Object Migration for Distributed-Memory Multiprocessors

Anthony D. Joseph and Eric A. Brewer*
Parallel Software Group
{adj,brewer}@lcs.mit.edu

1 Migration in Multiprocessors

Process migration has been used to improve load balancing, overall performance, and availability [Dou90]. The vast majority of work in this area involves the migration of Unix processes in distributed systems. This paper examines the problems of thread and object migration in homogeneous multiprocessors and it introduces fix-up blocks, a novel mechanism for providing object migration with a near-zero cost when objects are not migrating.

The primary goal of a migration mechanism is minimal impact on the performance of objects that do not migrate. Our approach, by achieving this goal, provides an attractive mechanism to language and run-time system designers.

A secondary goal in our design is the portability of our ideas. Thus, we make no assumptions about the underlying hardware. However, when available, we will exploit hardware support for a global address space with caching, virtual memory, tags, or full/empty bits.

We expect migration to be a useful tool for improving locality within a parallel program, eliminating the latency of remote accesses, and reducing network bandwidth requirements. Object migration is also useful for moving objects, since it provides a simple mechanism for moving data without having to marshal it explicitly. For example, if two objects are competing for resources on the same processor, migration could be used to move one of them to an idle processor.

2 Migrating Active Objects

Migrating idle objects is a relatively simple problem involving moving the object's data and ensuring that all references to the object remain valid. Many of the references to the object will be local addresses, mostly for performance reasons. These local addresses are invalidated by the object's migration and must be located and updated. We use a technique based upon a combination of residence checks and globally unique *object identifiers*. The techniques are well established and have been implemented in Emerald [Jul88]. Unlike Emerald, we will (essentially) eliminate the residence check on architectures with tag bits by causing a trap on access to objects that are no longer local.

The addition of active threads severely complicates migration; in particular a method must be able handle the migration of its object in the middle of the code, not just at the beginning or end.

We modify Emerald's *Object-Frame Invariant*: the frame for an executing method must be in the same address space as the method's object. This invariant allows the code of the method to access the object using pointers and without using residence checks. Without this invariant, the normal-case performance would deteriorate severely.

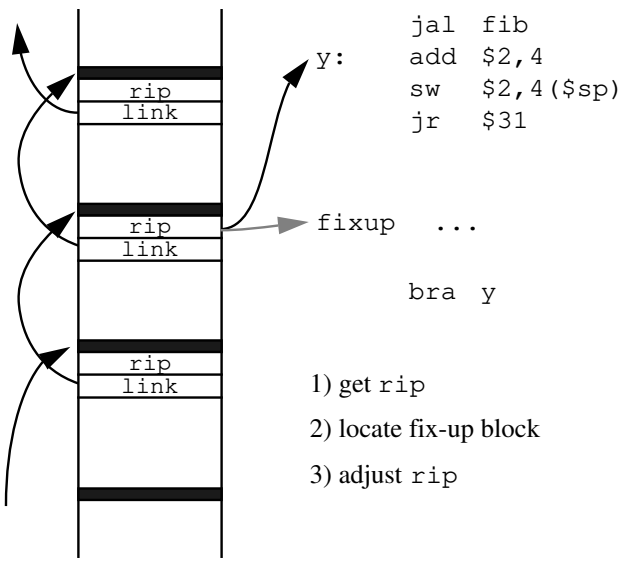

Figure 1: This figure depicts a portion of a stack with the `rip` for the middle activation record being updated to point to the corresponding fix-up block.

3 Migration Points

We introduce a novel invariant: An object cannot migrate unless all of its threads are at *migration points*. A migration point is simply a point in the code at which the object is allowed to migrate. This invariant improves performance because code can assume that its object will not migrate *between* migration points, so it can use pointers arbitrarily in these regions. Such pointers become invalid *across* migration points if the object migrates.

An implication of this invariant is that migration may be delayed. To prevent unbounded delay, we ensure that migration points are relatively frequent. For example, all invocations and synchronization points are migration points.

In general, crossing a migration point requires verifying that the object has not migrated, and migrating the frame if it has moved. The challenging part of migration points is reducing the cost of crossing one when the object does not migrate.

4 Frame Patching

We propose a mechanism called *frame patching*. The first point to note is that when an object (locally) decides to migrate, all other frames on the processor are suspended, since only one frame runs at a time. Each frame has a saved program counter, called its *return instruction pointer* or `rip`.

The basic idea behind frame patching, shown in Figure 1, is that when an object migrates, the `rip` for each of the proces-

*Anthony Joseph and Eric Brewer are members of Bill Weihl's Parallel Software Group. This work was supported in part by ARPA under Contract N00014-91-J-1698, by grants from IBM and AT&T, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U. S. government.

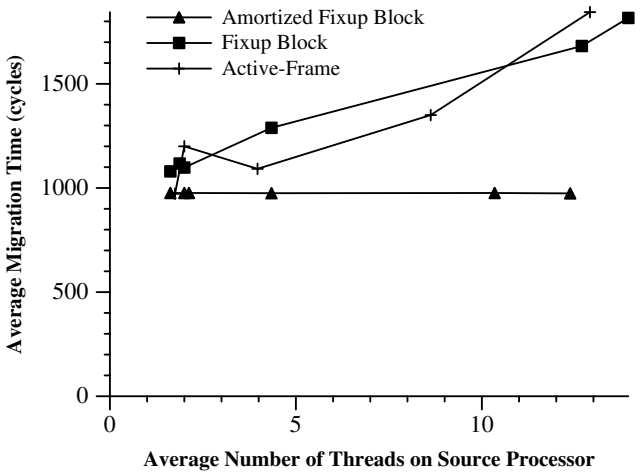

Figure 2: This figure shows the average time to migrate an object for a synthetic application.

sor's threads is updated so that it points to the corresponding *fix-up block* for that migration point. A fix-up block is a piece of compiler generated code that knows the layout of the frame and the register usage at the migration point. Since each frame must be at a migration point for the object to migrate, there is a valid fix-up block for every such `rip`. A hash table is used to optimize locating fix-up blocks. When the frame resumes execution, it will execute the fix-up block, which first installs a fix-up block for the parent frame. Next, if the object for the frame has moved, it moves the frame to the new host and translates any local addresses in saved registers or pointers within the frame to the correct addresses for the new host. Finally, the fix-up block jumps to the previous `rip` value, which causes the method to continue execution. Since all of the local addresses have been updated the code continues without a hitch.

Adding a fix-up block to every thread at migration time, makes the time to migrate an object a function of the number of threads on the processor. As an optimization, we have developed algorithms that perform frame patching for the top of each stack lazily, thus allowing the cost to be spread out over time and amortizing some of the cost across multiple migrations.

One version maintains two queues for runnable threads, a dirty queue of threads that need fix-up blocks before they execute and a clean queue of threads that are runnable as is. We use the clean queue during normal operation. When an object migrates, we move all of the current threads to the dirty queue (a constant time operation). When the end of the clean queue is reached, instead of looping back and running the first thread in the queue, a thread is removed for execution from the dirty queue and placed on the clean queue after having a fix-up block added.

5 Experiments

We have implemented the code required for migration points, fix-up blocks and frame patching for the PRELUDE language running on the PROTEUS simulator [Bre91, Wei91]. For comparison purposes, we also implemented an active-frame mechanism, similar to that used by Emerald [Jul88].

An active-frame mechanism maintains a linked list of active frames within the object; when the object moves, all of the associated frames are moved with it. This has the disadvantage that if the object moves frequently relative to how often the threads run, then the threads are being moved unnecessarily. Normal execution is affected by the need to maintain the linked list. Optimizations reduce the cost to 4-9 cycles per invocation return and our experiments found that 1-6% of application execution time is spent maintaining the list. The time to perform a context-switch increased by 4-7 cycles. In addition, the time to migrate an object becomes a function of the number of active frames within the object.

Figure 2 plots the average time to move an object as a function of the number of threads on the source processor for several executions of a synthetic workload consisting of several moving objects, each of which contained several threads accessing its state. Using the implementation of amortized fix-up blocks, the time to migrate an object is independent of the number of threads (and the number of active frames within the object). This is expected, since the addition of fix-up blocks is delayed until context-switch time. Using regular fix-up blocks, however, the time is dependent upon the number of threads, since this method adds fix-up blocks at migration time. Finally, for active-frame movement, the time is dependent upon the number of active threads within the object, which in this case is dependent upon the number of threads on the processor.

While the amount of time spent migrating an object is usually small in comparison to the total execution time (0.2-2% typical, 16.6% maximum), we found the overall performance of amortized fix-up blocks matches or exceeds that of active-frame movement. The primary reason for this is the elimination of the overhead of maintaining the object's linked list.

6 Conclusion

This work examines migration in distributed-memory multiprocessors and addresses different problems than those of Unix-based migration. The contribution of this work includes the development of migration points, fix-up blocks, and frame patching.

Our on-going experiments demonstrate that the invariants we define, combined with near zero-cost mechanisms for the common case that objects do not migrate, yield a run-time system that combines the resource-balancing benefits of migration with the high performance of direct access to the fields within an object.

Acknowledgments: Thanks to Bill Weihl, Wilson Hsieh, and Carl Waldspurger.

[Bre91] E. A. Brewer et al. PROTEUS*: A High-Performance Parallel-Architecture Simulator*. MIT/LCS/TR-516, September 1991.

[Dou90] F. Douglis. *Transparent Process Migration in the Sprite Operating System*. PhD thesis, TR UCB/CSD 90/598, University of California At Berkeley, September 1990.

[Jul88] Eric Jul et al. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Wei91] W. E. Weihl et al. PRELUDE*: A System for Portable Parallel Software*. MIT/LCS/TR-519, October 1991.