10. Performance

Overview

This is not a course about performance analysis or about writing efficient programs, although it often touches on these topics. Both are much too large to be covered, even superficially, in a single lecture devoted to performance. There are many books on performance analysis[1] and a few on efficient programs[2].

Our goal in this handout is more modest: to explain how to take a system apart and understand its performance well enough for most practical purposes. The analysis is necessarily rather rough and ready, but nearly always a rough analysis is adequate, often it's the best you can do, and certainly it's much better than what you usually see, which is no analysis at all. Note that performance *analysis* is not the same as performance *measurement*, which is more common.

What is performance? The critical measures are *bandwidth* and *latency*. We neglect other aspects that are sometimes important: availability (discussed later when we deal with replication), connectivity (discussed later when we deal with switched networks), and storage capacity

When should you work on performance? When it's needed. Time spent speeding up parts of a program that are fast enough is time wasted, at least from any practical point of view. Also, the march of technology, also known as Moore's law, means that in 18 months a computer will cost the same but be twice as fast and have twice as much storage; in five years it will be ten times as big and fast. So it doesn't help to make your system twice as fast if it takes two years to do it; it's better to just wait. Of course it still might pay if you get the improvement on new machines as well, and if a 4 x speedup is needed.

How can you get performance? There are techniques for making things faster: better algorithms, fast paths for common cases, and concurrency. And there is methodology for figuring out where the time is going: analyze and measure the system to find the bottlenecks and the critical parameters that determine its performance, and keep doing so both as you improve it and when it's in service. As a rule, a rough back-of-the-envelope analysis is all you need. Putting in a lot of detail will be a lot of work, take a lot of time, and obscure the important points.

What is performance: bandwidth and latency

Bandwidth and latency are usually the important metrics. Bandwidth tells you how much work gets done per second (or per year), and latency tells you how long something takes from start to finish: to send a message, process a transaction, or referee a paper. In some contexts it's customary to call these things by different names: throughput and response time, or capacity and delay. The ideas are exactly the same.

[1] Try R. Jain, *The Art of Computer Systems Performance Analysis*, Wiley, 1991, 720 pp.
[2] The best one I know is J. Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982, 170 pp.

Here are some examples of communication bandwidth and latency on a single link.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*
Alpha chip	on-chip bus	4	GB/s	2	ns	64
PC board	PCI I/O bus	266	MB/s	250	ns	32
Wires	Fibrechannel	125	MB/s	200	ns	1
	SCSI	20	MB/s	500	ns	16
LAN	Gigabit Ethernet	125	MB/s	100 +	µs	1
	Fast Ethernet	12.5	MB/s	100 +	µs	1
	Ethernet	1.25	MB/s	100 +	µs	1

Here are examples of communication bandwidth and latency through a switch that interconnects multiple links.

Medium	*Switch*	*Bandwidth*		*Latency*		*Links*
Alpha chip	register file	24	GB/s	2	ns	6
Wires	Cray T3E	122	GB/s	1	µs	2K
LAN	ATM switch	10	GB/s	10	µs	52
	Ethernet switch	40	MB/s	100–1200	µs	32
Copper pair	Central office	80	MB/s	125	µs	50K

Finally, here are some examples of other kinds of work, different from simple communication.

Medium	*Bandwidth*		*Latency*	
Disk	20	MB/s	10	ms
RPC on Giganet with VIA	30	calls/ms	30	µs
RPC	3	calls/ms	1	ms
Airline reservation transactions	3000	trans/s	1	sec
Published papers	20	papers/yr	2	years

Specs for performance

How can we put performance into our specs? In other words, how can we specify the amount of real time or other resources that an operation consumes? For resources like disk space that are controlled by the system, it's quite easy. Add a variable `spaceInUse` that records the amount of disk space in use, and to specify that an operation consumes no more than `max` space, write

```
<< VAR used: Space | used <= max => spaceInUse := spaceInUse + used >>
```

This is usually what you want, rather than saying exactly how much space is consumed, which would restrict the implementation too much.

Doing the same thing for real time is a bit trickier, since we don't usually think of the advance of real time as being under the control of the system. The spec, however, has to put a limit on how much time can pass before an operation is complete. Suppose we have a procedure `P`. We can specify `TimedP` that takes no more than `maxPLatency` to complete as follows. The variable `now` records the current time, and `deadlines` records a set of latest completion times for operations in progress. The thread `Clock` advances `now`, but not past a deadline. An operation like `TimedP` sets a deadline before it starts to run and clears it when it is done.

```
VAR now      : Time
    deadlines: SET Time

THREAD Clock() = DO now < deadlines.min => now + := 1 [] SKIP OD

PROC TimedP() = VAR t : Time
    << now < t /\ t < now + maxPLatency /\ ~ t IN deadlines =>
        deadlines := deadlines + {t} >>;
    P();
    << deadlines := deadlines - {t}; RET >>
```

This may seem like an odd way of doing things, but it does allow exactly the sequences of transitions that we want. The alternative is to construct `P` so that it completes within `maxPLatency`, but there's no straightforward way to do this.

Often we would like to write a probabilistic performance spec; for example, service time is drawn from a normal distribution with given mean and variance. There's no way to do this directly in Spec, because the underlying model of non-deterministic state machines has no notion of probability. The best we can do is to keep track of actual service times and declare a failure if they get too far from the desired form. Then you can interpret the spec to say: either the observed performance is a reasonably likely consequence of the desired distribution, or the system is malfunctioning.

How to get performance: Methodology

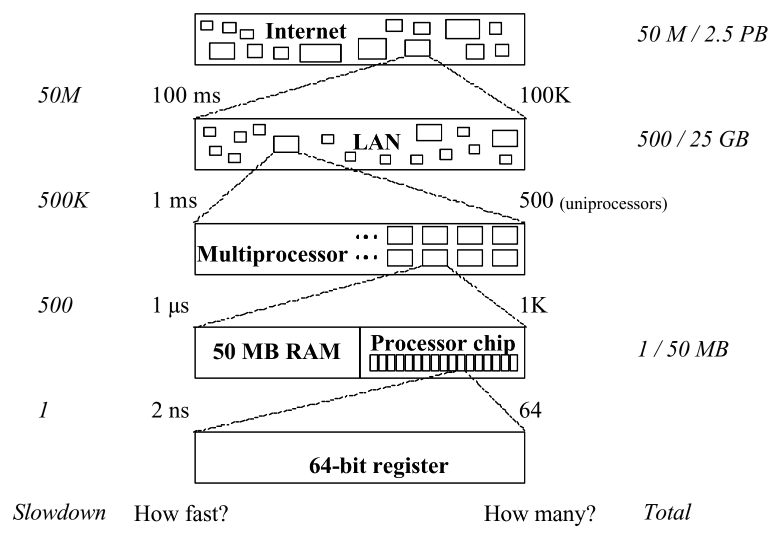
First you have to choose the right scale for looking at the system. Then you have to model or analyze the system, breaking it down into a few parts that add up to the whole, and measure the performance of the parts.

Choosing the scale

The first step in understanding the performance of a system is to find the right scale on which to analyze it. The figure shows the scales from the processor clock to an Internet access; there is a range of at least 50 million in speed and 50 million in quantity. Usually there is a scale that is the right one for understanding what's going on. For the performance of an inner loop it might be the system clock, for a simple transaction system the number of disk references, and for a Web browser the number of IP packets.

In practice, systems are not deterministic. Even if there isn't inherent non-determinism caused by unsynchronized clocks, the system is usually too complex to analyze in complete detail. The way to simplify it is to approximate. First find the right scale and the right primitives to count, ignoring all the fine detail. Then find the critical parameters that govern performance at that scale: number of RPC's per transaction, cache miss rate, clock ticks per instruction, or whatever. In this way you should be able to find a simple formula that comes within 20% of the observed performance, and usually this is plenty good enough.

For example, in the 1994 election DEC ran a Web server that provided data on the California election. It got about 80k hits/hour, or 20/sec, and it ran on a 200 MIPS machine. The data was



Scales of interconnection. Relative speed and size are in italics.

probably all in memory, so there were no disk references. A hit typically returns about 2 KB of data. So the cost was about 10M instructions/hit, or 5K instructions/byte returned. Clearly this was not an optimized system.

By comparison, a simple debit-credit transaction (the TPC-A benchmark) when carefully implemented does slightly more than two disk i/o's per transaction (these are to read and write per-account data that won't fit in memory). If carefully implemented it takes about 100K instructions. So on a 1000 MIPS machine it will consume 100 µs of compute time. Since two disk i/o's is 20 ms, it takes 200 disks to keep up with this CPU for this application.

As a third example, consider sorting 10 million 64 bit numbers; the numbers start on disk and must end up there, but you have room for the whole 80 MB in memory. So there's 160 MB of disk transfer plus the in-memory sort time, which is $n \log n$ comparisons and about half that many swaps. A single comparison and half swap might take 10 instructions with a good implementation of Quicksort, so this is a total of 10 * 10 M * 24 = 2.4 G instructions. Suppose the disk system can transfer 20 MB/sec and the processor runs at 500 MIPS. Then the total time is 8 sec for the disk plus 5 sec for the computing, or 13 sec, less any overlap you can get between the two phases. With considerable care this performance can be achieved. On a parallel machine you can do perhaps 30 times better.[3]

[3] Andrea Arpaci-Dusseau et al., High-performance sorting on networks of workstations. SigMod 97, Tucson, Arizona, May, 199, http://now.cs.berkeley.edu/NowSort/nowSort.ps .

Here are some examples of parameters that might determine the performance of a system to first order: cache hit rate, fragmentation, block size, message overhead, message latency, peak message bandwidth, working set size, ratio of disk reference time to message time.

Modeling

Once you have chosen the right scale, you have to break down the work at that scale into its component parts. The reason this is useful is the following principle:

> If a task x has parts a and b, the cost of x is the cost of a plus the cost of b, plus a system effect (caused by contention for resources) which is usually small.

Most people who have been to school in the last 15 years seem not to believe this. They think the system effect is so large that knowing the cost of a and b doesn't help at all in understanding the cost of x. But they are wrong. Your goal should be to break down the work into a small number of parts, between two and ten. Adding up the cost of the parts should give a result within 10% of the measured cost for the whole.

If it doesn't then either you got the parts wrong, or there actually is an important system effect. This is not common, but it does happen. Such effects are always caused by contention for resources, but this takes two rather different forms:

- *Thrashing* in a cache, because the sum of the working sets of the parts exceeds the size of the cache. The important parameter is the cache miss rate. If this is large, then the cache miss time and the working set are the things to look at. For example, SQL server on Windows NT running on a DEC Alpha 21164 in 1997 executes .25 instructions/cycle, even though the processor chip is capable of 2 instructions/cycle. The reason turns out to be that the instruction working set is much larger than the instruction cache, so that essentially every block of 4 instructions (16 bytes or one cache line) causes a cache miss, and the miss takes 64 ns, which is 16 4 ns cycles, or 4 cycles/instruction.
- *Clashing* or queuing for a resource that serves one customer at a time (unlike a cache, which can take away the resource before the customer is done). The important parameter is the queue length. It's important to realize that a resource need not be a physical object like a CPU, a memory block, a disk drive, or a printer. Any lock in the system is a resource on which queuing can occur. Typically the physical resources are instrumented so that it's fairly easy to find the contention, but this is often not true for locks. In the Alta Vista web search engine, for example, CPU and disk utilization were fairly low but the system was saturated. It turned out that queries were acquiring a lock and then page faulting; during the page fault time lots of other queries would pile up waiting for the lock and unable to make progress.

In the section on techniques we discuss how to analyze both of these situations.

Measuring

The basic strategy for measuring is to count the number of times things happen and observe how long they take. This can be done by sampling (what most profiling tools do) or by logging significant events such as procedure entries and exits. Once you have collected the data, you can use statistics or graphs to present it, or you can formulate a model of how it should be (for

example, time in this procedure is a linear function of the first parameter) and look for disagreements between the model and reality.[4] The latter technique is especially valuable for continuous monitoring of a running system. Without it, when a system starts performing badly in service it's very difficult to find out why.

Measurement is usually not useful without a model, because you don't know what to do with the data. Sometimes an appropriate model just jumps out at you when you look at raw profile data, but usually you have to think about it and try a few things.

How to get performance: Techniques

There are three main ways to make your program run faster: use a better algorithm, find a common case that can be made to run fast, or use concurrency to work on several things at once.

Algorithms

There are two interesting things about an algorithm: the 'complexity' and the 'constant factor'. An algorithm that works on n inputs can take roughly k (constant) time, or $k \log n$ (logarithmic), or $k\ n$ (linear), or $k\ n^2$ (quadratic), or $k\ 2^n$ (exponential). The k is the constant factor, and the function of n is the complexity. Usually these are 'asymptotic' results, which means that their percentage error gets smaller as n gets bigger. Often a mathematical analysis gives a worst-case complexity; if what you care about is the average case, beware. Sometimes a 'randomized' algorithm that flips coins internally can make the average case overwhelmingly likely.

For practical purposes the difference between $k \log n$ time and constant time is not too important, since the range over which n varies is likely to be 10 to 1M, so that $\log n$ varies only from 3 to 20. This factor of 6 may be much less than the change in k when you change algorithms. Similarly, the difference between $k\ n$ and $k\ n \log n$ is usually not important. But the differences between constant and linear, between linear and quadratic, and between quadratic and exponential are very important. To sort a million numbers, for example, a quadratic insertion sort takes a trillion operations, while the $n \log n$ Quicksort takes only 20 million in the average case (unfortunately the worst case for Quicksort is also quadratic). On the other hand, if n is only 100, then the difference among the various complexities (except exponential) may be less important than the values of k.

Another striking example of the value of a better algorithm is 'multi-grid' methods for solving the n-body problem: lots of particles (atoms, molecules or asteroids) interacting according to some force law (electrostatics or gravity). By aggregating distant particles into a single virtual particle, these methods reduce the complexity from n^2 to $n \log n$, so that it is feasible to solve systems with millions of particles. This makes it practical to compute the behavior of complex chemical reactions, of currents flowing in an integrated circuit package, or of the solar system.

[4] See Perl and Weihl, Performance assertion checking. *Proc. 14th ACM Symposium on Operating Systems Principles*, Dec. 1993, pp 134-145.

Fast path

If you can find a common case, you can try to do it fast. Here are some examples.

Caching is the most important: memory, disk (virtual memory, database buffer pool), web cache, memo functions (also called 'dynamic programming'), ...

Receiving a message that is an expected ack or the next message in sequence.

Acquiring a lock when no one else holds it.

Normal arithmetic vs. overflow.

Inserting a node in a tree at a leaf, vs. splitting a node or rebalancing the tree.

Here is the basic analysis for a fast path.

1 = fast time, $1 << 1 + s$ = slow time, m = miss rate = probability of taking the slow path.

$\boxed{\texttt{t = time = 1 + m * s}}$

There are two ways to look at it:

The slowdown from the fast case. If $m = 1/s$ then $t = 2$, a 2 x slowdown.

The speedup from the slow case. If $m = .5$ then $t = s/2 + 1$, nearly a 2 x speedup,

The analysis of fast paths is most highly developed in the study of computer architecture.[5]

Batching has the same structure:

1 = unit cost, s = startup (per-batch) cost, b = batch size.

t = time = $(b + s) / b = 1 + s/b$. So b is like $1/m$.

Amdahl's law for concurrency (discussed below) also has the same structure.

Concurrency

Usually concurrency is used to increase bandwidth. Hence it is most interesting when there are many 'independent' requests, such as web queries or airline reservation transactions.

Using concurrency to reduce latency requires a parallel algorithm, and runs into Amdahl's law, which is another case of the fast path analysis in which the fast path is the part of the program that can run in parallel, and the slow path is the part that runs serially. The conclusion is the same: if you have 100 processors, then your program can run 100 times faster if it all runs in parallel, but if 1% of it runs serially then it can only run 50 times faster, and if half runs serially then it can only run twice as fast. Usually we take the slowdown view, because the ideal is that we are paying for all the processors and so every one should be fully utilized. Then a 99%

[5] Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1995. The second edition has a great deal of new material.

parallel / 1% serial program, which achieves a speedup of 50, is only half as fast as our ideal. You can see that it will be difficult to make efficient use of 100 processors on a single problem.

Returning to concurrency for bandwidth, there can be multiple identical resources or several distinct resources. In the former case the main issue is load balancing (there is also the cost of switching a task to a particular resource). The most common example is multiple disks. If the load if perfectly balanced, the i/o rate from n disks is n times the rate from one disk. The debit-credit example above showed how this can be important. Getting the load perfectly balanced is hard; in practice it usually requires measuring the system and moving work among the resources.

When the resources are distinct, we have a 'queuing network' in which jobs 'visit' different resources in some sequence, or in various sequences with different probabilities. Things get complicated very quickly, but the most important case is quite simple. Suppose there is a single resource and tasks arrive independently of each other ('Poisson arrivals'). If the resource can handle a single request in a service time s, and its utilization (the fraction of time it is busy) is u, then the average request gets handled in a response time

$\boxed{\texttt{r = s/(1 - u)}}$

The reason is that a new request needs to get s amount of service before it's done, but the resource is only free for 1 - u of the time. For example, if u = .9, only 10% of the time is free, so it takes 10 seconds of real time to accumulate 1 second of free time.

Look at the slowdowns for different utilizations.

2 x at 50%
10 x at 90%
Infinite at 100% ('saturation')

Note that this rule applies only for Poisson (memoryless or 'random' arrivals). At the opposite extreme, if you have periodic arrivals and the period is synchronized with the service time, then you can do pipelining, drop each request into a service slot that arrives soon after the request, and get $r = s$ with $u = 1$. One name for this is "systolic computing".

A high u has two costs: increased r, as we saw above, and increased sensitivity to changing load. Doubling the load when $u = .2$ only slows things down by 30%; doubling from $u = .8$ is a catastrophe. High u is OK if you can tolerate increased r and you know the load. The latter could be because of predictability, for example, a perfectly scheduled pipeline. It could also be because of aggregation and statistics: there are enough random requests that the total load varies very little. Unfortunately, many loads are "bursty", which means that requests are more likely to follow other requests; this makes aggregation less effective.

When there are multiple requests, usually one is the bottleneck, the most heavily loaded component, and you only have to look at that one (of course, if you make it better then something else might become the bottleneck).

Servers with finite load

Many papers on queuing theory analyze a different situation, in which there is a fixed number of customers that alternate between thinking (for time z) and waiting for service (for the response time z). Suppose the system in steady state (also called 'equilibrium' or 'flow balance'), that is, the number of customers that demand service equals the number served, so that customers don't pile up in the server or drain out of it. You can find out a lot about the system by just counting the number of customers that pass by various points in the system

A customer is in the server if it has entered the server (requested service) and not yet come out (received all its service). If there are n customers in the server on the average and the throughput (customers served per second) is x, then the average time to serve a customer (the response time) must be $r = n/x$. This is "Little's law", usually written $n = rx$. It is obvious if the customers come out in the same order they come in, but true in any case. Here n is called the "queue length", though it includes the time the server is actually working as well.

If there are N customers altogether and each one is in a loop, thinking for z seconds before it enters the server, and the throughput is x as before, then we can use the same argument to compute the total time around the loop $r + z = N/x$. Solving for r we get $r = N/x - z$. This formula doesn't say anything about the service time s or the utilization u, but we also know that the throughput $x = u/s$ ($1/s$ degraded by the utilization). Plugging this into the equation for r we get $r = Ns/u - z$, which is quite different from the equation $r = s/(1 - u)$ that we had for the case of uniform arrivals. The reason for the difference is that the population is finite and hence the maximum number of customers that can be in the server is N.

Summary

Here are the most important points about performance.

- Moore's law: The performance of computer systems at constant cost doubles every 18 months, or increases by ten times every five years.
- To understand what a system is doing, first do a back-of-the-envelope calculation that takes account only of the most important one or two things, and then measure the system. The hard part is figuring out what the most important things are.
- If a task x has parts a and b, the cost of x is the cost of a plus the cost of b, plus a system effect (caused by contention for resources) which is usually small.
- The time for a task which has a fast path and a slow path is $1 + m * s$, where the fast path takes time 1, the slow path takes time $1 + s$, and the probability of taking the slow path is m (the miss rate). This formula works for batching as well, where the batch size is $1/m$.
- If a shared resource has service time s to serve one request and utilization u, and requests arrive independently of each other, then the response time is $s/(1 - u)$. It tends to infinity as u approaches 1.

.

Paper: Performance of Firefly RPC

Michael D. Schroeder and Michael Burrows[1]

Abstract

In this paper, we report on the performance of the remote procedure call implementation for the Firefly multiprocessor and analyze the implementation to account precisely for all measured latency. From the analysis and measurements, we estimate how much faster RPC could be if certain improvements were made.

The elapsed time for an inter-machine call to a remote procedure that accepts no arguments and produces no results is 2.66 milliseconds. The elapsed time for an RPC that has a single 1440-byte result (the maximum result that will fit in a single packet) is 6.35 milliseconds. Maximum inter-machine throughput of application program data using RPC is 4.65 megabits/second, achieved with 4 threads making parallel RPCs that return the Maximum sized result that fits in a single RPC result packet. CPU utilization at maximum throughput is about 1.2 CPU seconds per second on the calling machine and a little less on the server.

These measurements are for RPCs from user space on one machine to user space on another, using the installed system and a 10 megabit/second Ethernet. The RPC packet exchange protocol is built on IP/UDP, and the times include calculating and verifying UDP checksums. The Fireflies used in the tests had 5 MicroVAX II processors and a DEQNA Ethernet controller.

1. Introduction

Remote procedure call (RPC) is now a widely accepted method for encapsulating communication in a distributed system. With RPC, programmers of distributed applications need not concern themselves with the details of managing communications with another address space or another machine, nor with the detailed representation of operations and data items on the communication channel in use. RPC makes the communication with a remote environment look like a local procedure call (but with slightly different semantics).

In building a new software system for the Firefly multiprocessor [9], we decided to make RPC the primary communication paradigm, to be used by all future programs needing to communicate with another address space, whether on the same machine or a different one. Remote file transfers as well as calls to local operating systems entry points are handled via RPC. For RPC to succeed in this primary role, it must be fast enough that programmers are not tempted to design their own special purpose communication protocols. Because of the primary role of RPC

[1] Authors' address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301. A slightly different version of this paper appeared in *ACM Transactions on Computer Systems*, **8**, 1, February 1990.

however, we were able to structure the system software to expedite the handling of RPCs and to pay special attention to each instruction on the RPC "fast path".

This paper reports measurements of Firefly RPC performance for inter-machine calls. It also details the steps of the fast path and assigns an elapsed time to each step. Correspondence of the sum of these step times with the measured overall performance indicates that we have an accurate model of where the time is spent for RPC. In addition, this detailed understanding allows estimates to be made for the performance improvements that would result from certain changes to hardware and software.

Good inter-machine RPC performance is important for good performance of distributed applications. When RPC is also used for obtaining services from other programs running on the same machine (including the local operating system), good same-machine performance is also important. For the system measured here, the performance of same-machine RPC is adequate; the minimal same-machine RPC takes 937 microseconds of elapsed time, about 60 times the latency of a local procedure call. We have paid some attention to same-machine performance. For example, a special path through the scheduler is used to minimize the cost of the two context switches. But same-machine RPC in our system has not been as thoroughly worked over as inter-machine RPC. Much better performance for same-machine RPC is possible, as demonstrated by Bershad *et al.* [1], who achieve 20 times the latency of a local procedure call. Fortunately, their scheme for fast same-machine RPC and our scheme for fast inter-machine RPC fit together well in the same system.

1.1 Hardware and system characteristics

The Firefly multiprocessor allows multiple VAX processors access to a shared memory system via coherent caches. The Firefly version measured here had 16 megabytes of memory and 5 MicroVAX II CPUs [9], each of which provides about 1 MIPS of processor power[2]. One of these processors is also attached to a QBus I/O bus [5]. Network access is via a DEQNA device controller [4] connecting the QBus to a 10 megabit/second Ethernet. In the Firefly, the DEQNA has access to about 16 megabits/second of QBus bandwidth.

The Firefly system kernel, called the Nub, implements a scheduler, a virtual memory manager, and device drivers. The Nub executes in VAX kernel mode. The virtual memory manager provides multiple user address spaces for application programs, one of which contains the rest of the operating system. The scheduler provides multiple threads per address space, so that the Nub, operating system, and application programs can be written as true concurrent programs that execute simultaneously on multiple processors. The system is structured to operate best with multiple processors.

1.2 Overview of RPC structure

The Firefly RPC implementation follows the standard practice of using stub procedures [2]. The caller stub, automatically generated from a Modula-2+ [8] interface definition, is included in the calling program to provide local surrogates for the actual remote procedures. When a procedure

[2] Since the measurements reported here were made, Fireflies have been upgraded with faster CVAX processors and more memory.

in this stub is called, it allocates and prepares a call packet into which are marshaled the interface and procedure identifications, and the arguments. The stub calls the appropriate transport mechanism to send the call packet to the remote server machine and then blocks, waiting for a corresponding result packet. (Other threads in the caller address space are still able to execute.) When the result packet arrives, the stub unmarshals any results, frees the packet, and returns control to the calling program, as though the call had taken place within the same address space.

Similar machinery operates on the server. A server stub is included in the server program. This stub receives calls from the transport mechanism on the server machine when a suitable call packet arrives. The stub unmarshals the arguments and calls the identified procedure. After completing its task, the server procedure returns to the stub, which marshals the results and then calls the appropriate transport mechanism to send the result packet back to the caller machine.

A more detailed description of the structure of Firefly RPC appears in section 3.

2. Measurements

In this section, we report the overall performance of Firefly RPC. All measurements in this paper were made on the installed service system, software that was used by more than 50 researchers. Except where noted, all tests used automatically generated stubs for a remote `Test` interface that exports three procedures:

```
PROCEDURE: Null();
PROCEDURE: MaxResult(VAR OUT buf: ARRAY OF CHAR);
PROCEDURE: MaxArg(VAR IN buf: ARRAY OF CHAR);
```

`MaxArg` and `MaxResult` were called with the following variable as the `buf` argument:

```
VAR b: ARRAY [0..1439] OF CHAR;
```

Calls to `Null()` measure the base latency of the RPC mechanism. The Ethernet packets generated for the call and return of this procedure, which accepts no argument and produces no result, consist entirely of Ethernet, IP, UDP, and RPC headers and are the 74-byte minimum size generated for Ethernet RPC.

Calls to `MaxResult(b)` measure the server-to-caller throughput of RPC. The single 1440-byte `VAR OUT` argument produces the minimal 74-byte call packet and a result packet with 1514 bytes, the maximum packet size allowed on an Ethernet.[3] The `VAR OUT` designation tells the RPC implementation that the argument value need only be transferred in the result packet. `MaxArg(b)` moves data from caller to server in the same way. The `VAR IN` designation means that the argument value need only be transferred in the call packet.

[3] The Firefly RPC implementation allows arguments and results larger than 1440 bytes, but such calls are transmitted in multiple packets using a synchronous acknowledgment per packet. Thus the throughput of such calls is similar to the rate achieved by one thread calling `MaxArg(b)` or `MaxResult(b)` synchronously and is lower than several threads calling them in parallel.

2.1 Latency and throughput

As an overall assessment of RPC performance on the Firefly, we measured the elapsed time required to make a total of 10,000 RPCs using various numbers of caller threads. The caller threads ran in a user address space on one Firefly, and the multithreaded server ran in a user address space on another. Timings were done with the two Fireflies attached to a private Ethernet to eliminate variance due to other network traffic.

Table I: Time for 10,000 RPCs

# of caller	Calls to Null()		Calls to MaxResult(b)	
threads	seconds	RPCs/sec	seconds	Mbits/sec
1	26.61	375	63.47	1.82
2	16.80	595	35.28	3.28
3	16.26	615	27.28	4.25
4	15.45	647	24.93	4.65
5	15.11	662	24.69	4.69
6	14.69	680	24.65	4.70
7	13.49	741	24.72	4.69
8	13.67	732	24.68	4.69

From Table I we see that the base latency of the Firefly RPC mechanism is about 2.66 milliseconds and that 7 threads can do about 740 calls of `Null()` per second. Latency for a call to `MaxResult(b)` is about 6.35 milliseconds and 4 threads can achieve a server-to-caller throughput of 4.65 megabits/second using this procedure. This characterizes the rate at which a multi-threaded calling program can transfer data from a multi-threaded server program using RPC. In our system, RPC is used for most bulk data transfer, including file transfer. We observed about 1.2 CPU seconds per second being used on the caller machine, slightly less on the server machine, to achieve this maximum throughput. The CPU utilization included the standard system background threads using about 0.15 CPU seconds per second on each Firefly.

2.2 Marshaling time

RPC stubs are automatically generated from a Modula-2+ definition module. The stubs are generated as Modula-2+ source, which is compiled by the normal compiler. The stubs can marshal most Modula-2+ data structures, including multilevel records and pointer-linked structures. For most argument and result types, the stub contains direct assignment statements to copy the argument or result to/from the call or result packet. Some complex types are marshaled by calling library marshaling procedures.

We measured the following times for passing various argument types with the automatically generated stubs. The measurements reported are the incremental elapsed time for calling a procedure with the indicated arguments over calling `Null()`. The differences were measured for calls to another address space on the same machine in order to factor out the Ethernet transmission time for different sizes of call and result packets. Such same-machine RPC uses the same stubs as inter-machine RPC. Only the transport mechanism is different: shared memory rather than IP/UDP and the Ethernet. Because the pool of packet buffers (the same pool used for

Ethernet transport) is mapped into each user address space, the time for local transport is independent of packet size.

Table II: 4-byte integer arguments, passed by value

# of arguments	Marshaling time in µ-seconds
1	8
2	16
4	32

Integer and other fixed-size arguments passed by value are copied from the caller's stack into the call packet by the caller stub, and then copied from the packet to the server's stack by the server stub. Such arguments are not included in the result packet.

Table III: Fixed length array, passed by VAR OUT

Array size in bytes	Marshaling time in µ-seconds
4	20
400	140

Table IV: Variable length array, passed by VAR OUT

Array size in bytes	Marshaling time in µ-seconds
1	115
1440	550

In Modula-2+, `VAR` arguments are passed by address. The additional `OUT` or `IN` designation tells the stub compiler that the argument is being passed in one direction only. The stub can use this information to avoid transporting and copying the argument twice. A `VAR OUT` argument is actually a result and is transported only in the result packet; it is not copied into the call packet by the caller stub. If this result fits in a single packet then the server stub passes the address of storage for the result in the result packet buffer to the server procedure, from where the server procedure can directly write it, so no copy is performed at the server. The single copy occurs upon return when the caller stub moves the value in the result packet back into the caller's argument variable. `VAR IN` arguments work the same way, *mutatis mutandis*, to transfer data from caller to server. `VAR OUT` and `VAR IN` arguments of the same type have the same incremental marshaling costs. For single-packet calls and results, the marshaling times for array arguments scale linearly with the values reported in Tables III and IV.

Table V: Text.T argument

Array size in bytes	Marshaling time in µ-seconds
NIL	89
1	378
128	659

Table V illustrates how much slower marshaling can be when storage allocation is required and library procedures must be called. A `Text.T` is a text string that is allocated in garbage collected storage and is immutable. The caller stub must copy the string into the call packet. The server stub must allocate a new `Text.T` from garbage collected storage at the server, copy the string into it, and then pass a reference to this new object to the server procedure. Stubs manipulate `Text.Ts` by calling library procedures of the `Text` package.

3. Analysis

In this section we account for the elapsed time measured in section 2.1. We start by describing in some detail the steps in doing an inter-machine RPC. Then we report the time each step takes and compare the total for the steps to the measured performance.

3.1 Steps in a remote procedure call

The description here corresponds to the fast path of RPC. The fast path usually will be followed when other calls from a caller address space to the same remote server address space have occurred recently, within a few seconds, so that server threads are waiting for calls from this caller. Part of making RPC fast is arranging that the machinery for retransmission, for having enough server threads waiting, for multi-packet calls or results, for acknowledgments, and other features of the complete RPC mechanism intrude very little on the fast path. Consequently, the description of the fast path can ignore these mechanisms. The fast path is followed for more than 95% of RPCs that occur in the operational system.

Firefly RPC allows choosing from several different transport mechanisms at RPC bind time. Our system currently supports transport to another machine by a custom RPC packet exchange protocol layered on IP/UDP, transport to another machine by DECNet byte streams, and transport to another address space on the same machine by shared memory. The choice of transport mechanism is embodied in the particular versions of the transport procedures named `Starter`, `Transporter`, and `Ender` that are invoked by the caller stub. At the server, the choice is represented by the Receiver procedure being used. In this paper, we measure and describe the first of these transport options, using Ethernet. This custom RPC packet exchange protocol follows closely the design described by Birrell and Nelson for Cedar RPC [2]. The protocol uses implicit acknowledgments in the fast path cases.

3.1.1 Caller stub

When a program calls a procedure in a remote interface, control transfers to a caller stub module for that interface in the caller's address space. Assuming that binding to a suitable remote instance of the interface has already occurred, the stub module completes the RPC in five steps:

1. Call the `Starter` procedure to obtain a packet buffer for the call with a partially filled-in header.
2. Marshal the caller's arguments by copying them into the call packet.
3. Call the `Transporter` procedure to transmit the call packet and wait for the corresponding result packet.
4. Unmarshal the result packet by copying packet data to the caller's result variables.
5. Call the `Ender` procedure to return the result packet to the free pool.

When the stub returns control to the calling program, the results are available as if the call had been to a local procedure.

3.1.2 Server stub

The server stub has a similar job to do. When it receives a call packet on an up call from the `Receiver` procedure on the server machine, it performs three steps:

1. Unmarshal the call's arguments from the call packet. Depending on its type, an argument may be copied into a local stack variable, copied into newly allocated garbage collected storage, or left in the packet and its address passed. The call packet is not freed.
2. Call the server procedure.
3. When the server procedure returns, marshal the results in the saved call packet, which becomes the result packet.

When the server stub returns to the `Receiver` procedure, the result packet is transmitted back to the caller. The server thread then waits for another call packet to arrive.

3.1.3 Transport mechanism

The `Transporter` procedure must fill in the RPC header in the call packet. It then calls the `Sender` procedure to fill in the UDP, IP, and Ethernet headers, including the UDP checksum on the packet contents. To queue the call packet for transmission to the server machine, the `Sender` invokes the Ethernet driver by trapping to the Nub in kernel mode,.

Because the Firefly is a multiprocessor with only CPU 0 connected to the I/O bus, the Ethernet driver must run on CPU 0 when notifying the Ethernet controller hardware. Control gets to CPU 0 through an interprocessor interrupt; the CPU 0 interrupt routine prods the controller into action.

Immediately after issuing the interprocessor interrupt, the caller thread returns to the caller's address space where the `Transporter` registers the outstanding call in an RPC call table, and then waits on a condition variable for the result packet to arrive. The time for these steps is not part of the fast path latency as the steps are overlapped with the transmission of the call packet, the processing at the server, and the transmission of the result packet. For the RPC fast path, the calling thread gets the call registered before the result packet arrives.

Once prodded, the Ethernet controller reads the packet from memory over the QBus and then transmits it to the controller on the server machine. After receiving the entire packet, the server

controller writes the packet to memory over the server QBus and then issues a packet arrival interrupt.

The Ethernet interrupt routine validates the various headers in the received packet, verifies the UDP checksum, and then attempts to hand the packet directly to a waiting server thread. Such server threads are registered in the call table of the server machine. If the interrupt routine can find a server thread associated with this caller address space and called address space, it attaches the buffer containing the call packet to the call table entry and awakens the server thread directly.

The server thread awakens in the server's `Receiver` procedure.[4] The `Receiver` inspects the RPC header and then calls the stub for the interface ID specified in the call packet. The interface stub then calls the specific procedure stub for the procedure ID specified in the call packet.

The transport of the result packet over the Ethernet is handled much the same way. When the server stub returns to the `Receiver`, it calls the server's `Sender` procedure to transmit the result packet back to the caller machine. Once the result packet is queued for transmission, the server thread returns to the `Receiver` and again registers itself in the call table and waits for another call packet to arrive.

Back at the caller machine, the Ethernet interrupt routine validates the arriving result packet, does the UDP checksum, and tries to find the caller thread waiting in the call table. If successful, the interrupt routine directly awakens the caller thread, which returns to step 4 in the caller stub described above.

The steps involved in transporting a call packet and a result packet are nearly identical, from calling the `Sender` through transmitting and receiving the packet to awakening a suitable thread in the call table. We refer to these steps as the "send+receive" operation. A complete remote procedure call requires two send+receives -- one for the call packet and one for the result packet.

3.2 Structuring for low latency

The scenario just outlined for the fast path of RPC incorporates several design features that lower latency. We already mentioned that the stubs use custom generated assignment statements in most cases to marshal arguments and results for each procedure, rather than library procedures or an interpreter. Another performance enhancement in the caller stub is invoking the chosen `Starter`, `Transporter`, and `Ender` procedures through procedure variables filled in at binding time, rather than finding the procedures by a table lookup.

Directly awakening a suitable thread from the Ethernet interrupt routine is another important performance optimization for RPC. This approach means that demultiplexing of RPC packets is done in the interrupt routine. The more traditional approach is to have the interrupt handler awaken an operating system thread to demultiplex the incoming packet. The traditional approach lowers the amount of processing in the interrupt handler, but doubles the number of wakeups required for an RPC. By carefully coding the demultiplexing code for RPC packets, the time per packet in the interrupt handler can be kept within reasonable bounds (see Table VI). Even with only two wakeups for each RPC, the time to do these wakeups can be a major contributor to RPC

[4] We lump three procedures of the actual implementation under the name `Receiver` here.

latency. Considerable work has been done on the Firefly scheduler to minimize this cost. The slower path through the operating system address space is used when the interrupt routine cannot find the appropriate RPC thread in the call table, when it encounters a lock conflict in the call table, or when it handles a non-RPC packet. The latency of an RPC to `Null()` is about 4.5 milliseconds when the slower path through the operating system is followed for both the call and the result packet.

The packet buffer management scheme we have adopted also increases RPC performance. We mentioned above that the server stub retains the call packet to use it for the results. We also arrange for the receive interrupt handler to immediately replace the buffer used by an arriving call or result packet. Each call table entry occupied by a waiting thread also contains a packet buffer. In the case of a calling thread it is the call packet; in the case of a server thread it is the last result packet. These packets must be retained for possible retransmission. The RPC packet exchange protocol is arranged so that arrival of a result or call packet means that the packet buffer in the matching call table entry is no longer needed. Thus, when putting the newly arrived packet into the call table, the interrupt handler removes the buffer found in that call table entry and adds it to the Ethernet controller's receive queue. Since the interrupt handler always checks for additional packets to process before terminating, on-the-fly receive buffer replacement can allow many packets to be processed per interrupt. Recycling is sufficiently fast that we have seen several hundred packets processed in a single receive interrupt.

The alert reader will have suspected another feature of our buffer management strategy: RPC packet buffers reside in memory shared among all user address spaces and the Nub. These buffers also are permanently mapped into VAX I/O space. Thus, RPC stubs in user spaces, and the Ethernet driver code and interrupt handler in the Nub, all can read and write packet buffers in memory using the same addresses. This strategy eliminates the need for extra address mapping operations or copying when doing RPC. While its insecurity makes shared buffers unsuitable for use in a time sharing system, security is acceptable for a single user workstation or for a server where only trusted code executes (say a file server). This technique would also work for, say, kernel to kernel RPCs. For user space to user space RPCs in a time sharing environment, the more secure buffer management required would introduce extra mapping or copying operations into RPCs.

Like the pool of RPC buffers, the RPC call table also is shared among all user address spaces and the Nub. The shared call table allows the interrupt handler to find and awaken the waiting (calling or server) thread in any user address space.

Several of the structural features used to improve RPC performance collapse layers of abstraction. Programming a fast RPC is not for the squeamish.

3.3 Allocation of latency

We now try to account for the time an RPC takes.

Table VI shows a breakdown of time for the send+receive operation that is executed twice per RPC: once for the argument packet, once for the result packet. The first seven actions are activities of the sending machine. The next three are Ethernet and hardware controller delays. The last four are actions performed by the receiving machine. All of the software in this table is

written in assembly language. Table VI shows that Ethernet transmission time and QBus/controller latency are dominant for large packets, but software costs are dominant for small packets. The biggest single software cost is the scheduler operation to awaken the waiting RPC thread.

Table VI: Latency of steps in the send+receive operation

Action	µsecs for 74 byte packet		µsecs for 1514 byte packet (if different)	
Finish UDP header (Sender)	59	**a**		
Calculate UDP checksum	45	**b**	440	**b**
Handle trap to Nub	37	**a**		
Queue packet for transmission	39	**a**		
Interprocessor interrupt to CPU 0	10	**c**		
Handle interprocessor interrupt	76	**a**		
Prod Ethernet controller	22	**a**		
QBus/Controller transmit latency	70	**d**	815	**e**
Transmission time on Ethernet	60	**d**	1230	**e**
QBus/Controller receive latency	80	**d**	835	**e**
General I/O interrupt handler	14	**a**		
Handle interrupt for received pkt	177	**a**		
Calculate UDP checksum	45	**b**	440	**b**
Wakeup RPC thread	220	**a**		
Total for send+receive	954		4414	

Key for table VI:

a Calculated by adding the measured execution times of the machine instructions in this code sequence.
b Measured by disabling UDP checksums and noting speedup.
c Estimated.
d Measured with logic analyzer.
e Measurements d adjusted assuming 10 megabit/sec Ethernet, 16 megabit/sec QBus transfer, and no cut through.

Table VII shows where time is spent in the user space RPC runtime and stubs for a call to `Null()`. The procedures detailed in Table VII are written in Modula-2+. The times were calculated by adding up the instruction timings for the compiler-generated code. The Modula-2+ code listed in Table VII includes 9 procedure calls. Since each call/return takes about 15 microseconds, depending on the number of arguments, about 20% of the time here is spent in the calling sequence.

In Table VIII we combine the numbers presented so far to account for the time required for a complete call of `Null()` and of `MaxResult(b)`.

Table VII: Latency of stubs and RPC runtime

Machine	Procedure	μ-seconds
Caller	Calling program (loop to repeat call)	16
	Calling stub (call & return)	90
	Starter	128
	Transporter (send call pkt)	27
Server	Receiver (receive call pkt)	158
	Server stub (call & return)	68
	Null (the server procedure)	10
	Receiver (send result pkt)	27
Caller	Transporter (receive result pkt)	49
	Ender	33
	Total	606

Table VIII: Calculation of latency for RPC to Null() and MaxResult(b)

Procedure	Action	μ-seconds
Null()	Caller, server, stubs, RPC runtime	606
	Send+receive 74-byte call packet	954
	Send+receive 74-byte result packet	954
	Total	2514
MaxResult(b)	Caller, server, stubs, RPC runtime	606
	Marshal 1440-byte result packet	550
	Send+receive 74-byte call packet	954
	Send+receive 1514-byte result pkt	4414
	Total	6524

The best measured total latency for a call to `Null()` is 2645 microseconds, so we've failed to account for 131 microseconds. The best measured total latency for a call to `MaxResult(b)` is 6347 microseconds, so we've accounted for 177 microseconds too much. By adding the time of each instruction executed and of each hardware latency encountered, we have accounted for the total measured time of RPCs to `Null()` and `MaxResult(b)` to within about 5% .

4. Improvements

One of the important steps in improving the performance of Firefly RPC over its initial implementation was to rewrite the Modula-2+ versions of the fast path code in the Ethernet send+receive operation in VAX assembly language. In this section, we illustrate the speed-ups achieved by using machine code.

We then use the analysis and measurement reported so far to estimate the impact that other changes could have on overall RPC performance. It is hard to judge how noticeable these possible improvements would be to the normal user of the system. The Firefly RPC implementation has speeded up by a factor of three or so from its initial implementation. This improvement has produced no perceived change in the behavior of most applications. However, lower latency RPC may encourage programmers to use RPC interfaces where they might previously have been tempted to use *ad hoc* protocols before, and encourage the designers of new systems to make extensive use of RPC.

4.1 Assembly language vs. Modula-2+

In order to give some idea of the improvement obtained when Modula-2+ code fragments are recoded in assembly language, the following table shows the time taken by one particular code fragment at various stages of optimization. This fragment was chosen because it was the largest one that was recoded and is typical of the improvements obtained for all the code that was rewritten.

Table IX: Execution time for main path of the Ethernet interrupt routine

Version μ-seconds	Time in
Original Modula-2+	758
Best Modula-2+	547
Assembly language	177

The "Original Modula-2+" was the state of the interrupt routine before any assembly language code was written. The "Best Modula-2+" code was structured so that the compiler output would follow the assembly language version as closely as possible. The "Assembly language" version, however, runs in one third the time. Writing in assembly language is hard work and also makes the programs harder to maintain. Because RPC is the preferred communication paradigm for the Firefly, however, it seemed reasonable to concentrate considerable attention on its key code sequence. (There cannot be too much assembly language in the fast path, or it would not be fast!) The Modula-2+ compiler used here apparently generates fairly inefficient code. The speedup achieved by using assembly language will be less dramatic starting from a better compiler, but would still be substantial.

4.2 Speculations on future improvements

While improving the speed of the RPC system, we noticed several further possibilities for improving its performance and also considered the impact that faster hardware and networks would have. In this section, we speculate on the performance changes such improvements might generate. Some of these changes have associated disadvantages or require unusual hardware. For each change, we give the estimated speedup for a call to `Null()` and a call to `MaxResult(b)`. The effect on maximum throughput has not been estimated for all the changes, since this figure is likely to be limited by a single hardware component.

Some estimates are based on best conceivable figures, and these may ignore some practical issues. Also, the effects discussed are not always independent, so the performance improvement figures cannot always be added.

4.2.1 Different network controller

A controller which provided maximum conceivable overlap between Ethernet and Qbus transfers would save about 300 microseconds on `Null()` (11%), and about 1800 microseconds (28%) on `MaxResult(b)`. It is more difficult to estimate the improvement in throughput with multiple threads, since the Ethernet controller is already providing some overlap in that case. We think improvement is still possible on the transmission side, since the saturated reception rate is 40% higher than the corresponding transmission rate.

4.2.2 Faster network

If the network ran at 100 megabits/second and all other factors remained constant, the time to call `Null()` would be reduced by 110 microseconds (4%) and the time to call `MaxResult(b)` would be reduced by 1160 microseconds (18%).

4.2.3 Faster CPUs

If all processors were to increase their speed by a factor of 3, the time to call `Null()` would reduce by about 1380 microseconds (52%). The time to call `MaxResult(b)` would reduce by 2280 microseconds (36%).

4.2.4 Omit UDP checksums

Omitting UDP checksums saves 180 microseconds (7%) on a call to `Null()` and 1000 microseconds (16%) on a call to `MaxResult(b)`. At present, we use these end-to-end software checksums because the Ethernet controller occasionally makes errors after checking the Ethernet CRC. End-to-end checksums still would be essential for crossing gateways in an internet.

4.2.5 Redesign RPC protocol

We estimate that by redesigning the RPC packet header to make it easy to interpret, and changing an internal hash function, it would be possible to save about 200 microseconds per RPC. This represents approximately 8% of a call to `Null()` and 3% of a call to `MaxResult(b)`.

4.2.6 Omit layering on IP and UDP

We estimate that direct use of Ethernet datagrams, omitting the IP and UDP headers, would save about 100 microseconds per RPC, assuming that checksums were still calculated. This is about 4% of a call to `Null()` and 1-2% of a call to `MaxResult(b)`. This change would make it considerably more difficult to implement RPC on machines where we do not have access to the kernel. It would also make it impossible to use RPC via an IP gateway. (Some of the fields in IP and UDP headers would have to be incorporated into the RPC header.)

4.2.7 Busy wait

If caller and server threads were to loop in user space while waiting for incoming packets, the time for a wakeup via the Nub would be saved at each end. This is about 440 microseconds per RPC, which is 17% of a call to `Null()` and 7% of a call to `MaxResult(b)`. Allowing threads to busy wait (in such a way that they would relinquish control whenever the scheduler demanded) would require changes to the scheduler and would make it difficult to measure accurately CPU usage for a thread.

4.2.8 Recode RPC runtime routines (except stubs)

If the RPC runtime routines in Table VII were rewritten in hand-generated machine code, we would expect to save approximately 280 microseconds per RPC. This corresponds to 10% of a call to `Null()` and 4% of a call to `MaxResult(b)`. This figure is based on an expected speedup of a factor of 3 in 422 microseconds of routines to be recoded, which is typical of other code fragments that have been rewritten.

5. Fewer processors

The Fireflies used in the tests reported here had 5 MicroVAX II processors. The measurements in other sections were done with all 5 available to the scheduler. In this section, we report the performance with fewer available processors.

At first we were unable to get reasonable performance when running with a single available processor on the caller and server machines. Calls to `Null()` were taking around 20 milliseconds. We finally discovered the cause to be a few lines of code that slightly improved multiprocessor performance but had a dramatic negative effect on uniprocessor performance. The good multiprocessor code tends to lose about 1 packet/second when a single thread calls `Null()` using uniprocessors, producing a penalty of about 600 milliseconds waiting for a retransmission to occur. Fixing the problem requires swapping the order of a few statements at a penalty of about 100 microseconds for multiprocessor latency. The results reported in this section were measured with the swapped lines installed. (This change was not present for results reported in other sections.)

These measurements were taken with the RPC Exerciser, which uses hand-produced stubs that run faster than the standard ones (because they don't do marshaling, for one thing). With the RPC Exerciser, the latency for `Null()` is 140 microseconds faster and the latency for `MaxResult(b)` is 600 microseconds faster than reported in Table I. Such hand-produced stubs might be used in performance-sensitive situations, such as kernel-to-kernel RPCs, where one could trust the caller and server code to reference all arguments and results directly in RPC packet buffers.

Table X: Calls to Null() from one thread with varying numbers of processors

caller processors	server processors	seconds for 1,000 calls
5	5	2.69
4	5	2.73
3	5	2.85
2	5	2.98
1	5	3.96
1	4	3.98
1	3	4.13
1	2	4.21
1	1	4.81

Table X shows 1 thread making RPCs to `Null()`, with varying numbers of processors available on each machine. When the calls are being done one at a time from a single thread, reducing the number of caller processors from 5 down to 2 increases latency only about 10%. There is a sharp jump in latency for the uniprocessor caller. Reductions in server processors seem to follow a similar pattern. Latency with uniprocessor caller and server machines is 75% longer than for 5 processor machines.

Table XI: Throughput in megabits/second of MaxResult(b) with varying numbers of processors and threads

caller processors	5	1	1
server processors	5	5	1
1 caller thread	2.0	1.5	1.3
2 caller threads	3.4	2.3	2.0
3 caller threads	4.6	2.7	2.4
4 caller threads	4.7	2.7	2.5
5 caller threads	4.7	2.7	2.5

Table XI shows the effect on the data transfer rate of varying the number of processors on RPCs to `MaxResult(b)`. In this test each thread made 1,000 calls. RPC throughput in this system is quite sensitive to the difference between a uniprocessor and a multiprocessor. Uniprocessor throughput is slightly more than half of 5 processor performance for the same number of caller threads.

We have not tried very hard to make Firefly RPC perform well on a uniprocessor machine. The fast path for RPC is followed exactly only on a multiprocessor. On a uniprocessor, extra code gets included in the basic latency for RPC, such as a longer path through the scheduler. Also, the fast path is abandoned more often by lock conflicts on a uniprocessor. Even with just one caller thread, the interrupt code can require a lock held by the caller or server thread. It is plausible that better uniprocessor throughput could be achieved by an RPC design, like Amoeba's [7], V's [3], or Sprite's [6], that streamed a large argument or result from a single call in multiple packets, rather than depended on multiple threads transferring just a packet's worth of data per call. The streaming strategy requires fewer thread-to-thread context switches.

6. Other systems

A sure sign of the coming of age of RPC is that others are beginning to report RPC performance in papers on distributed systems. Indeed, an informal competition has developed to achieve low latency and high throughput. Table XII collects the published performance of several systems of interest. All of the latency measurements were for inter-machine RPCs to the equivalent of `Null()`. The throughput measurements were made using either single-threaded multipacket calls or multithreaded single-packet calls, depending on which worked best on a particular system. All measurements were for a 10 megabit Ethernet, with the exception that the Cedar measurements used a 3 megabit Ethernet. No other paper has attempted to account exactly for the measured performance, as we have tried to do.

Table XII: Performance of remote RPC in other systems

System	Machine - Processor	~ MIPS	milliseconds /call	megabits /sec
Cedar[2]	Dorado - custom	1 x 4	1.1	2.0
Amoeba[7]	Tadpole - M68020	1 x 1.5	1.4	5.3
V[3]	Sun 3/75 - M68020	1 x 2	2.5	4.4
Sprite[6]	Sun 3/75 - M68020	1 x 2	2.8	5.6
Amoeba/Unix[7]	Sun 3/50 - M68020	1 x 1.5	7.0	1.8
Firefly	FF - MicroVAX II	1 x 1	4.8	2.5
Firefly	FF - MicroVAX II	5 x 1	2.7	4.6

Amoeba advertises itself as the world's fastest distributed system. But the Cedar system achieved 20% lower latency 4 years earlier (using a slower network and a faster processor). Determining a winner in the RPC sweepstakes is tricky business. These systems vary in processor speed, I/O bus bandwidth, and controller performance. Some of these RPC implementations work only kernel to kernel, others work user space to user space. Some protocols provide Internet headers, others work only within a single Ethernet. Some use automatically generated stubs, others use hand-produced stubs. Some generate end-to-end checksums with software, others do not. The implementations are written in different languages with varying quality compilers. Researchers can disagree about which corrections to apply to normalize the reported performance of different systems.

It is clear from the literature that developers of distributed systems are learning how to get good request/response performance from their systems. Many system developers and users now understand that it is not necessary to put up with high latency or low throughput from RPC-style communication. Some RPC implementations appear to drive current Ethernet controllers at their throughput limit[5] and to provide basic remote call latency only about 100 times slower than that for statically-linked calls within a single address space.

[5] In the case of Firefly RPC, throughput has remained the same as the last few performance improvements were put in place. The CPU utilization continued to drop as the code got faster.

7. Conclusions

Our objective in making Firefly RPC the primary communication mechanism between address spaces, both inter-machine and same-machine, was to explore the bounds of effectiveness of this paradigm. In making the RPC implementation fast, we sought to remove one excuse for not using it. To make it fast we had to understand exactly where time was being spent, remove unnecessary steps from the critical path, give up some structural elegance, and write key steps in hand-generated assembly code. We did not find it necessary to sacrifice function; RPC still allows multiple transports, works over wide area networks, copes with lost packets, handles a large variety of argument types including references to garbage collected storage, and contains the structural hooks for authenticated and secure calls. The performance of Firefly RPC is now good enough that programmers accept it as the standard way to communicate.

8. Acknowledgments

Andrew Birrell designed the Firefly RPC facility and Sheng Yang Chiu designed the driver for the Ethernet controller. Andrew and Sheng Yang, along with Michael Burrows, Eric Cooper, Ed Lazowska, Sape Mullender, Michael Schroeder, and Ted Wobber have participated in the implementation and improvement of the facility. The recent round of performance improvements and measurements were done by Michael Burrows and Michael Schroeder, at Butler Lampson's insistence. The RPC latency includes two wakeup calls to the scheduler, whose design and implementation was done by Roy Levin. Andrew Birrell, Mark Brown, David Cheriton, Ed Lazowska, Hal Murray, John Ousterhout, Susan Owicki, and Doug Terry made several suggestions for improving the presentation of the paper.

9. References

1. Brian Bershad et al., Lightweight remote procedure call. *Proceedings of the 13th ACM Symposium on Operating System Principles*, Dec. 1989.
2. Andrew D. Birrell and Bruce Nelson, Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Feb. 1984, pp. 39-59.
3. David R. Cheriton, The V distributed system. *Communications of the ACM*, Mar. 1988, pp. 314-333.
4. Digital Equipment Corp., *DEQNA Ethernet -- User's Guide*, Sep. 1986.
5. Digital Equipment Corp., *Microsystems Handbook*, 1985, Appendix A.
6 John K. Ousterhout et al., The Sprite network operating system. *Computer*, Feb. 1988, pp. 23-35.
7. Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum, Performance of the world's fastest distributed operating system. *Operating Systems Review*, Oct. 1988, pp. 25-34.
8 Paul R. Rovner, Extending Modula-2 to build large, integrated systems. *IEEE Software*, Nov. 1986, pp. 46-57.
9. C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite Jr., Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, Aug. 1988, pp. 909-920.