Experience with Process Migration in Sprite*

Fred Douglis
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
douglis@sprite.Berkeley.EDU

Abstract

This paper reports on experience with the Sprite process migration facility. Sprite provides transparent remote execution to support load sharing through the use of idle workstations. Process migration is used to reclaim workstations when their owners return. On Sun 3/75 workstations, the cost of selecting an idle host and invoking a remote process is about 400 milliseconds. This time is substantially greater than the cost of creating the same process locally, but it is much less than the typical execution time of programs that are run remotely, such as compilations and text formatting. The cost of migrating an active process is a function of the number of dirty pages it has, the number of file blocks that must be flushed from the host's file cache, and the number of open files it has. This time ranges from 110 milliseconds to migrate a small process with no open files, to several seconds to migrate a process with many dirty pages and file blocks and several open files. Remote execution has been used regularly for approximately 9 months to perform compilations in parallel. I draw conclusions about the usefulness of remote execution for parallel compilation, and I present lessons we learned about process migration and system building in general.

1 Introduction

By executing independent tasks in parallel on idle workstations, applications may substantially reduce turnaround time. However, the usefulness of remote execution is limited if processes must be terminated to reclaim a workstation when its owner returns, or if processes behave differently when they are run remotely. Sprite [8] provides a transparent process migration facility to allow noninvasive access to idle workstations. An application invokes a program remotely by performing a system call that combines migration with *exec*, replacing the process's execution image with a new program on the other host. If the owner of the remote host returns, a daemon migrates the remote process back to its own host. The primary client of migration in Sprite is a parallel version of *make* (called *pmake*), which uses idle hosts to perform compilations and other tasks in parallel. This paper discusses the experience we have had with process migration, from experimenting with an initial prototype in 1986-87 to using migration daily over the past 9 months.

*This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and in part by the National Science Foundation under grant ECS-8351961.

The next section provides some background on Sprite's process migration facility, summarizing what has appeared elsewhere [2, 3]. In Section 3, I discuss the history of process migration in Sprite, from its initial implementation to its current state. We found that migration was much harder to get working than we had expected, and even harder to keep working as the rest of the system evolved. Once migration was in daily use, however, changes in the system that affected migration were noticed immediately and corrected.

Section 4 analyzes the performance of remote execution and process migration using four metrics: the time to invoke a remote program, the time to migrate a process after it has been executing at length, the execution penalty due to transparent remote execution, and the overall speedup of application programs using remote execution to perform tasks in parallel.

Section 5 considers the lessons we have learned from implementing and using process migration over a period of time. From an implementation standpoint, we found that file system bookkeeping was the hardest aspect of process migration to get right, and we found that transparency could be provided with low overhead as long as important operations are location-independent (particularly interactions with the file system). I also draw lessons about systems in general: for example, a feature such as process migration must be used periodically if it is to work despite changes to the system.

In Section 6, I conclude the paper and discuss current and future work.

2 Goals and Design

This section summarizes the goals and design of Sprite's process migration facility. I define some terminology used throughout the paper. I then discuss the means by which transparency is supported during remote execution, and the mechanism for migrating active processes.

The primary goals of process migration in Sprite are transparency and noninvasiveness. Sprite provides *transparency* by making processes appear in all ways to execute on a single host throughout their lifetimes. The host on which the process appears to execute is termed its *home*, and the host on which it physically executes at any given time is its *physical host*. If the process's physical host is different from its home, then it is executing *remotely*. Sprite provides *noninvasiveness* by migrating a remote process during execution if its host becomes unavailable, leaving no residual dependencies on the remote host after migration. Finally, I refer to the host initiating process migration as the *source*, and the recipient of the process as the *target*.

In order to support transparent remote execution, Sprite has several relevant characteristics:

- **Shared file system.** The system has a single file system namespace, so a file name refers to the same object regardless of location. (Section 3 below discusses the poor performance of remote execution when the same name can refer to different objects on different hosts.) Processes can access files and devices on remote hosts transparently.

- **Inter-process communication through the file system.** Communication with other processes is performed using file system objects such as pipes and *pseudo-devices* [13]. Pseudo-devices are used for system services such as the X Window System and access to the internet, for which location transparency would otherwise present a problem. By using the file system for communication with the internet server, processes appear to internet hosts to be on a single host throughout their lifetime; the Sprite file system automatically forwards communication between the internet server and a remote process as necessary.

- **Location-transparent system calls.** All system calls by a remote process that depend on its location are forwarded to its home host for evaluation. Calls that interact with remote processes, such as sending signals, are redirected from the home to the physical host as needed.

- **Transparency to the user.** A remote process appears in a listing of processes on its home and retains the same process identifier throughout its lifetime. The parent-child relationships between processes are maintained regardless of where they execute, with all synchronization of exiting processes performed on the home host. Furthermore, the home host alone is responsible for knowing the current location of all processes that are tied to it; this host is similar to the LOCUS "origin site", which is the host on which a process is created [9]. However, the home host in Sprite is inherited, so children of remote processes behave as though they were created on the same host as their parent.

Processes are migrated by encapsulating their state on the source and transferring the state to the target via kernel-to-kernel remote procedure calls (RPC). The transfer cost is typically dominated by the time to send the process's open files and virtual memory to the target. To encapsulate the state of an open file, the kernel sends information about the file itself (its unique identifier, including which server stores the file, and state depending on the type of the file encapsulated) and the process's stream for the file (*e.g.*, the offset into the file, and the mode in which the file is accessed). File transfer is costly primarily because of Sprite's file system cache consistency algorithm (described in detail in [7]). Read-only files are cachable on multiple hosts simultaneously, and if a file is read and written by only one host then that host may cache the file. However, any time a file is open for writing on one host while another host accesses the file for reading or writing, the file is cached only by the server storing the file. If a file is cached by a host and then caching for the file is disabled, dirty blocks for the file must be flushed to the host storing the file, and clean blocks are discarded. When a process migrates, any files it has open for writing are briefly open for writing simultaneously on multiple hosts, and caching of those files is disabled. Measurements of the cost of cache flushing are presented below in Section 4.

To transfer a process's virtual memory, Sprite writes the process's dirty pages to a shared file server. The pages are retrieved from the server as the process page-faults. By comparison, Locus, V [11], and Charlotte [1] transfer the entire address space, which may take orders of magnitude more time than transferring the rest of the process's state. Accent addresses the "process migration bottleneck" by transferring virtual memory in a lazy fashion: the target of the migration retrieves memory from the source as it is referenced, thus amortizing the cost of memory transfer over the execution of the process [14]. Although lazy virtual memory transfer makes the act of migration faster than direct memory-to-memory transfer, it requires that the source of a migration dedicate memory to the process after the migration has completed. When a Sprite workstation is reclaimed, all resources used by foreign processes are relinquished as the processes are migrated back to their home host.

3 History of Implementation Effort

The path to a usable migration facility was long and difficult, but in retrospect was worth the effort. Migration was first implemented in Sprite in 1986, and we were able to perform measurements of its performance in the 1986-87 academic year. Our initial measurements suggested obvious areas for improvement, most notably in the area of distinguishing between location-dependent and location-independent operations. The original implementation forwarded nearly all system calls home, including calls that involved locating files, because each host maintained a distinct prefix table that mapped file system domains to servers [12]. Rather than keeping copies of the prefix table consistent between multiple hosts, naming was performed on the home host using its prefix table. Forcing naming operations to be redirected via the process's home slowed down compilation benchmarks by approximately 20%. In fact, there was no particular reason to permit the same prefix on different hosts to refer to different domains, and we solved this performance problem by legislating the equivalence of prefix tables among multiple hosts.

Although migration worked well enough to perform simple tests at this point, some features were missing: certain types of files, such as pseudo-devices, could not be encapsulated; there was no automatic host selection, so tools such as *pmake* could not yet take advantage of migration; and there was no recovery, so the failure of a host with a foreign process could affect other processes (or the kernel) on the process's home as well. Using migration on a regular basis had to await changes to fix these problems.

While we implemented additional functionality relating to process migration at the user level, the file system underwent major changes to add recovery after hosts reboot. The changes to the internal state associated with each file caused file descriptor encapsulation to become entirely unusable. Because migration was not yet in regular use, we were not even aware that the changes presented a problem until we tried working with migration again in the fall of 1987. The file system was about to be redesigned to fix a number of problems, including issues relating to process migration, so process migration itself was put on hold pending the file system changes. Those changes were completed in late spring of 1988, at which point work on process migration resumed.

Getting migration working again was difficult, mostly due to interactions with the reorganized file system. Bookkeeping between file servers and migrating processes on client workstations proved to be extremely complicated, compared to the rest of the migration facility. In particular, locking and updating the data structures for an open file on multiple hosts simultaneously provided numerous opportunities for deadlocks, race conditions, and inconsistent reference counts.

Once the reintegration with the file system was complete, we were able to implement and test the other missing pieces—error recovery and host selection—and we started using migration regularly in the fall of 1988. Regular use provided the opportunity to find and correct some additional problems that did not arise with simpler test cases. More importantly, the few changes to the rest of the system that impacted process migration were detected almost immediately and corrected.

4 Performance

Many more remote processes execute to completion than are evicted, so the user's view of the system is affected more by the overhead of remote invocation and execution than by the time to migrate an active process. The most important measurements for remote execution are the time to select an idle host, the time to start a program on another host, and the performance penalty incurred by executing remotely rather than locally. The success of remote execution may be evaluated by the overall performance improvement from parallel execution of actual applications on idle hosts. On the other hand, the success of eviction depends upon the degree to which Sprite meets its goal of noninvasiveness: in practice, the time to evict all foreign processes from a workstation is on the order of a few seconds, during which workstation owners do not appear to notice any obvious degradation in performance. Section 4.1 discusses remote execution, and Section 4.2 discusses eviction.

4.1 Remote Execution

To start a program remotely in Sprite, a process obtains the use of an idle host and then performs a remote *exec* to invoke the remote program. Methods of selecting hosts for distributing load, with and without process migration, have been discussed at length in the literature (*e.g.*, [6, 11]). Sprite uses a shared file that contains the load average and idle time of each host, as well as information about the number of foreign tasks currently using the host. To find an idle host, a process uses a library routine to lock the shared file, select a host appropriate for offloading (low load average, idle for at least five minutes, and no foreign tasks currently using it), update the count of foreign tasks, and unlock the file. When the host is no longer needed, the file is locked while the entry for the host is updated again. Sprite currently takes approximately 160 milliseconds to select and release a host, running on Sun 3/75 workstations, because all accesses to the file require network remote procedure calls.

State transfer for remote invocation is much like migration, except that no virtual memory is transferred. It currently takes 188 milliseconds on Sun 3/75's to *fork* locally, *exec* a process on a remote host with the standard set of three file descriptors (standard input, standard output, and standard error) and no dirty file blocks, and wait for the remote process to exit; this compares to 86 milliseconds when the *exec* is performed locally. Additional overhead from open files and dirty file blocks is discussed below in Section 4.2.

The total time to select an idle workstation and start a program on it compares favorably to the cost of other remote execution facilities, such as the Digital Systems Research Center distant process (**dp**) facility [10]. **Dp** takes 1 second on Firefly workstations (using multiple MicroVAX-II processors) to start a new distant process. However, **dp** takes 6 seconds to initialize before being usable, so the SRC parallel *make* facility does not use **dp** unless enough tasks may be offloaded to amortize the overhead. The cost in Sprite is relatively constant, and *pmake* will offload tasks any time idle hosts are available, even if only one task is executed at a time. By offloading tasks whenever possible, Sprite minimizes the effect of CPU-intensive operations on interactive response.

The degradation due to remote execution depends on the ratio of location-dependent system calls to other operations, such as computation and file I/O. Figure 1 shows the total execution time to run several programs, listed in Table 1, both entirely locally and entirely on a single remote host. One might expect remote execution to be slower than

Name	Description
pmake-P	recompile *pmake* source sequentially using *pmake*
ditroff	run *grap* | *eqn* | *ditroff* on a 15000-word document
rcp	copy a 1 Mbyte file to another host using TCP
fork	fork and wait for child, 1000 times
gettime	get the time of day 10000 times

Table 1: Workload for comparisons between local and remote execution.

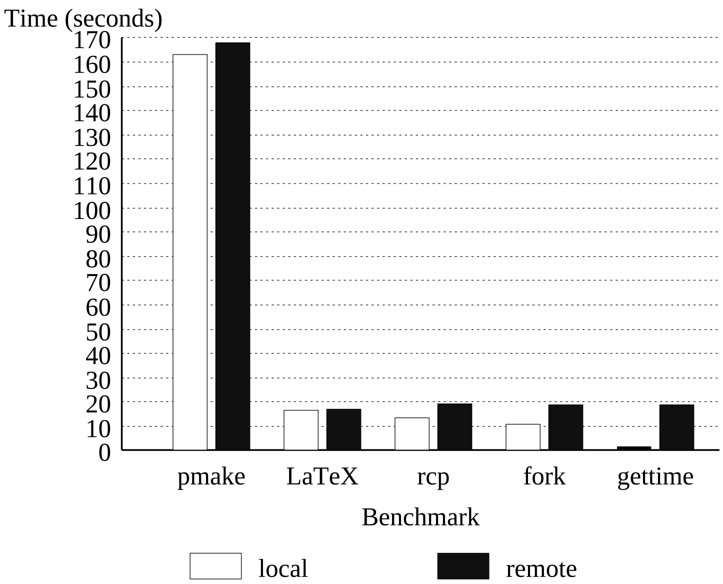

Figure 1: Comparison between local and remote execution of programs.

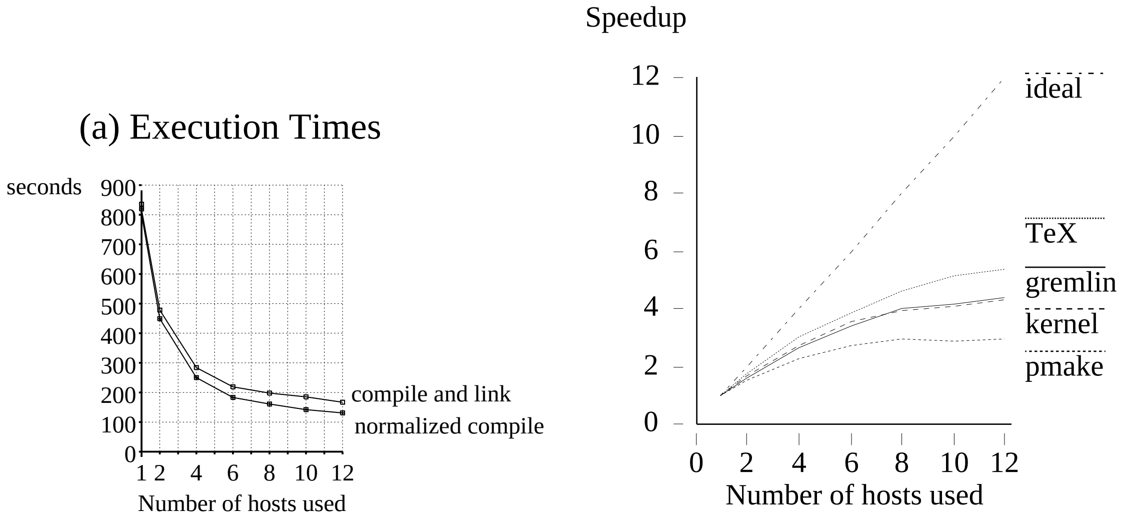

Figure 2: Performance of recompiling the Sprite file system using a varying number of hosts. Each graph shows the measured performance and the normalized (parallelizable) performance. The speedup is the reciprocal of the time saved, by comparison to using a single host.

local execution due to overhead from forwarding location-dependent system calls. As may be seen in Figure 1, however, applications such as compilations and text formatting show little effect from remote execution. In fact, executing the *ditroff* pipeline was slightly faster remotely than locally, due to differences in process scheduling while performing remote procedure calls. The next benchmark, *rcp*, copies data using TCP; it communicates with a user-level TCP server on the home node of the process performing the copy, so forwarding TCP operations to the server on the home node causes *rcp* to perform about 20% more slowly when run remotely than locally. It is also possible for a program to perform many location-dependent system calls without much user-level computation, thereby suffering a large performance penalty from running remotely. The last two benchmarks, *fork* and *gettime*, are contrived examples of this type of degradation.

The usefulness of process migration in our environment may be demonstrated by the performance of the primary application that uses migration, namely *pmake*. Figure 2 shows the total elapsed time to recompile and relink the Sprite file system using a varying number of machines in parallel, and the speedup obtained from using idle hosts. The benchmark consists of 39 independent compilations, followed by loading the resulting object files into a single file. Each migration is performed at the level of a Makefile command (*i.e.*, a single compilation). A new host is requested for each Makefile command and returned to the pool of available hosts when the command is complete. Figure 2(a) includes two curves, showing the measured elapsed times and the same times with fixed overhead removed: starting *pmake* and determining out-of-date dependencies takes about 26 seconds, and loading the object files into a single image takes 17 seconds. Figure 2(b) shows the relative improvement, for both the actual elapsed time and the portion of the compilation that could be executed in parallel. For example, using two hosts was about twice as fast as using one host, while using ten hosts was 5.5 times as fast overall as a single host. Using ten hosts showed a 7-fold improvement for the portion of the compilations that could be parallelized.

Figure 2(b) demonstrates that benefits of using a small number of hosts in parallel adequately compensate for the combined overhead of host selection, migration, and remote execution. The number of hosts that may be effectively used depends upon the relative speeds of the file server and the hosts performing the compilation. In this benchmark, the speedup was linear for small degrees of parallelism, but with 10 hosts compiling in parallel, the marginal improvement was small and the file server CPU was in use 90% of the time.

4.2 Eviction

The time to evict a process depends upon the number of dirty pages it has, the number of open files it has, and the number of dirty file blocks that must be flushed. Each dirty 8 Kbyte page takes approximately 14 milliseconds to be transferred over the network to memory on the shared backing store (plus additional time if the server's cache is full and data must be written to disk). Sprite takes about 14 milliseconds to transfer the descriptor for each open file, and 7 milliseconds to flush each dirty 4 Kbyte file block to memory on a file server. The total time (in milliseconds) to migrate a long-running process on Sun 3/75 workstations is approximated by the following formula:

$$\text{time to migrate} = 110 + 14s + 7b + 14f$$

$$\begin{aligned} s &= \text{number of dirty 8 Kbyte pages} \\ b &= \text{number of dirty 4 Kbyte file blocks} \\ f &= \text{number of open files} \end{aligned}$$

For example, to migrate a 1 Mbyte process with 50 dirty pages, 20 dirty file blocks, and 4 open files, Sprite would take 1.0 seconds. If the entire 1 Mbyte address space were dirty, migration would take 2.1 seconds.

5 Lessons

As of this writing, process migration has been in regular use in Sprite for approximately 9 months. We have had the opportunity to reach some conclusions regarding process migration and systems in general:

1. Distributed bookkeeping is difficult.
2. Insulating migration from the rest of the system is difficult.
3. Keeping the right number of idle hosts busy is difficult.
4. Hiding remote execution simplifies applications.
5. Global naming simplifies transparency dramatically.
6. Migration is expensive, to be used only as a last resort.
7. Above all, "use it or lose it."

Distributed bookkeeping is difficult

File system bookkeeping was by far the hardest part of the remote execution facility to implement. Because Sprite file servers maintain state about open files, the server must update its references when a stream to a file changes hosts. The offset with a stream may be accessed by multiple hosts as a result of migration, so the server maintains state for each stream (including the offset) as well as each file. Streams and files have reference counts associated with them, with one reference per host that accesses the stream or file, but different types of files use reference counts in slightly different ways. When a descriptor migrates, the reference count changes depending on what other references to the object exist and on the type of the file. Implementing the code to encapsulate and deencapsulate file descriptors, therefore, required intimate knowledge of the internal implementation of the file system and the state associated with each file.

Insulating migration is difficult

Sprite is not alone in finding that process migration tends to impact the rest of the system and vice-versa. Theimer refers to migration facilities as being "fragile": in an environment in which the kernel is often modified, migration can break unless everyone modifying the kernel keeps the migration facility in step with other kernel changes [11]. Finkel and Artsy, on the other hand, report that they were able to keep migration sufficiently modular to keep changes to migration from breaking other parts of the kernel and changes elsewhere in the kernel from breaking migration [5].

Although file encapsulation proved to be a thorn in the side of process migration for some time, migration has evolved to be generally orthogonal to the rest of the system. Many kernel modules in Sprite maintain state on behalf of each process. Originally, to encapsulate the state of a process, the process migration facility called a predetermined set of encapsulation procedures, one per module, and each module's portion of the process state was transferred in a separate RPC. When a new module was added to the system, migration would break temporarily unless the state of the new module were encapsulated. We therefore changed migration to use a set of "callbacks" into each module to encapsulate its own portion of a process's state. The migration facility on the source requests the size of the encapsulated state of each module, allocates a buffer to hold the collective state, makes the callbacks to encapsulate the state, and transfers the state in a single RPC to the target. New modules may be added to the system by adding an entry to the callback table; changes to existing modules may be performed without affecting the migration facility itself, by updating the module-specific encapsulation routine whenever the format of the process state changes.

Separating the functionality of migration on a per-module basis proved to have a useful side-effect: implementing process migration on a new architecture required only that a small number of machine-dependent state encapsulation routines be rewritten. It took only about half a day to implement migration on the Decstation 3100, given the existing implementation for Sun workstations.

Keeping idle hosts busy

Pmake performs unquestionably well when performing a small number of independent tasks, but large tasks present some problems. On the one hand, the server's CPU is a bottleneck if too many hosts are used simultaneously. On the other, *pmake* sometimes has trouble using more than a single host. While we can't do much about the server except to get faster and more plentiful CPU's, getting *pmake* to do more in parallel could be beneficial. As an example, the Sprite kernel is stored hierarchically, with each module having its own Makefile and a single Makefile at the top level of the source tree. If *pmake* is invoked at the top level with a high degree of parallelism, permitting it to invoke several *pmake* processes on idle hosts, then those *pmake*s must be careful not to use much parallelism or they will saturate the server. If they are invoked with low parallelism, then a large module will slow down the entire compilation when it is performed sequentially after the other modules are completed. Currently, only one recursive *pmake* is ever performed at a time, so the child *pmake* can use a high degree of parallelism. However, when the child hits a synchronization point, such as loading all the object files in a module into a single image, only one host is used.

Ideally, we would like to be able to build the kernel in parallel with a single *pmake* controlling the degree of parallelism. One module could be compiled in parallel as one or more modules were completing their linking phase. The problem of independent modules is most likely an artifact of the way we chose to structure the source hierarchy before parallel compilation was available, and we have learned our lesson.

Hiding remote execution

If changing a process's location can change the effects of its execution, then users must take special care to use remote execution only when they know *a priori* that a program is location-independent. For example, the V System preemptable remote execution facility is restricted to applications that execute "only operations whose output is independent of the location at which they are executed" [11]. Although compilations and text formatting are location-independent, many other programs are not: for example, what if *rcp* could not run remotely, and a user invoked *rcp* from within a Makefile? In general, any program that one can invoke from *pmake* should be capable of executing remotely and being evicted when necessary. Sprite only restricts processes that map kernel memory into their address space, and processes that are pseudo-device servers, such as the X Window System display manager.

To the users of applications such as *pmake*, remote execution is invisible. The application merely appears to execute much faster than one would expect it to on a single host. If a set of processes is evicted from another host, they immediately start executing on the home host, perhaps with some performance degradation due to sharing the host with other active processes. We hope to implement a mechanism by which processes may be automatically re-migrated to another idle host if they are evicted, but eviction happens so infrequently that the lack of automatic re-migration does not seem to present a problem.

To a user reclaiming his or her workstation, eviction is invisible as well—or it would be if the daemon evicting processes did not announce the eviction in the system log. We found that messages informing the user when eviction takes place promote goodwill, because users can see that their performance is not impacted as a result of foreign processes.

Global naming is a must

When process migration was first designed, each Sprite host was a distinct system with its own file system namespace and its own process identifiers. The simplest method of guaranteeing location transparency was to forward nearly all system calls home, but performance suffered significantly. Over time, Sprite shifted toward making most system calls location-independent: file naming operations go directly to the server for a file system domain, since file names mean the same across multiple hosts; and process identifiers include the process's home host, so a remote process may send a signal using the standard signalling mechanism on its physical host. By reducing the amount of forwarding required to support remote processes, we were able to improve the performance of remote execution while simplifying it substantially.

Migration is expensive

Our experience with the relative costs of remote invocation and migration corroborate the results of Eager, *et al.*, who used a theoretical model and simulation to compare migratory and nonmigratory load sharing. They concluded that migrating processes for load sharing performance does not generally yield significant improvement over policies with only remote invocation, and they suggested that "costlier but simpler" migration may be appropriate if migration is done primarily for purposes other than load sharing (such as permitting workstation owners to reclaim their hosts) [4].

Remote invocation in Sprite is inexpensive enough to provide performance improvements for all but extremely short-lived processes, assuming that the local host is already highly utilized. Migrating active processes, on the other hand, is often measured in seconds rather than milliseconds. The disparity between migrating new processes and processes with many dirty pages and file blocks suggests that migration is unlikely to be useful for dynamic load balancing. As a last resort to guarantee the response time to the owner of a workstation, however, eviction has proved an appropriate use for migration.

Use it or lose it!

Our single greatest mistake when implementing process migration was to let it sit idle while the rest of the system evolved. We did not have the manpower at the time to add the features described in Section 3, but we could have run simple test cases on a regular basis to ensure that problems would be apparent shortly after being introduced to the system. If we had known quickly that the changes to implement file system recovery had affected migration, the recovery support could presumably have been modified in the process of fixing other problems with it. Instead, we were not aware of a problem until well after the changes had become "carved in stone". The changes to support recovery, which involved several data structures that had been designed without taking the possibility of migration into account, would have required too much effort to fix—given that the entire file system was to be rewritten. Instead, when the file system was redesigned, we paid careful attention to the effects of migration and implemented special functionality to handle migration. This functionality could and should have been incorporated into the system at a much earlier point, given that it was ultimately necessary.

Since migration has been in general use, there have been several occasions when changes elsewhere in the kernel caused problems for migration. Because migration is used frequently for compilations and other tasks, in each case we quickly observed that migration had been affected. By catching the problems quickly, we were able to correct them relatively easily.

6 Conclusions and Future Work

Some time ago, shortly before the file system was to be reimplemented, we had a lengthy discussion about the future of process migration in Sprite. The consensus at the time was that migration was probably a mistake: it was too difficult to implement, and the performance of a single workstation was sufficient for our needs. However, we believed that the marginal cost to put migration into general use was small enough to justify finishing the implementation and giving migration a chance to prove itself.

In retrospect, I may safely say that our initial lack of faith was misplaced. Process migration has evolved from a toy prototype to a mature, extremely useful facility. Users are thankful not only for the significant performance improvement they see when using other hosts, but for the minimal impact other users have on their own workstations.

Our present work with process migration may be divided into three categories: basic support; extensions; and measurement and analysis. Migration is currently usable only on Sun 2, Sun 3, and Decstation 3100 workstations, and only between two machines of the same architecture. We plan to port migration to Sun 4 workstations, and if possible, provide the ability to perform remote *exec*s between machines of different types. The ability to perform heterogeneous remote *exec*s, along the lines of the LOCUS *rexec* system call [9], could considerably expand the pool of idle hosts available to a single program. We would also like to add automatic remigration after eviction to keep eviction from degrading the performance of the home host.

Finally, we intend to instrument the process migration and host selection facilities to evaluate more aspects of the system, such as migration overhead, host availability, and system bottlenecks. Preliminary measurements of the rates of remote execution and eviction suggest that eviction in practice is rare (perhaps one eviction per 50 remote executions) and takes well under a second on Sun 3/75's for typical compilations. Initial measurements of host usage indicate that about one-third of our workstations are available for migration during the day, on average, and over the course of a weekend closer to two-thirds are available. Server CPU utilization is the most likely bottleneck that would affect overall speedup from parallel execution, but we must await faster servers before we can obtain useful measurements of our new client workstations: for example, a Sun 3/180 file server was 50% utilized servicing requests from two Decstation 3100 clients compiling in parallel. Access to the shared file containing host availability may also prove to be a bottleneck, and we are exploring alternative methods for host selection that might scale better with the size and speed of the system.

Acknowledgements

Peter Danzig, John Hartman, Mendel Rosenblum, Mark Sullivan, and Brent Welch provided valuable comments on early drafts of this paper. John Ousterhout has been instrumental in the design of the process migration facility. Michael Nelson and Brent Welch implemented file descriptor encapsulation and provided debugging assistance. Finally, I thank the referees

for their recommendations, which helped to improve the clarity and organization of this paper.

References

[1] Y. Artsy and R. Finkel. Simplicity, efficiency, and functionality in designing a process migration facility. In *The 2nd Israel Conference on Computer Systems*, May 1987.

[2] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[3] F. Douglis and J. Ousterhout. Process migration in Sprite: A status report. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):8–10, Winter 1989.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *ACM SIGMETRICS 1988*, May 1988.

[5] R. Finkel and Y. Artsy. The process migration mechanism of Charlotte. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):11–14, Winter 1989.

[6] P. E. Krueger. *Distributed Scheduling for a Changing Environment.* PhD thesis, University of Wisconsin, Madison, Wisconsin, June 1988. Computer Sciences Technical Report #780.

[7] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[8] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[9] G. J. Popek and B. J. Walker, editors. *The LOCUS Distributed System Architecture.* Computer Systems Series. The MIT Press, 1985.

[10] E. Roberts and J. Ellis. *parmake* and *dp*: Experience with a distributed, parallel implementation of make. In *Proceedings from the Second Workshop on Large-Grained Parallelism.* Software Engineering Institute, Carnegie-Mellon University, November 1987. Report CMU/SEI-87-SR-5.

[11] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[12] B. B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed filesystem. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 184–189, Boston, Mass., May 1986. IEEE.

[13] B. B. Welch and J. K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *USENIX 1988 Summer Conference*, pages 37–49, San Francisco, CA, June 1988.

[14] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–22, Austin, TX, November 1987.