Sprite on Mach

Michael D. Kupfer
University of California at Berkeley

Sprite on Mach

Michael D. Kupfer

University of California, Berkeley†

Sprite is a distributed operating system that supports a fast, single-image network file system and transparent process migration. Over a period of 19 months we ported Sprite to run as a server on top of the Mach 3.0 microkernel. Although the resulting server does not implement some Sprite features, it can run in an existing Sprite cluster, and it supports standard UNIX programs like vi, gcc, and make.

Porting Sprite to Mach was generally straightforward, though there were some difficulties. Many of the problems were related to asynchronous interactions between the Sprite server, Mach, and Sprite user processes. Others resulted from trying to maintain native Sprite's internal interfaces in the Sprite server.

The Sprite server is 22% smaller than an equivalent Sprite kernel, and it contains almost no machine-dependent code. These improvements should significantly simplify porting Sprite to new hardware platforms. Unfortunately, the Sprite server runs the Andrew benchmark at only 38% of the speed of native Sprite. None of the performance problems appears insurmountable, but they could require a long time to track down and fix.

1. Introduction

Sprite is a distributed operating system that was developed at the University of California, Berkeley [14]. It features a fast, single-image network file system [19], transparent process migration [6], and a high-performance log-structured local file system [17]. Sprite was originally written from scratch to support the SPUR multiprocessor project at Berkeley [10], so the kernel is multithreaded and uses fine-grain locking.

We have ported the Sprite kernel to run as a server on top of Mach 3.0. The server does not have complete Sprite functionality, but it can run most UNIX commands as a client of native Sprite file servers. We used the modified Andrew benchmark [15] to partially tune the server and to analyze the remaining performance problems.

Section 2 of this paper explains why we ported Sprite to Mach. Section 3 sketches the design of the Sprite server and discusses a few of the problems that arose during design and testing. Section 4 shows how the Sprite server is smaller and more portable than native Sprite. Section 5 shows that the server is significantly slower than native Sprite, and it explains some of the known bottlenecks. Section 6 evaluates the Sprite server, and Section 7 lists possible future work. Section 8 closes with some general conclusions from the project.

† Author's current address: SunSoft, Inc., 2550 Garcia Avenue, MS MTV05-40, Mountain View, CA 94043-1100.
kupfer@eng.sun.com

2. Why mix Sprite and Mach?

The Sprite project became interested in Mach for three reasons: to make Sprite more portable, to make Sprite easier to distribute to external sites, and to verify whether Mach is a suitable platform for building distributed systems.

The Sprite group at Berkeley has always been small (5–8 people), and very few sites outside Berkeley run Sprite. Time spent doing ports, or time spent preparing and supporting external releases, is time that cannot be spent doing research, and a small group cannot easily afford this lost time. We hoped that by implementing Sprite as a Mach-based server, we could rely on the Mach community to write most of the code to support new hardware (e.g., device drivers and low-level memory management code). We also hoped that by distributing Sprite as a server, rather than as a complete operating system, we could reduce the amount of time that our staff spent supporting external sites.

On the other hand, we were concerned about the potential performance loss of a server-based system. Although the UNIX server [9] had demonstrated that a server-based UNIX system could perform adequately, and the shared memory server [8] had demonstrated that Mach was sufficiently flexible to support a system like Sprite, there were no servers that provided complete operating system support with heavy reliance on the network. We concluded that Sprite would be an excellent test case for Mach's ability to support high-functionality distributed systems.

3. Server design and issues

Four design goals for the Sprite server emerged from the project goals listed in Section 2.

1. The server should have as little machine-dependent code as possible, so as to make it more portable.
2. The design should be simple, so as not to complicate future Sprite research.
3. The design should minimize changes to existing Sprite code, so as to reduce development time and to simplify the eventual integration of new features and bug fixes from native Sprite.
4. Performance should be comparable to native Sprite, though slight performance degradation is acceptable in return for improved portability.

These goals led to a design that is generally similar to that of the UNIX server. Nonetheless, there are important differences between the two servers. Furthermore, there were important design issues that our reading of the Mach literature had not prepared us for.

3.1 Design overview

As a first approximation, Mach can be thought of as a high-level abstract machine. It provides processes, scheduling, a simple interface for accessing a process's memory, and a machine-independent interface for accessing local devices such as disks or networks. The C Threads library [5] provides threads, locks, and condition variables. Thus the first step in porting Sprite was to replace low-level native code from the Sprite kernel with equivalent Mach-based code. The second step was to replace system call stubs in Sprite's C runtime library with emulation routines that make Matchmaker RPC requests to the Sprite server. These steps are similar to the transformation that was used to port UNIX to Mach [3, 9].

The result is the system shown in Figure 1. User processes communicate with the Sprite server using Matchmaker RPCs, which use Mach interprocess communication (IPC) as a transport. The emulation code in each process can also make direct kernel requests (e.g., to allocate scratch memory). The Sprite server manipulates user processes and performs I/O by sending requests to the kernel. The server also acts as an external pager [20], so that Sprite file servers can provide backing store for processes.

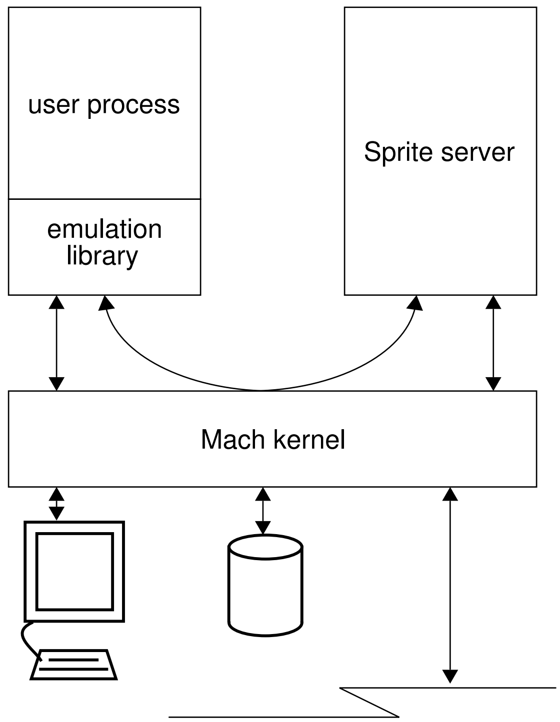

FIGURE 1. Architecture of Sprite on Mach. Each user process communicates with the Sprite server via emulation routines that use Matchmaker RPCs instead of system calls. The emulation routines and the server can also make direct kernel requests. The kernel mediates access to devices like terminals, disks, and the network.

For some code, particularly for device management, the native Sprite routines mapped well to Mach primitives, so they could be replaced by a simple facade. A few routines, such as the process scheduler and the kernel debugger stub, did not require any replacement code at all. Other code did require substantial rewriting, though as Section 4 shows, this was a small fraction of the total code. The rewritten code includes

- the guts of `fork` and `exec`, which required substantial changes to work with the Mach task and thread primitives.
- the "system call" code, which had to accept Matchmaker RPC requests instead of traps.
- an internal lock package, which was rewritten to sit on top of C Threads.
- the virtual memory code, which was rewritten to use Mach primitives and to provide a Sprite external pager.

With two exceptions, the Sprite server required no changes to the standard Mach 3.0 distribution. (This doesn't count a few bug fixes which have since been incorporated into the Mach release.) The first exception was a couple small changes to improve networking performance; see Section 5.1 for details. The second exception was to adopt the thread-local data package from the multi-server [11] version of C Threads. This package made it easier for the Sprite server to bind an internal thread to a specific Sprite user process while handling a request from that process (see Section 3.4).

The Sprite server and emulation code use Mach IPC only for local communication. For remote communication, the Sprite server uses Sprite RPCs; Mach serves only as a high-level interface to the network. There were two reasons for this approach. First, it required no changes to native Sprite servers. Second, we believed that using Mach IPC for remote communication would be slow because of performance problems with the `netmsg` server.

One difference between the UNIX and Sprite servers is how the emulation code is set up in each user process. The UNIX emulation code lives at a fixed address, is inherited when a process forks, and is typically invoked by redirecting a system trap. The Sprite emulation code does not have a dedicated address space; it is simply a part of the C runtime library and is invoked by a normal procedure call. The disadvantage of this approach is that it complicates binary compatibility. The advantage is that it was easy to implement, and we could change it later (e.g., after initial tuning) without throwing away much code.

A more important difference between the UNIX and Sprite servers is how they use external pagers. The UNIX server provides an external pager for mapped files, including program text, but it uses the Mach default pager for swap storage (i.e., a process's heap and stack). The Sprite server provides an external pager that backs the entire address space of a user process, including its heap and stack. This design lets a process page from a network file server, as is done in native Sprite. Sprite uses network paging to support diskless operation—almost all Sprite workstations are diskless—and to support process migration.

Most of these changes were straightforward to design and implement. Nonetheless, some changes presented unexpected difficulties. Many of the problems were related in some way to asynchronous interactions between the Sprite server, Sprite user processes, and Mach. The rest of this section will discuss some of the problems that arose during design and testing. None of the problems was insurmountable, but each required extra time to get the details of the design and implementation right.

3.2 Shared data structures

Some data structures, such as memory objects and process table entries, are logically shared between the Sprite server and Mach, and special care is needed to ensure that the data structures are managed correctly.

For example, consider the Sprite internal routine `Vm_MakeAccessible`, which converts a user address range to a kernel address range and guarantees that the kernel can access the range without faulting. The native Sprite kernel uses this routine in some places instead of `copyin`[1] to access user memory. Figure 2 outlines the steps for a Mach-based implementation of

1. The actual name of the Sprite routine is `Vm_CopyIn`, but we'll generally use the names of UNIX equivalents throughout the paper.

Vm_MakeAccessible. Note that Step 2 requires that the Sprite server know the name port for every memory object that it has created. Unfortunately, Mach uses an asynchronous callback to tell the Sprite server what each memory object's name port is. During early development Vm_MakeAccessible frequently failed because it was unable to find the name port that vm_region had returned, which happened because Mach had not yet registered the name port with the server. The fix for this problem was to have the code in Step 2 read from the user address before looking up the name port. This forced Mach to finish initializing the memory object, which provided the name port to the server.

1. Use vm_region to get the name port of the memory object for the given user address.
2. Use the name port to find the memory object (e.g., using a hash table or linked list in the server).
3. Convert the user address to the corresponding memory object offset.
4. Map the memory object page into the Sprite server.

FIGURE 2. Steps for Vm_MakeAccessible.

For a more involved example, consider what the Sprite server must do to clean up when a process with a mapped file exits. As shown in Figure 3, Sprite represents the mapped file as a Vm_Segment, which contains a handle for the file as well as ports for the corresponding Mach memory object. When the process exits, the server asks the kernel to clean any dirty pages in the memory object. It then asks the kernel to delete the memory object. Eventually the kernel responds with a callback RPC, informing the server that it can null out its remaining references to the memory object and free the Vm_Segment.

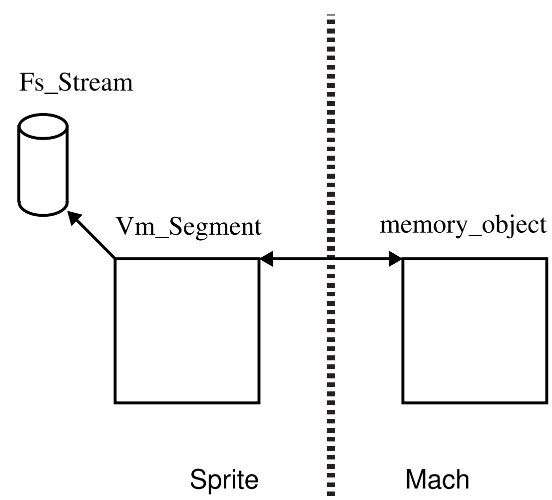

FIGURE 3. Data structures for a mapped file.

Now suppose that a second user process wants to map the same file in after the server has requested cleaning but before the server has received the callback RPC. Should the server wait until the old `Vm_Segment` is gone before creating the new one? Such an approach might lead to performance problems or deadlocks. Alternatively, should the server try to reuse the `Vm_Segment`? This would complicate code that is already somewhat hairy. Or should the server create a new `Vm_Segment`? This would invalidate an initial assumption in the server's design, which was that at most one `Vm_Segment` would exist at any time for a given file.

The Sprite server currently takes the first choice: it waits for the old `Vm_Segment` to go away before creating a new one. This approach did not seem to cause problems during development or tuning. Nonetheless, for production use it may be better to reuse the `Vm_Segment`, at least while the segment is being cleaned.

3.3 copyin() failures

Another problem that cropped up during development was how to implement `copyin` correctly. One obvious implementation of `copyin` uses `vm_read` to get the desired bytes from the user process and then uses `vm_write` or `bcopy` to write the bytes into the server's buffer[2]. Such an implementation is simple to code, and it can leave much of the error checking to `vm_read` (e.g., to verify that the given user address range is valid).

Unfortunately, `vm_read` only checks whether the user's address range is mapped; it doesn't check whether the bytes are actually accessible. If any page in the requested range is not in memory, the server won't discover an accessibility problem until it tries to use the bytes, the page-in fails, and the server gets an address exception. This design might be acceptable for a system such as UNIX, because the backing store is usually a local disk, and failures should be very rare. Unfortunately, this design is not acceptable for Sprite because the backing store is a Sprite file server, which is subject to arbitrary (and sometimes frequent) failures.

This means that the Sprite server must catch its own addressing exceptions. The exception handler must check whether the exception happened during a `copyin` or `copyout`, so that it can either panic or cause the copy operation to fail gracefully. The Sprite server does not currently have such an exception handler, but file server failures were infrequent enough that they did not seriously impede development or tuning. Nonetheless, it would be necessary to add an address exception handler before putting the Sprite server in production use.

3.4 Maintaining internal interfaces

Maintaining Sprite's internal interfaces also required some effort. Native Sprite is like UNIX in that user processes run in "kernel mode" after an interrupt or after causing a trap. As a consequence, native Sprite was designed so that a process can kill or suspend only itself. An attempt to kill or suspend a different process is simply a request, which the target process eventually acts on.

In contrast, the Sprite server is a separate Mach task that user processes invoke via RPCs. To maintain the illusion of a "current process" that has trapped into the kernel, the Sprite server uses the C Threads local data package to bind a user process to the thread that is handling that process's RPC. Unfortunately, this approach has some problems. First, special care is needed, e.g., in `exit` and `exec`, to ensure that the threads correctly manage internal resources like buffers. Second,

2. Of the two, `bcopy` is simpler and usually faster; see Section 5.2.2.

because the Sprite server does not handle time slice interrupts, it cannot guarantee that user processes will notice attempts to kill or suspend them. Thus, unlike in native Sprite, user processes must be able to kill or suspend other user processes. This requirement led to many changes in the locking strategy for the process management and signals code in the Sprite server, and these changes were the source of many bugs.

3.5 Asynchronous signal delivery

An additional problem with signals was how to invoke a user signal handler for asynchronous signals (that is, when a process invokes `kill` on some other process). The basic idea is simple: suspend the target process, push the call frame for the handler onto the stack, change the process's program counter to invoke the handler, and resume the process.

The problem is that if the process is, say, doing a Mach system call, the handler should not be invoked until the process returns from the kernel. There are ways to deal with this problem, such as

- carefully examining the process to see what it's doing
- having a dedicated thread in the user process to handle signal synchronization

but at first glance these approaches seemed like they might be slow or too complex to get right.

The Sprite server currently uses an approach similar to that of the UNIX server. Each call to the Sprite server returns a flag telling whether there is a pending signal that has a registered handler. If there is such a signal, the emulation code gets the signal information from the server and calls the handler before returning from the original call. This solution is not ideal (a signal handler can't be called until the program makes a Sprite request), but it was adequate for a prototype. In retrospect, the "dedicated thread" approach, which is how the V [4] and Spring [12] projects handled this problem, might have been the better choice.

4. Status and code measurements

Despite the problems mentioned in the previous section, the Sprite server works and is about 75% complete. This section explains the server's current status and presents some code size measurements.

One person (the author) developed the Sprite server over a period of 19 months. The first 7 months of the project were a halftime effort; the remaining 12 months were full-time, including 2.5 months of performance tuning. Development began on a Sun 3 but later moved to a DECStation 5000. To simplify debugging, the Sprite server runs as a UNIX application, but it only depends on UNIX for "console" access and for obtaining Mach privileged ports.

The Sprite server supports standard UNIX programs like `vi`, `gcc`, and `make`. Nonetheless, the implementation is incomplete, lacking features such as

- binary compatibility (with either the vendor operating system or native Sprite)
- local disk access
- support for debugging user processes
- process migration

As shown in Figure 4, porting Sprite to Mach let us get rid of almost all the machine-dependent code in Sprite. A Sprite DECStation kernel with the same functionality as the Sprite server contains roughly 143,000 lines of code, of which 27,000 lines are machine-dependent, including 3,600 lines of assembly code. The Sprite server contains 111,000 lines of code, of which 1,300 lines are machine-dependent. The server contains 4 lines of assembly code to help debug locking errors.[3]

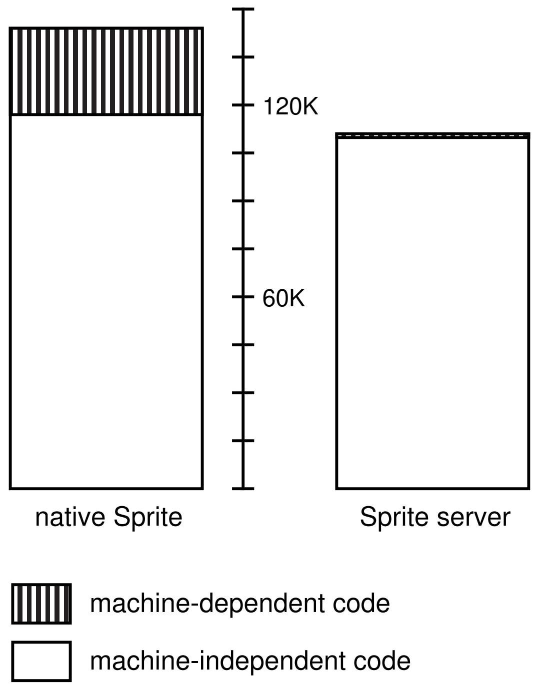

FIGURE 4. Comparison of code sizes (source lines).

Figure 5 shows how much code changed in converting the Sprite kernel to the Sprite server. We kept about 90,000 lines of code (63%). We threw away another 39,000 lines of code (27%), and we rewrote the remaining code, about 14,000 lines (10%), for use with Mach. The thrown away code consisted of low-level (and often machine-dependent) code for device, process, and memory management, plus code that duplicated Mach functionality (e.g., the process scheduler, a kernel debugger, and various C utility routines). Most of the rewritten code was for process and memory management, Matchmaker RPC processing (system calls in native Sprite), and Sprite's internal lock package.

To support the missing functionality mentioned earlier in this section, we would have to port an additional 58,000 lines of code, of which 2,400 lines are machine-dependent. About 52,000 lines of this code would probably stay the same, another 2,000 lines would probably get thrown away, and the remaining 4,000 lines would probably require rewriting.

3. The server obtains routines such as `bcopy` from the UNIX C library that is provided with Mach. Because the Sprite group would not have to maintain this code, it is not included in the Sprite server line counts.

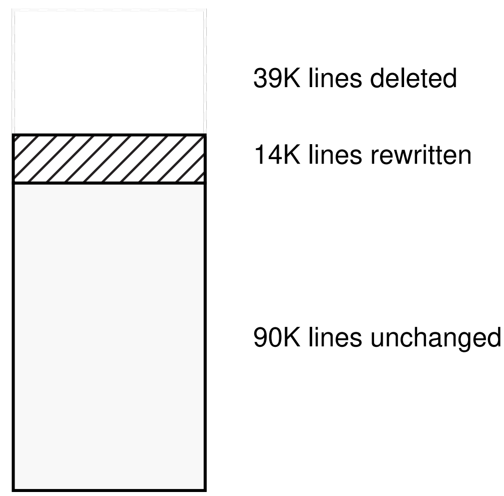

FIGURE 5. Code disposition, changing the Sprite kernel to the Sprite server.

5. Performance

Due to time constraints, we primarily used only one benchmark to tune the Sprite server: the modified Andrew benchmark [15]. This benchmark copies a file tree, scans the tree several times, and then compiles some window system code. It causes the Sprite server to generate around 8,200 Sprite RPCs.

This section discusses performance fixes to early versions of the server, plus current performance problems. All the performance numbers in this section were obtained on a DECStation 5000 with at least 32 megabytes of memory, using files stored on a native Sprite file server. RPC times are for two DECstation 5000s, one running native Sprite, the other running either the Sprite server or native Sprite.

5.1 Current status and early tuning

As shown in Figure 6, a diskless native Sprite workstation runs the Andrew benchmark in 91 seconds. The UNIX server (version UX34) and Ultrix 4.2 need around 118 seconds using files on a local disk. For an Ultrix 4.2 system accessing the files via NFS, the time goes up to 141 or 186 seconds, depending on whether the NFS server has a Prestoserve accelerator. The Sprite server's fastest time for the benchmark is 237 seconds.

Early versions of the Sprite server required 425 seconds to run the benchmark. Figure 6 shows successive improvements in the server's performance, which benefitted from three general fixes: caching of text (code) segments, reduced Sprite RPC latency, and more efficient string management in `exec`.

Although native Sprite caches unused program text, we originally disabled the cache in the Sprite server so that we wouldn't have to deal with the file system and virtual memory code for detecting and removing stale text pages. Unfortunately, this meant that the server spent much of its time

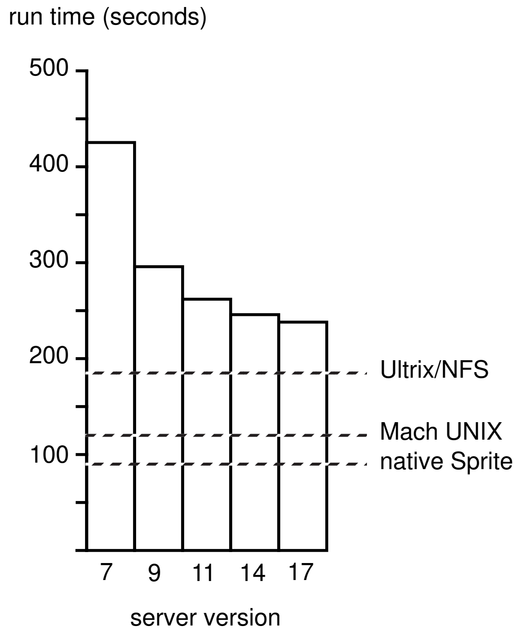

FIGURE 6. Sprite server performance. This chart shows the run time of successive versions of the Sprite server for the Andrew benchmark. Version 7 was the first version run with the benchmark. Version 9 cached unused program text. Versions 11 and 17 improved RPC performance. Version 14 used a faster version of `exec`. The chart also compares the Sprite server with related systems. Note that the Ultrix and native Sprite figures are for remote access, while the Mach UNIX figure is for local access.

faulting in the same pages for commonly used programs like `cp` and `grep`. Re-enabling the text cache reduced the benchmark run time by 130 seconds.

RPC performance in the Sprite server suffered for several reasons:

1. The UNIX server used a packet filter that accepted all packets. Thus Sprite operations would be delayed while the UNIX server read and discarded Sprite RPC packets. The solution was to fix the UNIX server's packet filter to ignore Sprite RPC packets.
2. Early versions of the server used synchronous network writes, even though they were slower than asynchronous writes. This problem was a consequence of native Sprite's interface to the network driver, and it was compounded by a server bug. Fortunately, changing the RPC code to use Mach meant that the writes could all be asynchronous, which is how the server currently operates.
3. The Sprite server was generally unable to make inband write requests because of the 128-byte size limit for inband `device` requests. This was unfortunate because inband writes were ten times faster than out-of-band writes (0.18 ms versus 1.8 ms). By raising the size limit for inband writes to 300 bytes, we were able to get the Sprite server to use inband writes for 90% of the packets that it sent.

The combination of these fixes reduced the null Sprite RPC time from 4 ms to 2.2 ms. Overall the RPC improvements (latency and throughput) translated to a 41-second speedup for the Andrew benchmark.

The other major performance fix was in the way that `exec` copied in environment and argument strings from a user process. Originally the process would pass an array of string addresses to the server, and the server would copy each string in one at a time. By changing the Matchmaker stub and Sprite emulation code to pass the strings in with the `exec` request, we were able to reduce the benchmark time by 17 seconds.

5.2 Current performance problems

Unfortunately, this still leaves a 146 second gap between native Sprite and the Sprite server. We can explain about 97 seconds of this gap (see Figure 7), and we have ideas for how to fix some of the problems. We know where most of the remaining 49 seconds are being spent, but not what is causing the delays. Tuning stopped in July 1992 when the Sprite server project was brought to a close.

The rest of this section will explain the current top 5 understood or partially understood bottlenecks.

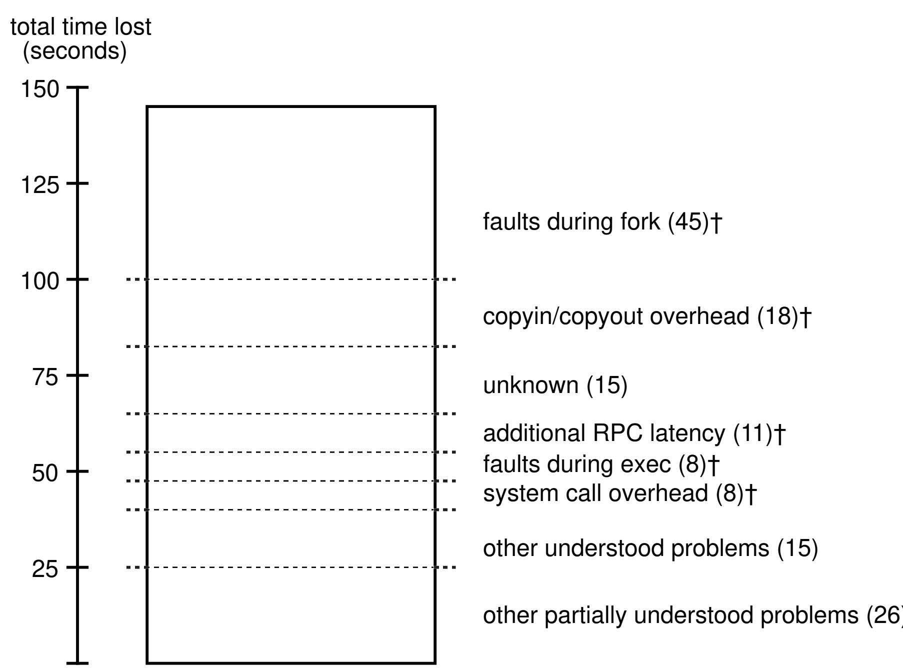

FIGURE 7. Remaining performance problems. This graph illustrates the current performance gap between the Sprite server and native Sprite for one run of the Andrew benchmark. The numbers in parentheses tell how many seconds are lost to each problem. Problems marked with a dagger (†) are discussed in the main text. “Unknown” refers to time that has not been accounted for. “Other understood problems” refers to well-understood problems that are not discussed in the paper. “Other partially understood problems” refers to problems where only the general area of the problem is known (e.g., somewhere in `fork`).

5.2.1 overhead in fork()

The largest single bottleneck is currently in `fork`, which spends an extra 45 seconds during the benchmark copying the parent's heap and stack. This copying is necessary because Mach does not currently support copy-on-write for externally managed memory objects.

The UNIX server avoids this problem by using the default pager for the heap and stack, which does support copy-on-write. It is not feasible to make the Sprite server the default pager because the default pager interface lacks the necessary hooks to support Sprite process migration. The correct solution is for Mach to support copy-on-write for externally managed objects. This support should appear as part of Joe Barrera's work to support distributed memory multiprocessors [2].

5.2.2 overhead in copyin() and copyout()

The second biggest bottleneck is in `copyin` and `copyout`. This bottleneck accounts for 18 seconds of the benchmark time, and it is primarily associated with user `read` and `write` calls.

The UNIX server uses mapped files to avoid this problem [9]. That is, the emulation library maps a portion of the file into the process's address space. The library satisfies `read` and `write` requests by copying directly from or into this mapped region. Unfortunately, native Sprite doesn't support consistent access to mapped files that are shared by multiple clients; it only supports consistent access using explicit `read` and `write` calls [1]. Thus we could change the Sprite server to use mapped files for `read` and `write`, but first we would have to fix native Sprite to provide consistent shared access to the files.

Fortunately, there are other options that are worth investigating. Instrumentation shows that the bottleneck results not from extra copying, but from the Mach kernel calls needed for the server to access the user process's address space. So, for example, it may suffice for the server to cache one or more mappings into the process's address space and use those mapped regions for `copyin` and `copyout`. Alternatively, it may be acceptable for the Sprite server to used mapped files in the normal (unshared) case, switching to explicit `read` and `write` calls only when the file is open for writing on multiple clients, which happens infrequently [1].

5.2.3 RPC latency

The third biggest bottleneck, which we estimate to be around 11 seconds for the Andrew benchmark, is higher Sprite RPC latency, despite the improvements that were made during tuning. The Sprite server requires 2.2 ms to do a null RPC to a native Sprite system, whereas a native Sprite system can do the same operation in 0.8 ms. We suspect that most of the time is spent in the network input path (packet filter, etc.), though we did not have time to verify this through measurements.

Assuming that Mach is responsible for most of the slowdown, there are two ways to fix it:

1. Improve general network performance. The work described by Reynolds and Heller [16] is an example of this approach.
2. Reduce the number of RPCs. Instrumentation shows that 36% of the Sprite RPCs that are generated by the Andrew benchmark are `open` RPCs. Adding a name cache would probably reduce this number considerably [18], though it would require changes to native Sprite.

5.2.4 overhead in exec()

The fourth biggest bottleneck is in `exec`, which appears to spend about 8 seconds in the Andrew benchmark faulting in new heap and stack pages. The problem here is that the `exec` code in the Sprite server currently destroys the heap and stack and then creates new ones. This causes an additional 8,300 external pager calls to the Sprite server, primarily to create zero-fill heap and stack pages, with each call taking an estimated millisecond of overhead. We expect that we could reclaim at least 7 of the 8 seconds by changing the server to reuse the old heap and stack.

5.2.5 "system call" overhead

The fifth biggest bottleneck, which accounts for another 8 seconds of benchmark time, is in the Sprite server's "system call" code, which processes Matchmaker RPCs from user processes. The slowdown appears to come from a context switch that happens every time a new request is received. The problem is that the server thread that receives a user request is not the thread that processes it. The intent of this design was to avoid a race between getting a request and reclaiming an entry in the process control table. The correct approach, which would eliminate the context switch, would be to use no-senders notification for reclaiming process table entries.

6. Evaluation

This section evaluates the Sprite server according to the design goals in Section 3.

1. *Portability*. The Sprite server appears to be highly portable. As shown in Figure 4, it contains less code than an equivalent Sprite kernel, plus there is very little machine-dependent code and essentially no assembly code. The unimplemented code from native Sprite is mostly machine-independent, so a fully functional Sprite server should still be very portable.
2. *Simplicity*. Although the Sprite server is not as simple as we'd like (because of the design complications described in Section 3), most of the Sprite server design and implementation were straightforward, so we believe that the server met its simplicity goal. Furthermore, debugging the server with `gdb` was generally easy. In fact, we were able to track down bugs that were inherited from native Sprite and had long been puzzling the Sprite group.
3. *Minimize changes*. Porting Sprite to Mach involved an acceptably small number of changes to native Sprite. Most of the server code is identical to the kernel code from which it is derived, and we made few changes to Sprite's internal interfaces. Furthermore, the Sprite server runs in an existing Sprite cluster. No changes—other than adding a new machine type—were made to the native Sprite systems to accommodate the Sprite server.
4. *Performance*. Unfortunately, the server did not meet the performance goal. Assuming that we can apply the fixes that we know of or expect to see soon, the expected performance should be about 155 seconds for the Andrew benchmark, which is only 60% of native Sprite's performance. We believe that additional tuning could reduce the benchmark time even further, but this would probably require another 3–6 months of work.

7. Future work

As mentioned earlier in the paper, the Sprite server project has been discontinued. If development were to continue, the obvious first task would be to continue performance tuning. More work is needed to make the Sprite server perform adequately on the Andrew benchmark, and there are other benchmarks that the server should be tested on [7]. Furthermore, a production-quality Sprite server would require the remaining functionality mentioned in Sections 3 and 4.

Beside additional development, it would be useful to conduct research using the Sprite server. For example, it would be interesting to compare the performance of the Sprite server with the performance of a similar server that uses Mach IPC for remote device access or for process migration [13].

8. Conclusions

Porting Sprite to Mach was a mixed success. On the one hand, it greatly reduced the amount of machine-dependent code in Sprite, which should make Sprite much easier to port to new hardware. The asynchronous interfaces provided by Mach require some unpleasant complexity in the Sprite server, but this complexity is manageable. On the other hand, the server is almost unusably slow, even after a couple months of tuning, and it appears that much work is still needed to bring performance within striking distance of the native system.

At least one third of the performance gap results from the distributed nature of Sprite. However, the slowdown is not primarily due to RPC latency or throughput problems. Rather, it is due to the Sprite server's heavy use of an external pager, plus problems such as Sprite's inability to use mapped files to avoid copy overhead. The lesson here seems to be that there is more to high-performance distributed systems than fast communication, and although Mach shows promise as a general platform for distributed computing, it still has some serious shortcomings. For some problems (e.g., copy-on-write for external pagers) it should be possible to fix Mach, but in other cases (e.g., use of mapped files for performance), it may be necessary to redesign the distributed system instead.

If one asks whether it is worth porting an existing system to run on top of Mach, the answer seems to be "it depends." For a research system like Sprite, with its small development community, increased portability seems attractive enough to warrant a large one-time porting and tuning effort. On the other hand, a large commercial vendor, particularly a hardware vendor that would have to do its own Mach ports, would probably be better off to spend the time redesigning internal interfaces. There are other potential advantages to running on Mach, such as interoperability with different environments, but portability by itself seems inadequate justification to convert existing code to use Mach.

9. Availability

The sources for the Sprite server are available via anonymous ftp from `sprite.berkeley.edu` (aka `allspice.berkeley.edu`). Please note that the sources are intended only for browsing, as they may not build in a standard Mach environment, and the server will not run outside of a Sprite cluster.

Acknowledgments

This work was supported in part by the Open Software Foundation and in part by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under contract NAG2–591. SunSoft, Inc. provided the time and facilities for producing this paper. The Sprite group and the Mach community both provided valuable technical assistance during the project. M. Satyanarayanan developed the original Andrew benchmark. Brent Callaghan, Dejan Milojicic, John Ousterhout, and the conference referees provided many helpful comments on drafts of the paper. Special thanks to John Ousterhout for providing the modified benchmark and for technical assistance and guidance.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. "Measurements of a Distributed File System", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 198–212.

[2] Joseph Barrera III. private communication, September 1992.

[3] David L. Black *et al*. "Microkernel Operating System Architecture and Mach", *Proceedings of the USENIX Microkernel Workshop*, April 1992, 11–29.

[4] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter. "Binary Emulation of UNIX Using the V Kernel", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 73–86.

[5] Eric C. Cooper and Richard P. Draves. "C Threads", Technical report CMU–CS–88–154, Carnegie Mellon University, February 1988.

[6] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software—Practice & Experience 21*, 8, August 1991.

[7] David Finkel, Robert E. Kinicki, Aju John, Bradford Nichols, and Somesh Rao. "Developing Benchmarks to Measure the Performance of the Mach Operating System", *Proceedings of the USENIX Mach Workshop*, October 1990, 83–100.

[8] Alessandro Forin, Joseph Barrera, and Richard Sanzi. "The Shared Memory Server", *Proceedings of the Winter 1989 USENIX Conference*, January 1989, 229–244.

[9] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. "UNIX as an Application Program", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 87–96.

[10] M. D. Hill *et al*. "Design Decisions in SPUR", *IEEE Computer 19*, 11, November 1986, 8–22.

[11] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. "Generalized Emulation Services for Mach 3.0—Overview, Experiences and Current Status", *Proceedings of the USENIX Mach Symposium*, November 1991, 13–26.

[12] Yousef A. Khalidi and Michael N. Nelson. "An Implementation of UNIX on an Object-oriented Operating System", *Proceedings of the Winter 1993 USENIX Conference*, January 1993, 469–480.

[13] Dejan S. Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese. "Task Migration on Top of the Mach Microkernel", *Proceedings of the Third USENIX Mach Symposium*, April 1993.

[14] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. "The Sprite Network Operating System", *IEEE Computer 21*, 2, February 1988, 23–36.

[15] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 247–256.

[16] Franklin Reynolds and Jeffrey Heller. "Kernel Support For Network Protocol Servers", *Proceedings of the USENIX Mach Symposium*, November 1991, 149–162.

[17] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 1–15.

[18] Ken W. Shirriff and John K. Ousterhout. "A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System", *Proceedings of the Winter 1992 USENIX Conference*, January 1992, 315–331.

[19] Brent Welch. "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System", PhD thesis, University of California, Berkeley, February 1990. Also available as Technical Report UCB/CSD 90/567.

[20] Michael Young *et al*. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987, 63–76.