THROOM — Running POSIX Multithreaded Binaries on a Cluster

Henrik Löf, Zoran Radović, and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden

Abstract

Most software distributed shared memory systems (SW-DSMs) lack industry standard interfaces that limit their applicability to a small set of shared-memory applications. In order to gain general acceptance, SW-DSMs should support the same look-and-feel of shared memory as hardware DSMs. This paper presents a runtime system concept that enables unmodified POSIX P1003.1c (Pthreads) compliant binaries to run transparently on clustered hardware. The key idea is to extend the single process model of multi-threading to a multi-process model where threads are distributed to processes executing in remote nodes. The distributed threads execute in a global shared address space made coherent by a fine-grain SW-DSM layer. We also present THROOM, a proof-of-concept implementation that runs unmodified Pthread binaries on a virtual cluster modeled as standard UNIX processes. THROOM runs on top of the DSZOOM fine-grain SW-DSM system with limited OS support.

1 Introduction

Clusters built from high-volume compute nodes, such as workstations, PCs, and small symmetric multiprocessors (SMPs), provide powerful platforms for executing large-scale parallel applications. Software distributed shared memory (SW-DSM) systems can create the illusion of a single shared memory across the entire cluster using a software run-time layer, attached between the application and the hardware. In spite of several successful implementation efforts [4, 10, 14, 16, 21], SW-DSM systems are still not widely used today. In most cases, this is due to the relatively poor and unpredictable performance demonstrated by the SW-DSM implementations. However, some recent SW-DSM systems have shown that this performance gap can be narrowed by removing the asynchronous protocol overhead [2, 14], and demonstrate a performance overhead of only 30-40 percent in comparison to hardware DSMs (HW-DSM) [14]. One obstacle for SW-DSMs is the fact that they often require special constructs and/or impose special programming restrictions in order to operate properly. Some SW-DSM systems further alienate themselves from HW-DSMs by relying heavily on very weak memory models in order to hide some of the false sharing created by their page-based coherence strategies. This often leads to large performance variations when comparing the performance of the same applications run on HW-DSMs. SW-DSMs should support the same look-and-feel of shared memory as the HW-DSMs. This includes support for POSIX threads running on some standard memory model and a performance footprint similar to that of HW-DSMs, i.e., the performance gap should remain approximately the same for most applications. The ultimate goal is that, binaries that run on HW-DSMs will also run on SW-DSMs, without manual modifications.

In this paper we present a new runtime system concept that allow POSIX threads (Pthreads) [7] applications to run on a non-coherent clustered architecture. Threads are distributed from their original process to other processes running on the same or other nodes of the cluster. By letting the distributed Pthreads access the original process' software context in a coherent way, we can create the illusion of a shared-memory multiprocessor. We show how to create a global shared address space and how to distribute Pthreads using a fine-grained SW-DSM and binary instrumentation techniques. We also show that this can be made transparent using library pre-loading functionality present in most UNIX dynamic linkers and program loaders.

2 DSZOOM — a Fine-Grained SW-DSM

Our initial implementation is based on the DSZOOM SW-DSM [14]. Each DSZOOM node can either be a uniprocessor, a SMP, or a CC-NUMA cluster. The node's hardware keeps coherence among its caches and its memory. The different cluster nodes run different kernel instances and do not share memory with each other in a hardware-coherent way. DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory in other nodes, similar to the remote `put/get` semantics found in the cluster version of the Scalable Coherent Interface (SCI), or the emerging InfiniBand interconnect proposal that supports `RDMA READ`/`WRITE` as well as the atomic operations `CmpSwap` and `FetchAdd` [8]. Another example is the Sun Fire (TM) Link interconnect hardware [20] that supports kernel bypass messaging via remote shared memory (RSM) interface, whereby shared memory regions on one machine can be mapped into the address space of another.

While traditional page-based SW-DSMs rely on TLB traps to detect coherence "violations," fine-grained SW-DSMs like Shasta [17], Blizzard-S [19], Sirocco-S [18], and DSZOOM [14] have to insert software coherence checks with executable editing. In DSZOOM, this is originally done by replacing each load and store that may reference shared data of the binary with a code *snippet* (short sequence of machine code). The binary instrumentation technique adds extra latency for each load or store operation to global data, independently if that data is locally available or not. In DSZOOM, the largest source of overhead comes from the in-line checks (ILC) for global loads and stores [14].

3 THROOM Overview

Most SW-DSM systems keep coherence inside a specified segment of the virtual address space. We call this segment global memory (G_MEM). In most SW-DSM implementations, the different nodes of the cluster all run some daemon or host process to maintain the G_MEM mappings and to deal with requests for coherency actions. In this paper, we use the term *user node* to refer to the cluster node in which the user executes the binary (the *user process*). All other nodes are called *remote nodes* and their daemon processes will be called *shadow processes*. This setup creates a *split-execution* system [22].

Transparency is achieved by using *library interposition* [22], which allow us to change the default behavior of a shared library call without recompiling the binary. Many operating systems implement the core system libraries such as `libc`, `libpthread`, and `libm` as shared libraries. Using interpositioning, we can catch a call to any shared library and redirect it to our own implementations. Original arguments can be altered and post or preprocessing of output can be applied. See Figure 1 for an example.

3.1 Distributing Threads

Threads are distributed by catching the `pthread_create()` call and copy the arguments to some shared scratch area of the address space. The call is then executed in a shadow process using the copied arguments. When the call returns in the shadow process, the output is written to the scratch area. The user process then reads the shadow output from the scratch area in the user

```
pthread_t pthread_self(void)
{
  static pthread_t (*func)();
  if( I_am_master ) {
    if(!func)
      func = (pthread_t(*)())dlsym(RTLD_NEXT, "pthread_self");
    return(func());
  }
  else {
    if(!func)
      func = (pthread_t(*)())dlsym(RTLD_NEXT, "pthread_self");
    return(REMOVE_NODE_ID(func(), _myid));
  }
}
```

Figure 1: Interposing agent example.

node and returns. The new distributed thread will start to execute in the shadow process, using arguments pointing to the context of its user process. A minimal requirement for the thread to execute correctly in a shadow process is that it must share the address space of the user process.

3.2 Creating a Global Shared Address Space

Code and global data are made accessible in a coherent way from all cluster nodes by copying memory containing code and global data of the application's virtual address space to the G_MEM and then divert accesses to the copy. This will make the application execute entirely in the global shared memory segment. To enable a multi-threaded application to run coherently, all global data referenced by threads have to reside in G_MEM. If we copy the `.text`, `.data`, and `.bss` segments to the G_MEM and modify the code to refer to these copies, we can move a thread to a shadow process and still access the global data.

The access diversion can be made transparent in several different ways. If the code is compiled as Position Independent Code (PIC), the Global Offset Table (GOT) can be modified so that the copy is referred to [23]. This is often not applicable, since most binaries are not compiled as PIC. The structure and placement of the GOT might also be hard to find at runtime. Another approach is to use binary instrumentation to change references to access the G_MEM instead of their original segments. It is natural to use this approach in THROOM, since the G_MEM is already made coherent by a fine-grain SW-DSM.

3.3 Cluster-Enabled Library Calls

Most application binaries use system calls and calls to shared libraries. If the arguments refer to thread-global data (call-by-reference), the access must be modified to use the G_MEM in order for modifications of the data to be coherent across the cluster. This can be done in at least two ways:

Instrument the library code This is unfortunately a difficult task, since library code normally is heavily optimized. This makes it hard for an instrumentation tool to rebuild the structure of the code and to produce a correctly instrumented binary.

Use library interposition A call that references potentially thread-global variables is caught and the references are modified in the interposing library. The referenced memory is validated to ensure that new copies of any invalidated data are used.

Instrumenting all library code is in principle, the best way to cluster-enable library calls. However, our instrumentation tool, EEL [11], was not able to instrument all of the libraries. Instead, we had to use the library interposition method. It enabled us to make cluster versions of system and/or library calls without recompiling or instrumenting the libraries. It also turned out to be of great use for debugging and reverse engineering. Using an interposed library, we need two primitives to make coherent accesses from and to the G_MEM:

`coherent_mem_store()` copies data from original text and data segments to G_MEM. Generates the coherence actions needed.

`coherent_mem_load()` loads data from G_MEM to the original text and data segments and resolves any invalidated copies.

An obvious disadvantage of this method is that we have to write interposing agents for many calls. Another disadvantage is the runtime overhead associated with data copying, especially for I/O operations. A better solution would be to generate the coherence actions on the original arguments before the call is made in the application binary [16]. This requires a very sophisticated instrumentation tool, which is outside the scope of this work. Some calls also need to be totally rewritten to work on a cluster. For example, the `malloc()` system call must allocate its memory in the G_MEM instead of using the standard heap.

3.4 THROOM in a Nutshell

To summarize, the following different steps need to be taken to transparently allow an unmodified POSIX binary to run on a cluster.

1. Threads need to be distributed to execute in several different processes on different OS kernels.
2. The distributed threads should reference shared data through a global shared memory segment made coherent by a fine-grain SW-DSM.
3. The initial `.text, .data`, and `.bss` segments need to be copied into the G_MEM and accesses to data in these segments need to be modified to hit the copy using binary instrumentation.
4. To make the whole system transparent, we implement it as a shared library to be interposed at program loading.
5. System calls or other shared library calls using pointers to global data must validate the referenced memory before any loads and stores are made.
6. Some calls such as `malloc()` and synchronization primitives must be made THROOM aware.

4 Implementation Details

We have implemented the THROOM system on a 2-node Sun WildFire prototype SMP cluster [5, 6]. The cluster is running a slightly modified version of Solaris 2.6 and the hardware is configured as a standard CC-NUMA architecture. Our cluster is built from two Sun Enterprise E6000 SMP machines, which we denote as cabinet 1 and cabinet 2. Processors in the different cabinets have been set up to access different node-private copies of the G_MEM, and the DSZOOM system is used to keep these copies coherent.

The runtime system is implemented as a shared library. A user sets the `LD_PRELOAD` environment variable to the path of the THROOM runtime library, and then executes the instrumented binary.

The DSZOOM address space is set up during initialization using the `.init` section and standard POSIX shared memory primitives. Control is then given to the application. The user process issues a `fork(2)` call to create a shadow process, which will inherit its parents mappings by the copy-on-write semantics of Solaris. The two processes are bound to the different cabinets using the WildFire first-touch memory initialization and the `pset_bind()` call. The home process then reads its own `/proc` file system to locate the `.text`, `.data`, and `.bss` segments and copies them to the G_MEM.

The shadow process waits on a process shared POSIX conditional variable to create remote threads for execution in the G_MEM. Parameters are passed through a shared memory mapping separated from the G_MEM. Since the remote thread is created in another process, thread IDs can no longer be guaranteed to be unique. To fix this, the remote node ID is copied into the most significant eight bits of the thread type, which in the Solaris 2.6 implementation is an unsigned integer. Similar techniques are used for other Pthread calls.

Since the Sun WildFire is a single-system-image cluster, we can simply use POSIX process shared synchronization primitives. Synchronization primitives called from within the application will not work, since we have not been able to instrument the Solaris Pthread library using EEL. Instead, we allocate process shared synchronization variables in a separate shared memory area before the application starts. When an application initializes a synchronization variable, we catch the call and redirect the variable (by switching addresses) to one of our pre-prepared variables. This is done by storing the address of the interposed lock in a field of the `pthread_mutex_t` data structure.

4.1 Binary Instrumentation

The binary instrumentation process must be compliant with all Sparc ABI specifications, and especially with global register usage. The DSZOOM engine requires two free global registers, at the insertion point during the instrumentation phase, to pass parameters to the coherence routines in an efficient way from in-line code snippets. On Sparc V8 (32-bit) and Sparc V8plus (64-bit) there are three global thread-*private* registers that are saved/restored during the thread-switching by the Solaris system libraries: %g2, %g3, and %g4; all other global registers are thread-*global* and are not saved during the switch. The thread-private registers are also called for *application registers*. On Sparc V9 (64-bit), on the other hand, only %g2 and %g3 are application registers, and the %g4 register is free for general use and is volatile across function calls together with %g1 and %g5. On all targets, registers %g6 and %g7 are reserved for system software and are not used during the binary modification process (in fact, Solaris' Pthread library uses one of those system registers itself).

Currently, DSZOOM also need a fast mechanism to lookup the node id for every running thread, which led us to reserve the global register %g2 for that task (this restriction can easily be removed if we use another convention for MTAG lookups[1] that is not dependent on node id). This is also why our target architecture for this proof-of-concept implementation is Sparc V8 and/or V8plus with three thread-private application registers. For simplicity reasons, we only instrument binaries without application-specific registers.[2] We have noticed that our test binaries are only about 1% slower if application registers are not used by the compilers.

The THROOM runtime system only imposes minor modifications to the original DSZOOM load and store snippets to divert the static data and access the `G_MEM` area. At most four additional machine instructions were enough compared to the original snippets to perform that task. The

[1] See [14] for more details about MTAGs.

[2] Sun's compilers can be instructed to reserve application registers by using the `-xarch=no%appl` compiler flag. On GNU's gcc, this is done with the `-mno-app-regs` flag.

```
  1: add          %o5, %g0, %ADDR_REG
  2: ld           [%ADDR_REG], %f7
  3: fcmps        %fcc1, %f7, %f7
  4: fbe,pt       %fcc1, hit

     ... call coherence routine ...

hit:
```

Figure 2: Floating-point load snippet example for the original DSZOOM system.

```
  1: add          %o5, %g0, %ADDR_REG

  2: srl          %ADDR_REG, 31, %g3
  3: brnz,pn %g3, L6
  4: sethi        %hi(0x80000000), %g4
  5: add          %ADDR_REG, %g4, %ADDR_REG

 L6: ld           [%ADDR_REG], %f7
  7: fcmps        %fcc1, %f7, %f7
  8: fbe,pt       %fcc1, hit

    ... call coherence routine ...

hit:
```

Figure 3: Floating-point load snippet example with support for the THROOM runtime system.

original DSZOOM floating-point load snippet is shown in Figure 2. The instrumented instruction in this example is `ld [%o5],%f7`. The efficient access control check [18], is performed on lines 3 and 4. In this example, the range check is performed inside the coherence routine to minimize the code expansion with this in-line snippet.

If we are unable to determine the effective address of the instrumented load during the binary modification phase, we should use the worst case THROOM snippet shown in Figure 3. The THROOM-related machine instructions are shown in lines 2 to 5. Lines 2 and 3 perform a simple range check to handle static data accesses. If this range check evaluates as true, i.e., the effective address of this load is below the `G_MEM` starting point, the same load will be performed at the %ADDR_REG + `0x80000000`. If this particular load can be classified as static by a more elaborate analysis of the binary, lines 2 and 3 can be eliminated and the G_MEM offset can be added directly. This optimization is currently not implemented in THROOM.

4.2 Modified System and Library Calls

The `malloc()` call is altered to allocate memory in G_MEM. In order to run our benchmark suite, a number of system and library calls had to be modified (for example, `getopt`, `gettimeofday`, `gets`, `fgetc`, `scanf`, `fscanf`, `sscanf`, etc.). The calls were identified by examining the undefined symbols of the binaries and checking, using the man-pages, whether a specific call could reference static data or not.

Program	Problem size, Iterations	Replaced Loads [%]	Replaced Stores [%]
FFT	1 048 576 points (48.1 MB)	44.6 (19.0)	32.8 (16.5)
LU-c	1024×1024, block 16 (8.0 MB)	48.3 (15.5)	23.0 (9.4)
LU-nc	1024×1024, block 16 (8.0 MB)	49.2 (16.7)	27.7 (11.1)
Radix	4 194 304 items (36.5 MB)	54.4 (15.6)	31.4 (11.6)
Barnes	16 384 bodies (8.1 MB)	56.6 (23.8)	55.4 (31.1)
Ocean-c	514×514 (57.5 MB)	50.6 (27.0)	31.2 (23.9)
Ocean-nc	258×258 (22.9 MB)	51.0 (11.6)	39.0 (28.0)
Radiosity	room (29.4 MB)	41.1 (26.3)	35.1 (27.1)
Water-nsq	2197 mol., 2 steps (2.0 MB)	50.4 (13.4)	38.0 (16.2)
Water-sq	2197 mol., 2 steps (1.5 MB)	48.5 (15.7)	32.5 (13.9)

Table 1: Problem sizes and replacement ratios for the ten SPLASH-2 applications studied. Instrumented loads and stores are showed as a percentage of the total amount of load or store instructions. The number in parenthesis shows the replacement ratio for the DSZOOM SW-DSM without THROOM.

5 Performance Study

Ten SPLASH-2 applications [24] were compiled using the GCC v2.95.2 compiler without optimization (-O0).[3] A standard Pthread PARMACS macro implementation was employed.[4] To exclude the initialization time for the THROOM runtime system, timings are started at the beginning of the parallel phase. All timings have been performed on the 2-node Sun WildFire [5, 6] configured as a traditional CC-NUMA architecture. Each node has 16 UltraSPARC II processors running at 250 MHz. The access time to node-local memory is about 330 ns (*lmbench* latency [12]). Remote memory is accessed in about 1800 ns (lmbench latency).

In Table 1, we see that more instructions are replaced in the case of THROOM since *all* references to static data have to be instrumented. This large difference in replacement ratio compared to DSZOOM is explained by the fact that DSZOOM can exploit the PARMACS programming model and use *program slicing* to remove accesses to static data that are not shared.

Figures 4 and 5 show execution times in seconds for 8- and 16-processor runs for the following THROOM configurations:

THROOM_HO All threads are created in the home node. No threads are scheduled in the shadow process, which means that no remote accesses are generated.

THROOM_RO All threads are created in the remote node (the shadow process) except for the master thread which runs in the home node.

THROOM_RR The threads are scheduled over the two nodes in a round-robin fashion.

DSZOOM Used as reference. Aggressive slicing and snippet optimizations. Optimized for a two-node fork-exec native PARMACS environment, see [14].

6 Discussion, Conclusions, and Future Work

A study of Figures 4 and 5 reveals that the present implementation is slower than a state-of-the-art SW-DSM such as DSZOOM. In some cases, THROOM_HO is faster than DSZOOM, since in this

[3] The code is compiled without optimization to eliminate any delay slots, which EEL cannot handle correctly.

[4] `c.m4.pthreads.condvar_barrier`

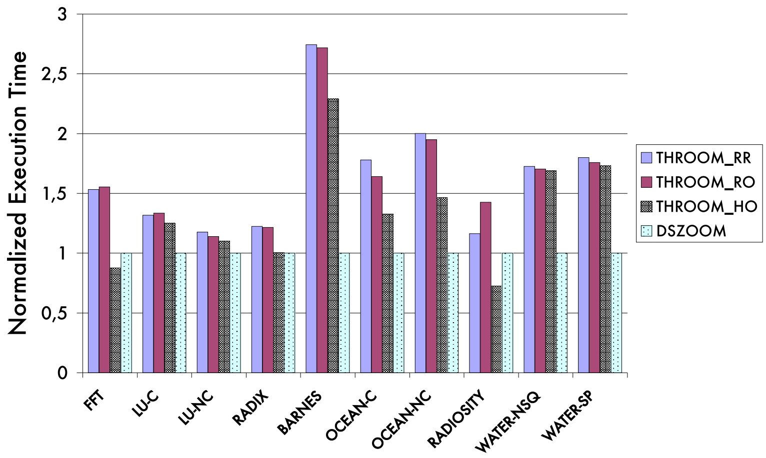

Figure 4: Runtime performance of the THROOM runtime system. A total of 8 processors were used.

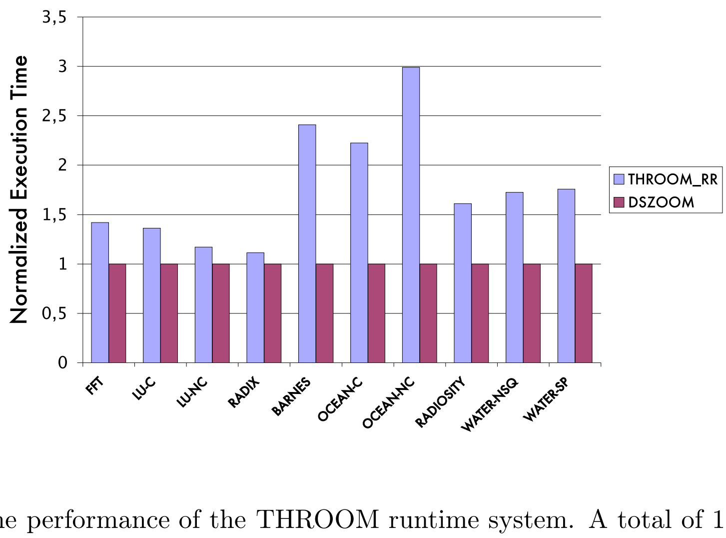

Figure 5: Runtime performance of the THROOM runtime system. A total of 16 processors were used.

case no remote accesses are generated as all activity is contained within a single node. The average runtime overhead compared to DSZOOM for THROOM_RR is 65% on 8 processors and 78% on 16 processors. In order to put these numbers into the context of total SW overhead, it should be noted that DSZOOM's overhead is 32% compared to a hardware-coherent implementation [14]. The most significant contribution to the high overhead when comparing DSZOOM to THROOM is the increased number of instrumentations needed to support the POSIX thread model. Another source of overhead is the rather inefficient implementation of synchronization primitives of the present THROOM implementation, see Section 4.

In conclusion, we have showed that it is *possible* to extend a single process address space to a multi-process model using fine-grain instrumentation techniques. Even though the current THROOM implementation relies on some of the WildFire's single system image properties, we are convinced that the THROOM concept can be extended to a pure cluster model. On a true cluster, additional issues need to be addressed. The shadow processes need to be set up without relying on a single system image cluster, possibly using a standard MPI runtime system. Synchronization needs to be handled more efficiently (see [15]), and we need to create more complete and more efficient support for I/O and other library calls.

An intermediate step could be to layer THROOM below a standard OpenMP runtime system [3]. Most OpenMP implementations use *subroutine outlining* to interface to the underlying OS, which means that `PRIVATE` variables are put on the stack.[5] Using OpenMP, we have more information about critical sections and shared data, which can be exploited to minimize the overhead associated with the increased number of instrumentation of static data.

THROOM will probably also benefit from many of the proposed optimizations to make SW-DSMs more efficient. Such as, better instrumentation tools, coherence protocol optimizations and new faster hardware.

7 Related work

To our knowledge, no SW-DSM system has yet been built that enables transparent and efficient execution of an unmodified POSIX binary. The Shasta system [16], come closest to our work and this system has showed that it is possible to run an Oracle database system on a cluster using a fine-grain SW-DSM technique. Shasta has solved the OS functionality issues in a similar way as is done in THROOM although they support a larger set of system calls and process distribution. THROOM differs from Shasta in that it relies on the DSZOOM SW-DSM, which does not suffer from the synchronous protocol processing. THROOM also supports thread distribution and a thread-enabled address space. Shasta motivates the lack of multi-threading support by claiming that the overhead associated with access checks lead to lower performance [16].

Another system announced recently is CableS [9] built on the GeNIMA page-based SW-DSM [2]. This system support a large set of system calls, but they have not been able to achieve binary transparency. Some source code modifications must be made and the code must be recompiled for the system to operate. Another work related to THROOM is the OpenMP interface [1] to the TreadMarks page-based SW-DSM [10], where a compiler front-end translates the OpenMP pragmas into TreadMark fork-join style primitives. The DSM-Threads system [13] provide a page-based SW-DSM interface similar to the Pthreads standard without binary transparency.

The library interposition techniques and a split-execution system is covered by the Multiple Bypass system [22]. Some work by Welsh et al. on how to create a global address space can be found on the web [23].

[5] Currently, THROOM views the stack as thread private which is not in compliance with POSIX. We could also instrument stack accesses, which would probably generate a large amount of overhead.

References

[1] T. Gross A. Scherer, H. Lu and W. Zwaenepoel. Transparent Adaptive Parallelism on NOWs using OpenMP. In *Principles Practice of Parallel Programming*, 1999.

[2] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, May 1999.

[3] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared Memory Programming. *IEEE Computational Science and Engineering*, 5(1), Jan/Mar 1998.

[4] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 260–269, January 1999.

[5] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach 3:rd edition*. Morgan Kaufman, 2003.

[7] IEEE Std 1003.1-1996, ISO/IEC 9945-1. *Portable Operating System Interface (POSIX)–Part1: System Application Programming Interface (API) [C Language]*, 1996.

[8] InfiniBand(SM) Trade Association. *InfiniBand Architecture Specification, Release 1.0*, oct 2000. Available from: `http://www.infinibandta.org`.

[9] P. Jamieson and A. Bilas. CableS: Thread Control and Memory Mangement Extentions for Shared Virtual Memory Clusters. In *Proceedings of the 8th International Symposium on High-Performance Computer Architeture (HPCA-8)*, February 2002.

[10] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[11] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[12] L. W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, January 1996.

[13] Frank Mueller. Distributed Shared-Memory Threads: DSM-Threads. In *Proceedings of the Workshop on Run-Time Systems for Parallel Programming*, 1997.

[14] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of Supercomputing 2001*, November 2001.

[15] Z. Radović and E. Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of Supercomputing 2002*, November 2002.

[16] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France*, October 1997.

[17] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[18] I. Schoinas, B. Falsafi, M. Hill, J. R. Larus, and D. A. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[19] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297–306, October 1994.

[20] S. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire(TM) Link Interconnect. In *Proceedings of the IEEE/ACM SC2002 Conference*, November 2002.

[21] 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 *Proceedings of the 16th ACM Symposium on Operating System Principle*, October 1997.

[22] D. Thain and M. Livny. Multiple Bypass: Interposition Agents for Distributed Computing. *Cluster Computing*, 4:39–47, 2001.

[23] M. Welsh and J. DeCristofano. Shared-Memory Multiprocessor Support for Split-C. `http://www.cs.berkeley.edu/~mdw/projects/split-c-smp/design/split-c-smp%.html`, 1995.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.