end of this phase, node N has been introduced into the system and can be used for remote thread creations. (iii) Create a thread on an already attached remote node.

The remaining thread management operations involve mostly state management, mainly, through direct reads and writes to global state in the ACB. For example, in the case of *pthread_join()*, a thread waits until the ACB indicates that the particular thread being waited for has completed its execution.

Most traditional SVM systems create one thread per processor; *CableS* allows multiple threads per processor. These threads are scheduled by the local OS and compete for global system resources. Threads can be terminated at any time via a cancel mechanism, or can terminate by completing execution. *CableS* provides mechanisms to terminate threads, and to dynamically detach a node when there are no longer any threads remaining on the node.

2.3. Synchronization Support

The *pthreads* API provides two synchronization constructs: *mutexes* and *conditions.* Current SVM APIs that mostly target compute-bound parallel applications provide two other synchronization primitives, locks and barriers. Since mutexes and locks are very similar, we use the underlying SVM lock mechanism to provide mutexes in the *pthreads* API. The *pthreads* condition is a synchronization construct in which a thread waits until another thread sends a signal. Mutexes and conditions can be implemented either by spinning on a flag or by suspending the thread on an OS event. Although implementations that use spinning consume processor cycles, they are more common in parallel systems to reduce wake–up latency. Our implementation of *pthreads* mutexes and conditional waits uses spinning, when there is fewer threads per processor in a node, and switches to locks that spin for a specified time and then locally block [22].

Finally, global synchronization (barriers) can be implemented in *pthreads* with mutexes (or conditions). However, to support legacy parallel applications efficiently *CableS* extends the *pthreads* API to support a barrier operation *pthread_barrier(number_of_threads)*.

2.4. 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 4 shows an example of a *CableS* program. To run any *pthreads* program on *CableS*, the following modifications are required:

1. Add the *pthread_start()* and *pthread_end()* library calls.
2. Prefix all static variables that will be globally shared with the *GLOBAL* identifier.
3. Link with *CableS* library.

```
/* 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 4. The current programming model for programs written for CableS

3. Results

In this section we present three types of results: (i) We provide microbenchmarks to measure the overhead of basic system operations. (ii) We demonstrate that legacy *pthreads* programs written for traditional hardware shared memory multiprocessors, such as SMPs, can run with minor modifications on *CableS*. We use a public domain OpenMP compiler, OdinMP [8], which is written for SMPs and hardware cache–coherent DSMs, to translate existing OpenMP programs to *pthreads* programs and run them directly on *CableS*. (iii) We study the impact of *CableS* on parallel programs that have been optimized for DSM systems by implementing the M4 macros on top of *pthreads* and running most of the SPLASH-2 applications.

3.1. Experimental Platform

The specific system we use is a 32–processor cluster consisting of sixteen, 2-way PentiumPro SMP nodes interconnected with a Myrinet network. Each SMP is running WindowsNT. The nodes in the system are connected with a low–latency, high–bandwidth Myrinet SAN [7]. The software infrastructure in the system includes a custom communication layer and a highly optimized SVM system. The communication layer we use on top of Myrinet is a user-level communication layer, Virtual Memory Mapped Communication (VMMC) [2, 10]. VMMC provides both explicit, direct remote memory operations (reads and writes) and notification–based send primitives. The SVM protocol used is GeNIMA [20], which is a home-based, page-level SVM protocol. The consistency model in the protocol is Release Consistency [13]. GeNIMA provides an API based on the M4 macros, which are extensively used for writing shared memory applications in the scientific computing community.

Table 3 shows the cost of basic VMMC operations on our cluster. Noticeable, VMMC provides a one way, end–to–end latency of around 7.8μs, which is to our knowledge, among the best performing systems using a Myrinet interconnect.

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 MBytes/s
Maximum fetch bandwidth	125 MBytes/s
Notification	18μs

Table 3. 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.

3.2. Microbenchmarks

Table 4 shows the results from our microbenchmarking. 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.

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 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. The pthread_create() times show the cost of a remote create and the potential for pooling threads on nodes to save time.

Unlike the remote mutex cost, the local mutex cost refers to the case where the mutex was last locked/unlocked by a thread within the node and there is no communication involved. The first time 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. 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.

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.

Segment migration involves determining if a segment has an owner and taking ownership of the page on the first touch. 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

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. 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.

page fault processing cost for a segment that does not need to migrate, but the page information needs to be examined. This processing is usually small but depends on whether the segment information is locally cached.

3.3. Supporting Legacy Pthreads Applications

To demonstrate the versatility of *CableS* we use OdinMP to compile three SPLASH-2 applications that have been written for OpenMP: FFT, LU, and OCEAN. OdinMP is written for translating OpenMP programs for SMP and hardware cache–coherent DSM systems. We also use three publicly available *pthreads* programs: (i) Prime numbers (PN), which 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 element stage consists of a calculation. Table 5 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 the average cost 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, PIPE, and the OpenMP programs provide an indication of the average execution time of remote operations in *CableS*. We see that remote operations are about three orders of magnitude slower than local operations. With respect to synchronization operations, conditional waits and mutex lock operations include the cost of communication and the 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. Table 6 shows the speedups of the three OpenMP SPLASH-2 applications. These applications are written for 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 DSM systems. The next section examines this aspect.

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

Table 6. Speedups for the three SPLASH-2 OpenMP programs on 4, 8, and 16 processors.

3.4. SPLASH-2 Applications

To investigate the overhead that *CableS* introduces in applications that have been tuned for the shared memory abstraction we provide an implementation of the M4 macros on *CableS* and run a subset of the SPLASH-2 applications

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
OMP_FFT	•		•	•		•	•	1235	54	0.52	1382	0.146	1.1	12302
OMP_LU	•		•	•		•	•	1247	133	1	327	0.134	0.401	12412
OMP_OCEAN	•		•	•		•	•	1312	49	2	494	0.293	0.606	14222

Table 5. 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.

on two configurations: The original, optimized SVM system that we started from [20] and *CableS*. 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 new call for the purpose of this comparison.

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 [20]. Their characteristics and behavior have been studied in [31, 21]. 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 [31], which affect their performance and the impact on SMP nodes. Water [31] and Radix [5, 16], 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 5 shows the execution times of each application in both system configurations for 1, 4, 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 currently present in *CableS*. From the execution traces, the original system could not execute OCEAN with 32 processors, because of memory registration limits. However, *CableS*, with its memory extensions, was able to run OCEAN on 32 processors. RADIX and VOLREND are more interesting: Since *CableS* relies on remapping 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 granularity results in improper page placement for many pages and in high protocol and communication costs. Figure 6 shows the percent-

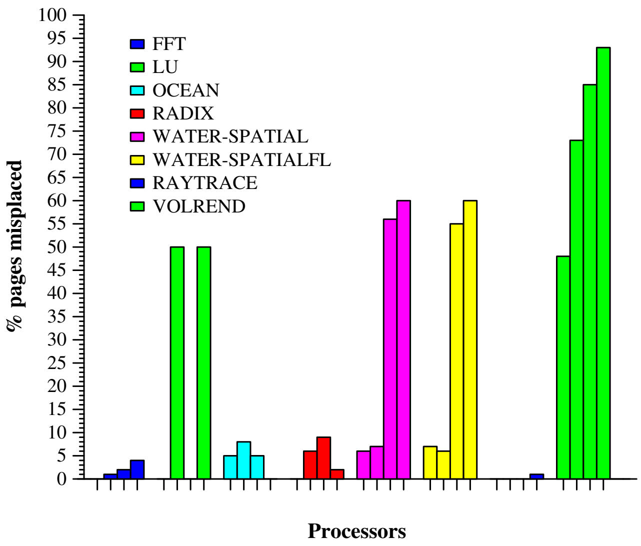
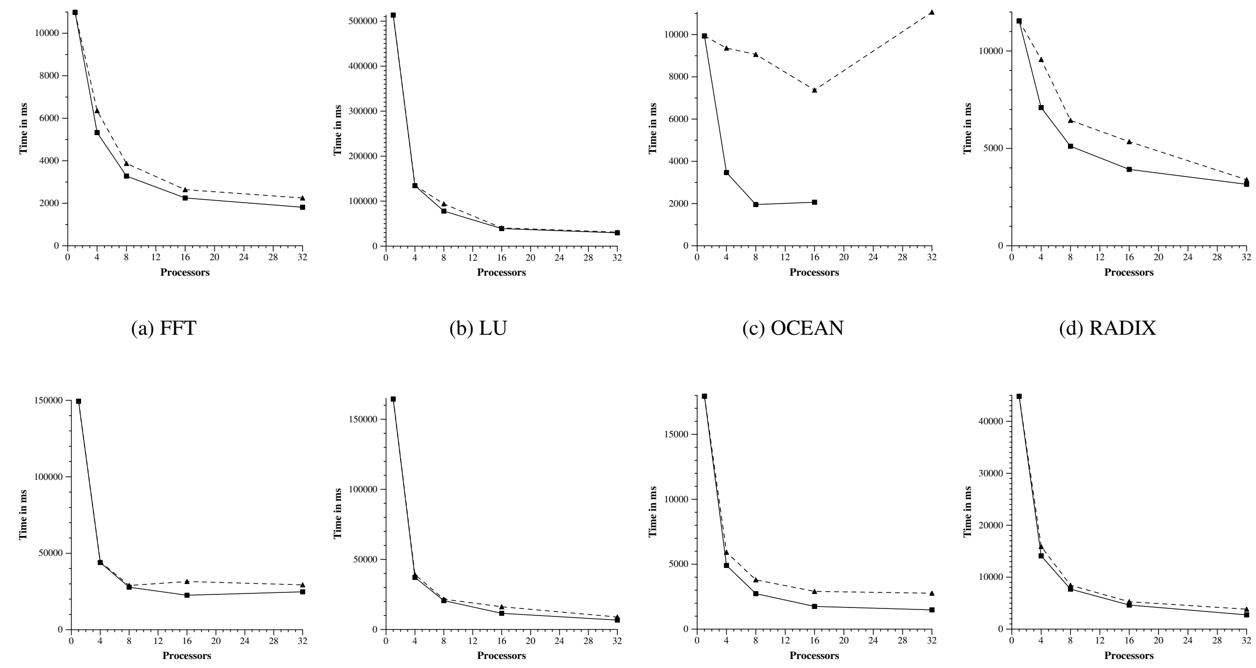


Figure 6. SPLASH-2 applications with their percentage of page misplacements for 4,8,16, and 32 processors

age 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



(a) FFT (b) LU (c) OCEAN (d) RADIX

(e) WATER-SPATIAL (f) WATER-SPAT-FL (g) VOLREND (h) RAYTRACE

Figure 5. SPLASH-2 M4 vs M4-pthread executions with 1, 4, 8, and 16 processors. Solid line is the M4 executions, and dashed line is M4-pthread executions

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. The execution time breakdowns are practically identical and are omitted for space reasons. 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.

4. Related Work

The *pthreads* standard is defined in [1]. *CableS* targets the implementation of a *pthreads* API on clusters of workstations. DSM-Threads [26] provides the same API on hardware cache–coherent DSM systems and discusses several implementation issues. *CableS*, instead, deals thoroughly with issues on modern SANs that support direct remote memory access. The authors in [12] 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 with providing dynamic thread and memory management. The Authors in [30] discuss home page migration issues in Cashmere. They examine protocol level extensions for migrating protocol pages among nodes. In our work, we investigate how SVM systems can be extended to support dynamic memory management. Shasta [27] is an instrumentation–based software shared memory system that was able to support challenging applications using the 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. There has also been some work on trying to eliminated registration limits at the NIC level. The authors in [9] 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 they do not address the related issues on the receive side. The authors in [4] 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.

Most other related work in the area has focused on the following four directions: (i) To improve the performance of SVM on clusters with SANs. There is a large body of work in this category [29, 23, 20, 32]. Our work relies on the experiences gained in this area and builds upon it to extending the functionality provided by today's clusters. (ii) To provide OpenMP implementations for clusters. Relatively little work has been done in this area. The authors in [24] provide an OpenMP implementation based on TreadMarks. They convert OpenMP directly into TreadMark system calls. In [28] the authors 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. (iii) 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 [25]. In many systems, this is the preferred API for multithreaded applications due to the portability advantages. (iv) Finally, 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. The authors in [3] 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. Our work is at a lower layer. For instance, a JVM written for the *pthreads* API, such as Kaffe [18] could be ported to our system.

5. Conclusions

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, especially in areas that exhibit dynamic behavior. We also perform a preliminary evaluation of basic system costs.

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. The rest of the overhead introduced by *CableS* is limited to the initialization and termination sections of these applications.

CableS is a first step towards enabling a wider range of new applications to run unmodified on clusters. To facilitate further work with publicly–available, server–type applications we are porting *CableS* to Linux. We are also considering supporting a complete single system image on top of clusters that includes file system and networking support to allow a wider range of applications that have been developed for SMPs to run on clusters. In this direction, we are planning to examine commercial workloads, such as the Apache WWW server, and the Kaffe JVM that are usually run on small–scale SMPs.

6. Acknowledgments

We would like to thank the reviewers of this paper for their valuable comments and insights. Also, we thankfully acknowledge the support of Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, Communications and Information Technology Ontario, and Nortel Networks.

References

[1] 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].

[2] J. S. A.Bilas, C Liao. 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.

[3] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A high performance cluster jvm presenting a pure single system image. In *ACM Java Grande 2000 Conference*, 2000.

[4] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 142–153, Apr. 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[8] C. Brunschen and M. Brorsson. Odinmp/ccp - a portable implementation of openmp for c. *The 1st European Workshop on OpenMP*, 1999.

[9] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. 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*, pages 193–203, San Jose, CA, Oct. 1998.

[10] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[11] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, Aug. 1997.

[12] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of the Second Merged Symp. IPPS/SPDP 1999)*, 1999.

[13] K. Gharachorloo, D. Lenoski, and et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[14] Giganet. Giganet cLAN family of products. http://www.emulex.com/products.html, 2001.

[15] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb. 1996.

[16] C. Holt, J. P. Singh, and J. Hennessy. Architectural and application bottlenecks in scalable DSM multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[17] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[18] T. T. Inc. Wherever you want to run java, kaffe is there.

[19] A. Itzkovitz and A. Schuster. Multiview and millipage - fine-grain sharing in page-based DSMs. In *Operating Systems Design and Implementation*, pages 215–228, 1999.

[20] D. Jiang, B. O'kelley, X. Yu, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of smps. In *The 13th ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[21] D. Jiang, H. Shan, and J. P. Singh. 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.

[22] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 41–55, Oct. 1991.

[23] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, Jan. 1994.

[24] H. Lu, Y. C. Hu, and W. Zwaenepoel. Openmp on networks of workstations. In *Proceedings Supercomputing*, 1998.

[25] F. Mueller. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, pages 29–41, Jan. 1993.

[26] F. Mueller. Distributed shared-memory threads: Dsm threads. *Workshop on Run-Time systems for Parallel Programming*, pages 31–40, April 1997.

[27] D. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Oct. 1997.

[28] A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent adaptive parallelism on nows using openmp. In *Principles Practice of Parallel Programming*, pages 96–106, 1999.

[29] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. 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.

[30] R. Stets, S. Dwarkadas, L. Kontothanassis, , U. Rencuzogullari, and M. L. Scott. 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.

[31] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1995.

[32] Y. Zhou, L. Iftode, and K. Li. 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.