Kernel Level Distributed Shared Memory over Gigabit Ethernet

John-Paul Fryckman

University of California at San Diego
La Jolla, CA 92093

15 June 2000

Abstract

The feasibility of distributed shared memory (DSM) greatly depends upon the implementation and the underlying physical transport layer. Gigabit network technology enables the former to make comparable performance advancements in terms of the memory speed. Further, one has to rethink the implementation due to the fact that protocols and non-network latency have a tremendous impact on overall performance. This paper places an upper bound on raw performance based on a kernel level DSM. Secondly, it describes the benefits of thread migration instead of page migration.

1 Introduction

Many algorithms lend themselves to a straightforward shared memory programming model. Driven by this concept, distributed shared memory was developed to extend the capabilities of distributed computing environments. As we see an explosion in distributed clusters, new applications are being developed or ported to this style of computing. A shared memory model has the potential of decreasing the development time as well as improving the performance of the application. DSM literature will be reviewed, followed by the design principles of this project. The kernel level implementation will be discussed along with the predicted performance.
2 DSM Review

Many papers exist on distributed shared memory. Of these, Munin, Ivy, Mirage, and Treadmarks display very novel concepts that are integrated into this implementation. Nitzberg and Lo write an excellent short summary of all the major DSM papers up until the early 1990’s [7].

2.1 Ivy

Li and Hudak formally researched the memory coherence problem in [6]. When anyone implements a distributed shared memory system, one has to worry about memory coherence whether its a full blown model or minimally provides the very basic consistency that makes it possible for higher levels to control their execution. Their paper went on to address how the performance of memory coherence models depends on the memory granularity and the coherence scheme. The former depends on hardware issues as well as network latency. A coherence scheme has to deal with page synchronization and ownership. Furthermore, one can deal with these two issues via a centralized scheme or a distributed one. However a centralized scheme creates bottlenecks, but provides for a simple implementation. A distributed method allows for less contention, but adds complexity to the overall system—

one now has to locate pages. Li was the principal designer of Ivy which incorporated this work. Ivy was a software implementation that had strict coherence and invalidate on write with a distributed page manager [5].

2.2 Munin

Munin was yet another DSM project that focused on the different types of data used in shared memory programs with an adaptive software cache management based on these objects. The Munin team identified several such objects: write-once, write-many, producer-consumer, private, migratory, result, read-mostly, and synchronization. The interested reader is directed to their paper for examples [1]. They realized that one simply cannot treat all of these objects the same way. They allow the user to give suggestions on the data types and synchronization. Munin implemented software release consistency(RC) and delayed update queues, in particular it studied lazy RC against eager RC. Munin also introduced a multiple write protocol that improved the false sharing problem. For a complete discussion on the implementation and performance of Munin, read [2].

2.3 Mirage

A kernel level implementation was first to be done by Mirage. Mirage offered
a strict coherence model and invalidate on write as well as a time window for coherence. They realized they could gain greater performance by stripping overhead and directly writing DSM into the kernel. Their approached DSM with cautious optimism, but they had hope in DSM’s future with faster processors and Ethernet devices [3].

2.4 Treadmarks

DSM research slowly dwindled in the beginning of the 1990’s. In the middle of the decade, a group of researchers from Rice introduced TreadMarks. The software package was implemented above the kernel, but focused on reducing the cost of communication. They used a lazy release consistency model as well as lazy diff creation and multiple-writer protocols. They proved to be very successful, but were again limited in the fact that it suffered from too much overhead. [4]

3 Design Principles

Given a loosely or tightly coupled network of workstation, one can effectively aggregate the memory of these machines to form a single virtual memory space for a process or a collection of lightweight threads. Over a loosely coupled network one has to worry about security and fault-tolerance, however they persist in lesser extent in a tightly coupled network, the reliability and performance are noticeably higher. The network design of a tightly coupled network is out of the scope of this paper, but one can go about creating a toroidal mesh for multi direction communication.

The goal of this project as well as the overhaul goal of DSM is to provide the user with a shared memory paradigm with minor modification to the programming environment. This has to be balanced with performance and the demand placed on the underlying system to adaptive to the code’s data and processor usage. Therefore, it is reasonable to ask the user to include special synchronization commands to minimize the amount of communication in conducting the consistency protocol. This release consistency model has proven itself to be effective in the forementioned papers.

A distributed page management system is the most effective method in removing the bottleneck of a centralized system. This adds complexity, but increases performance. However, when page ownership is allowed to change, searching for a page can be, in the worse case, O(N), where N is the number of machines. This happens when you have multiple writers and readers. The method adapted in the later papers fix a page locate and require diffs to be generated in the synchronization pro-
cess for a process requiring write access. If a majority of the machines require write access, a tremendous amount of overhead has to take place in the updating the original page. A diff is made by comparing the page with an original copy and marking recording the differences.

A solution to this problem is to have the actual thread migrate to the data location thus reducing the need for page migration and diff generation. A typical page is 4 Kilobytes, where as a typical process state requires 1 Kilobyte. This four to one ratio merits a detailed study of a thread migration approach. Further benefits include re-use of TLB entries on the machine holding the page to be written on as well as reducing the computation for diff generation and combining and easing the synchronization process. However it requires intelligents on the part of the DSM subsystem. The main problem with this approach is that it will cause threads to thrash between machines if it writes to multiple pages stored on different machines. The second problem is that it would cause "machine" hot-spots, e.g., a global reduction such as a dot product. Both of these problems are remedied by profiling the code beforehand and an intelligent distributed manager that keeps track of a thread’s memory usage and makes recommendations on where to store pages. Hot spots can be reduced by distributing the memory.

As for example, High Performance Fortran has directives to distribute blocks regularly, cyclic, or mise arrays. Furthermore, the compiler could generate multilevel code for a global reduction that selects various machines to reduce to and from them to reduce to a final machine.

The last key issue is that of overhead. Software level implementations require more overhead than if one directly inserted the code into the OS’s page fault handler. Active Messages served as a guide to this DSM’s philosophy, see [8]. Once a fault is made, the message is sent immediately. On the receiver, once interrupted by the Ethernet device, the request is processed and the data is then copied into the outgoing Tx buffer.

4 Kernel Implementation

The main goal is to eliminate all but the absolute necessary amount of overhead in implementing DSM. Inserting code directly into the fault handler is critical to reducing latency. When a page faults for either read or write access, the page fault handler is invoked. Once it is determined that it is a DSM page and the requisite page is located on a remote machine, a command packet is generated and loaded into the Tx ring of
the Ethernet device. The process then sleeps until the corresponding packet is received with an error code or the actually page of data.

The remote machine, once interrupted by the device, immediately reads through the Rx Ring. Upon finding a DSM packet in the ring, it executes the packet’s command. In most cases it is for a page copy. The page is copied unto a packet waiting on the Tx ring with the proper DSM packet protocol. In effect, once interrupted, the process will not cease to execute until it has sent the packet. Since now DSM is on the critical path, one has to ensure that it uses a minimum number of instructions and that it avoids deadlock.

4.1 Kernel DSM Code

The first function to be called in a page fault is do_page_fault. Once it has been determined that the faulting address is valid, it calls handle_mm_fault. This function allocated a page table entry and passes it to handle_pte_fault. It determines how to handle the page, i.e. does the page need to be physical swapped in, is it a copy on write page, etc. The DSM code is inserted here. It accesses the VM struct associated with the page, if the page is marked DSM the function do_dsm_page is called. The DSM struct (which is global and shared) is accessed to determine what machine has the memory.

4.2 Code Progress

After many weeks of code development, the author arrived at a page alignment problem. In order to test the code, two processes were executed on the same machine both having identical memory layout save for different values used to fill the memory arrays. One was made to fault. Once the process accessed the paged marked to fault, it triggered the DSM mechanism which accessed the corresponding page entry in the other process and copied back into the faulting process. Upon analysis of the resulting memory dump, it was found that exactly one page of the dump was filled with random numbers and not the dummy variable used to fill the other process’s memory.

5 Performance Tests

In order to test the maximum throughput of the system, one has to generate enough page faults to saturate the network. A process is started and a module is loaded that clears a range of page table entries in the process’s memory. The process is now allowed to trigger the page faults. Timing data is collected by placing a call to the system’s clock before and after the calls that generate the page faults. How-
ever the timing would sum over both compute and idle time. The remedy is
to have multiple runnable threads that
cause page faults. This will establish
an upper bound on performance.

Real world performance testing has
to cover a broad range of applications. The first set would include
standard mathematic operations like
matrix multiplies, dot products, and
n-dimensional finite difference calculations. The second set includes more
demanding data read/write patterns.
Among these are fast Fourier transforms, sorting, and the traveling salesman problem.

Prior tests have shown that the max-
imum transmission rate for a gigabit
Ethernet card is 97 MBps for a 32-
bit 33Mhz PCI bus on a Pentium III machine and 120 MBps on a 32-bit
66Mhz PCI bus on a like machine. The
theoretical maximum is 1 Gbps with
the cards manufactured by Packet En-
gines using the Hamachi chipset and a
driver written by Donald Becker and
optimized for high performance by the
High Energy Research Group at the University of California at San Diego.
On Linux, the page size is set to 4096
Bytes. Therefore, the predicted upper
bound for the transmission of pages is
3.4 μs and 4.2 μs respectively. That
translates into nearly 30,000 pages per
second. Further more on dual bus sys-
tems, connecting one card per bus, the
Group saw comparable in bound and
out bound performance at the rates
given above. One can easily imagine
an affordable toroidal communication
mesh based on dual bus systems.

6 Conclusion

Gigabit Ethernet breathes new life into
distributed shared memory. The Eth-
ernet card used here in this paper is
one to two orders of magnitude from
most of the early DSM papers. Also,
at the rates quoted above it is cer-
tainly faster than typical SCSI rates
(2-3 times). Furthermore, in order to
operate at gigabit speeds and main-
tain high throughput, DSM has to be
implemented in the kernel. Faster
turnaround time on requests means a
higher throughput. Release consistency
has shown it self to be crucial in im-
proving performance and will be pro-
vided in the first software release.

Thread migration is yet a rather un-
explored field when coupled to a strong
DSM system. Lightweight threads only
use on the order of a kilobyte of space to
represent themselves. Hence, when one
considers the cost of moving a page or a
process, it is clear which one costs less.
However, thread thrashing becomes a
problem, but can be mitigated by vari-
ous techniques both at compile and
run-time. As point out, high level lan-
guages can provide keywords for data
alignment as well as identify the data
type as described in the Munin work. Future work includes the actually development of the algorithm to handle both page and thread migration based on usage patterns.

References


