Distributed Shared Memory

Jason Hoch, Mike Rosensweig, Sashko Stubailo

December 10, 2012

1 Introduction

For our project we implemented a basic Distributed Shared Memory (DSM) system as a user space library on top of Linux. Using DSM, multiple PCs can work on the same problem simultaneously, and the system makes sure that the memory stays in sync as if all the CPUs were on one machine simply running in separate threads. The CPUs connected to the system share a virtual address space even though they do not share any physical memory. To the CPUs it looks like they are sharing the exact same memory, and the system must make sure that accesses to the same area of virtual memory will contain the same value across every CPU. The different CPUs must communicate with each other to ensure the consistency of the memory accesses so the individual CPUs can behave as if they share a physical address space as well.

2 Implementation

For our implementation of these, we used the SIGSEGV library in C, which allows the creation of page fault handlers in user mode Linux. We defined a region of virtual memory to be the DSM region for all the processes. This region is flexible and can be defined at compile time to be any number of pages that are accessible to user space and not already in use when the `dsm_start` is called. When a shared page is mapped, it is mapped into all the CPUs virtual address space as read only, and is initialized to have a value of zero. Reads will behave

as normal, but when a write occurs, there will be a page fault. The core of the DSM functionality is contained in handling faults such as this, and in the interprocess communication to bring the system back up to speed.

When a page fault occurs on a write attempt, the handler will make that page of memory writeable in the process that attempted the write. It will also send a signal to all the other CPUs telling them that any data they have stored in that page is now invalid. Upon receiving this message, each core that did not initiate the write will mark the page in their virtual address space as neither readable nor writeable using `mprotect`. Execution in each core will then resume as normal when the handler returns. If a read is attempted on a page that has been marked as invalid, it will fault again. If we fault on a read this means we must fetch the correct value from the CPU who has the updated copy of the page. The handler will ask the proper CPU for the contents of the page, map them properly into the faulting CPUs address space, mark the page read-only in that address space, return to the location of the fault, and resume execution normally.

In order to keep track of which CPU holds the correct copy of each shared page we use a distributed method of keeping track of the pages. Each core is assigned a fraction of the available shared pages ($\frac{1}{N}$ for an N CPU system) according to a very simple function: CPU i is responsible for page j if and only if $j \bmod N = i$. This method distributes the load across all the CPUs assuming the likelihood of a particular page faulting is the same for all pages. On a read fault the faulting CPU asks the page's owner for the correct page, and the owner will fetch the page and return it. We used locking at a page granularity to ensure there were no race conditions with concurrent writes or a concurrent write and read on the same memory.

3 Testing

To test the basic functionality of this library we made a very simply program that connected to our DSM system with 4 CPUs and 10 shared pages of memory. We chose one arbitrary memory address in those pages and wrote a value to it. Then in the other processes we attempt to read this value and make sure that it is the desired value as set by the writing process, to illustrate that memory was properly being shared across all of the CPUs. Due to time constraints we were not able to test our system with a more complicated distributed algorithm, but this case is sufficient to prove our memory is being shared properly.

4 Analysis and Conclusions

The performance under this implementation as expected is quite poor. The socket calls between process slow down the execution quite a lot, since a full page needs to be sent each time, and other accesses must wait for the lock to begin trying to access the memory. As suggested in the Treadmarks paper, some ways to increase speedup would be release consistency or lazy diff creation, both of which would drastically reduce the number of socket calls and the amount of data sent over the socket. Additional optimizations that would help would be improving our locking so that is either finer grained or more scalable, in order to avoid bottlenecks on locking pages between CPUs.