

lmbench: Portable tools for performance analysis

Larry McVoy

Silicon Graphics, Inc.

Carl Staelin

Hewlett-Packard Laboratories

Abstract

lmbench is a micro-benchmark suite designed to focus attention on the basic building blocks of many common system applications, such as databases, simulations, software development, and networking. In almost all cases, the individual tests are the result of analysis and isolation of a customer's actual performance problem. These tools can be, and currently are, used to compare different system implementations from different vendors. In several cases, the benchmarks have uncovered previously unknown bugs and design flaws. The results have shown a strong correlation between memory system performance and overall performance. lmbench includes an extensible database of results from systems current as of late 1995.

1. Introduction

lmbench provides a suite of benchmarks that attempt to measure the most commonly found performance bottlenecks in a wide range of system applications. These bottlenecks have been identified, isolated, and reproduced in a set of small micro-benchmarks, which measure system latency and bandwidth of data movement among the processor and memory, network, file system, and disk. The intent is to produce numbers that real applications can reproduce, rather than the frequently quoted and somewhat less reproducible marketing performance numbers.

The benchmarks focus on latency and bandwidth because performance issues are usually caused by latency problems, bandwidth problems, or some combination of the two. Each benchmark exists because it captures some unique performance problem present in one or more important applications. For example, the TCP latency benchmark is an accurate predictor of the Oracle distributed lock manager's performance, the memory latency benchmark gives a strong indication of Verilog simulation performance, and the file system latency benchmark models a critical path in software development.

lmbench was developed to identify and evaluate system performance bottlenecks present in many machines in 1993-1995. It is entirely possible that computer architectures will have changed and advanced enough in the next few years to render parts

of this benchmark suite obsolete or irrelevant.

lmbench is already in widespread use at many sites by both end users and system designers. In some cases, lmbench has provided the data necessary to discover and correct critical performance problems that might have gone unnoticed. lmbench uncovered a problem in Sun's memory management software that made all pages map to the same location in the cache, effectively turning a 512 kilobyte (K) cache into a 4K cache.

lmbench measures only a system's ability to transfer data between processor, cache, memory, network, and disk. It does not measure other parts of the system, such as the graphics subsystem, nor is it a MIPS, MFLOPS, throughput, saturation, stress, graphics, or multiprocessor test suite. It is frequently run on multiprocessor (MP) systems to compare their performance against uniprocessor systems, but it does not take advantage of any multiprocessor features.

The benchmarks are written using standard, portable system interfaces and facilities commonly used by applications, so lmbench is portable and comparable over a wide set of Unix systems. lmbench has been run on AIX, BSDI, HP-UX, IRIX, Linux, FreeBSD, NetBSD, OSF/1, Solaris, and SunOS. Part of the suite has been run on Windows/NT as well.

lmbench is freely distributed under the Free Software Foundation's General Public License [Stallman89], with the additional restriction that results may be reported only if the benchmarks are unmodified.

2. Prior work

Benchmarking and performance analysis is not a new endeavor. There are too many other benchmark suites to list all of them here. We compare lmbench to a set of similar benchmarks.

- **I/O (disk) benchmarks:** IOstone [Park90] wants to be an I/O benchmark, but actually measures the memory subsystem; all of the tests fit easily in the cache. IObench [Wolman89] is a systematic file system and disk benchmark, but it is complicated and unwieldy. In [McVoy91] we reviewed many I/O benchmarks and found them all lacking because they took too long to run and were too complex a solution to a fairly simple

problem. We wrote a small, simple I/O benchmark, `lmd` that measures sequential and random I/O far faster than either `IOstone` or `IObench`. As part of [McVoy91] the results from `lmd` were checked against `IObench` (as well as some other Sun internal I/O benchmarks). `lmd` proved to be more accurate than any of the other benchmarks. At least one disk vendor routinely uses `lmd` to do performance testing of its disk drives.

Chen and Patterson [Chen93, Chen94] measure I/O performance under a variety of workloads that are automatically varied to test the range of the system's performance. Our efforts differ in that we are more interested in the CPU overhead of a single request, rather than the capacity of the system as a whole.

- **Berkeley Software Distribution's microbench suite:** The BSD effort generated an extensive set of test benchmarks to do regression testing (both quality and performance) of the BSD releases. We did not use this as a basis for our work (although we used ideas) for the following reasons: (a) missing tests — such as memory latency, (b) too many tests, the results tended to be obscured under a mountain of numbers, and (c) wrong copyright — we wanted the Free Software Foundation's General Public License.

- **Ousterhout's Operating System benchmark:** [Ousterhout90] proposes several system benchmarks to measure system call latency, context switch time, and file system performance. We used the same ideas as a basis for our work, while trying to go farther. We measured a more complete set of primitives, including some hardware measurements; went into greater depth on some of the tests, such as context switching; and went to great lengths to make the benchmark portable and extensible.

- **Networking benchmarks:** `Netperf` measures networking bandwidth and latency and was written by Rick Jones of Hewlett-Packard. `lmbench` includes a smaller, less complex benchmark that produces similar results.

`ttcp` is a widely used benchmark in the Internet community. Our version of the same benchmark routinely delivers bandwidth numbers that are within 2% of the numbers quoted by `ttcp`.

- **McCalpin's stream benchmark:** [McCalpin95] has memory bandwidth measurements and results for a large number of high-end systems. We did not use these because we discovered them only after we had results using our versions. We will probably include McCalpin's benchmarks in `lmbench` in the future.

In summary, we rolled our own because we wanted simple, portable benchmarks that accurately measured a wide variety of operations that we consider crucial to performance on today's systems. While portions of other benchmark suites include similar work, none includes all of it, few are as portable,

and almost all are far more complex. Less filling, tastes great.

3. Benchmarking notes

3.1. Sizing the benchmarks

The proper sizing of various benchmark parameters is crucial to ensure that the benchmark is measuring the right component of system performance. For example, memory-to-memory copy speeds are dramatically affected by the location of the data: if the size parameter is too small so the data is in a cache, then the performance may be as much as ten times faster than if the data is in memory. On the other hand, if the memory size parameter is too big so the data is paged to disk, then performance may be slowed to such an extent that the benchmark seems to 'never finish.'

`lmbench` takes the following approach to the cache and memory size issues:

- All of the benchmarks that could be affected by cache size are run in a loop, with increasing sizes (typically powers of two) until some maximum size is reached. The results may then be plotted to see where the benchmark no longer fits in the cache.

- The benchmark verifies that there is sufficient memory to run all of the benchmarks in main memory. A small test program allocates as much memory as it can, clears the memory, and then strides through that memory a page at a time, timing each reference. If any reference takes more than a few microseconds, the page is no longer in memory. The test program starts small and works forward until either enough memory is seen as present or the memory limit is reached.

3.2. Compile time issues

The GNU C compiler, `gcc`, is the compiler we chose because it gave the most reproducible results across platforms. When `gcc` was not present, we used the vendor-supplied `cc`. All of the benchmarks were compiled with optimization `-O` except the benchmarks that calculate clock speed and the context switch times, which must be compiled without optimization in order to produce correct results. No other optimization flags were enabled because we wanted results that would be commonly seen by application writers.

All of the benchmarks were linked using the default manner of the target system. For most if not all systems, the binaries were linked using shared libraries.

3.3. Multiprocessor issues

All of the multiprocessor systems ran the benchmarks in the same way as the uniprocessor systems. Some systems allow users to pin processes to a particular CPU, which sometimes results in better cache reuse. We do not pin processes because it defeats the

MP scheduler. In certain cases, this decision yields interesting results discussed later.

3.4. Timing issues

- **Clock resolution:** The benchmarks measure the elapsed time by reading the system clock via the `gettimeofday` interface. On some systems this interface has a resolution of 10 milliseconds, a long time relative to many of the benchmarks which have results measured in tens of hundreds of microseconds. To compensate for the coarse clock resolution, the benchmarks are hand-tuned to measure many operations within a single time interval lasting for many clock ticks. Typically, this is done by executing the operation in a small loop, sometimes unrolled if the operation is exceedingly fast, and then dividing the loop time by the loop count.

- **Caching:** If the benchmark expects the data to be in the cache, the benchmark is typically run several times; only the last result is recorded.

If the benchmark does not want to measure cache performance it sets the size parameter larger than the cache. For example, the `bcopy` benchmark by default copies 8 megabytes to 8 megabytes, which largely defeats any second-level cache in use today. (Note that the benchmarks are not trying to defeat the file or process page cache, only the hardware caches.)

- **Variability:** The results of some benchmarks, most notably the context switch benchmark, had a tendency to vary quite a bit, up to 30%. We suspect that the operating system is not using the same set of physical pages each time a process is created and we are seeing the effects of collisions in the external caches. We compensate by running the benchmark in a loop and taking the minimum result. Users interested in the most accurate data are advised to verify the results on their own platforms.

Many of the results included in the database were donated by users and were not created by the authors. Good benchmarking practice suggests that one should

run the benchmarks as the only user of a machine, without other resource intensive or unpredictable processes or daemons.

3.5. Using the `lmbench` database

`lmbench` includes a database of results that is useful for comparison purposes. It is quite easy to build the source, run the benchmark, and produce a table of results that includes the run. All of the tables in this paper were produced from the database included in `lmbench`. This paper is also included with `lmbench` and may be reproduced incorporating new results. For more information, consult the file `lmbench-HOWTO` in the `lmbench` distribution.

4. Systems tested

`lmbench` has been run on a wide variety of platforms. This paper includes results from a representative subset of machines and operating systems. Comparisons between similar hardware running different operating systems can be very illuminating, and we have included a few examples in our results.

The systems are briefly characterized in Table 1. Please note that the list prices are very approximate as is the year of introduction. The SPECInt92 numbers are a little suspect since some vendors have been “optimizing” for certain parts of SPEC. We try and quote the original SPECInt92 numbers where we can.

4.1. Reading the result tables

Throughout the rest of this paper, we present tables of results for many of the benchmarks. All of the tables are sorted, from best to worst. Some tables have multiple columns of results and those tables are sorted on only one of the columns. The sorted column’s heading will be in **bold**.

5. Bandwidth benchmarks

By bandwidth, we mean the rate at which a particular facility can move data. We attempt to measure the data movement ability of a number of different

| Name used | Vender & model | Multi or Uni | Operating System | CPU | Mhz | Year | SPEC Int92 | List price |
|---------------|----------------|--------------|--------------------|--------------|-----|------|------------|------------|
| IBM PowerPC | IBM 43P | Uni | AIX 3.? | MPC604 | 133 | '95 | 176 | 15k |
| IBM Power2 | IBM 990 | Uni | AIX 4.? | Power2 | 71 | '93 | 126 | 110k |
| FreeBSD/i586 | ASUS P55TP4XE | Uni | FreeBSD 2.1 | Pentium | 133 | '95 | 190 | 3k |
| HP K210 | HP 9000/859 | MP | HP-UX B.10.01 | PA 7200 | 120 | '95 | 167 | 35k |
| SGI Challenge | SGI Challenge | MP | IRIX 6.2- α | R4400 | 200 | '94 | 140 | 80k |
| SGI Indigo2 | SGI Indigo2 | Uni | IRIX 5.3 | R4400 | 200 | '94 | 135 | 15k |
| Linux/Alpha | DEC Cabriolet | Uni | Linux 1.3.38 | Alpha 21064A | 275 | '94 | 189 | 9k |
| Linux/i586 | Triton/EDO RAM | Uni | Linux 1.3.28 | Pentium | 120 | '95 | 155 | 5k |
| Linux/i686 | Intel Alder | Uni | Linux 1.3.37 | Pentium Pro | 167 | '95 | ~ 280 | 7k |
| DEC Alpha@150 | DEC 3000/500 | Uni | OSF1 3.0 | Alpha 21064 | 150 | '93 | 84 | 35k |
| DEC Alpha@300 | DEC 8400 5/300 | MP | OSF1 3.2 | Alpha 21164 | 300 | '95 | 341 | ? 250k |
| Sun Ultra1 | Sun Ultra1 | Uni | SunOS 5.5 | UltraSPARC | 167 | '95 | 250 | 21k |
| Sun SC1000 | Sun SC1000 | MP | SunOS 5.5- β | SuperSPARC | 50 | '92 | 65 | 35k |
| Solaris/i686 | Intel Alder | Uni | SunOS 5.5.1 | Pentium Pro | 133 | '95 | ~ 215 | 5k |
| Unixware/i686 | Intel Aurora | Uni | Unixware 5.4.2 | Pentium Pro | 167 | '95 | ~ 280 | 7k |

Table 1. System descriptions.

facilities: library `bcopy`, hand-unrolled `bcopy`, direct-memory read and write (no copying), pipes, TCP sockets, the `read` interface, and the `mmap` interface.

5.1. Memory bandwidth

Data movement is fundamental to any operating system. In the past, performance was frequently measured in MFLOPS because floating point units were slow enough that microprocessor systems were rarely limited by memory bandwidth. Today, floating point units are usually much faster than memory bandwidth, so many current MFLOP ratings can not be maintained using memory-resident data; they are “cache only” ratings.

We measure the ability to copy, read, and write data over a varying set of sizes. There are too many results to report all of them here, so we concentrate on large memory transfers.

We measure copy bandwidth two ways. The first is the user-level library `bcopy` interface. The second is a hand-unrolled loop that loads and stores aligned 8-byte words. In both cases, we took care to ensure that the source and destination locations would not map to the same lines if the any of the caches were direct-mapped. In order to test memory bandwidth rather than cache bandwidth, both benchmarks copy an 8M¹ area to another 8M area. (As secondary caches reach 16M, these benchmarks will have to be resized to reduce caching effects.)

The copy results actually represent one-half to one-third of the memory bandwidth used to obtain those results since we are reading and writing memory. If the cache line size is larger than the word stored, then the written cache line will typically be read before it is written. The actual amount of memory bandwidth used varies because some architectures have special instructions specifically designed for the `bcopy` function. Those architectures will move twice as much memory as reported by this benchmark; less advanced architectures move three times as much memory: the memory read, the memory read because it is about to be overwritten, and the memory written.

The `bcopy` results reported in Table 2 may be correlated with John McCalpin’s `stream` [McCalpin95] benchmark results in the following manner: the `stream` benchmark reports all of the memory moved whereas the `bcopy` benchmark reports the bytes copied. So our numbers should be approximately one-half to one-third of his numbers.

Memory reading is measured by an unrolled loop that sums up a series of integers. On most (perhaps all) systems measured the integer size is 4 bytes. The loop is unrolled such that most compilers generate code that uses a constant offset with the load, resulting

¹ Some of the PCs had less than 16M of available memory; those machines copied 4M.

in a load and an add for each word of memory. The add is an integer add that completes in one cycle on all of the processors. Given that today’s processor typically cycles at 10 or fewer nanoseconds (ns) and that memory is typically 200-1,000 ns per cache line, the results reported here should be dominated by the memory subsystem, not the processor add unit.

The memory contents are added up because almost all C compilers would optimize out the whole loop when optimization was turned on, and would generate far too many instructions without optimization. The solution is to add up the data and pass the result as an unused argument to the “finish timing” function.

Memory reads represent about one-third to one-half of the `bcopy` work, and we expect that pure reads should run at roughly twice the speed of `bcopy`. Exceptions to this rule should be studied, for exceptions indicate a bug in the benchmarks, a problem in `bcopy`, or some unusual hardware.

| System | Bcopy | | Memory | |
|---------------|----------|------|--------|-------|
| | unrolled | libc | read | write |
| IBM Power2 | 242 | 171 | 205 | 364 |
| Sun Ultra1 | 85 | 167 | 129 | 152 |
| DEC Alpha@300 | 85 | 80 | 120 | 123 |
| HP K210 | 78 | 57 | 117 | 126 |
| Unixware/i686 | 65 | 55 | 214 | 86 |
| Solaris/i686 | 52 | 48 | 159 | 71 |
| DEC Alpha@150 | 46 | 45 | 79 | 91 |
| Linux/i686 | 42 | 57 | 205 | 56 |
| FreeBSD/i586 | 39 | 42 | 73 | 83 |
| Linux/Alpha | 39 | 39 | 73 | 71 |
| Linux/i586 | 38 | 42 | 74 | 75 |
| SGI Challenge | 35 | 36 | 65 | 67 |
| SGI Indigo2 | 31 | 32 | 69 | 66 |
| IBM PowerPC | 21 | 21 | 63 | 26 |
| Sun SC1000 | 17 | 15 | 38 | 31 |

Table 2. Memory bandwidth (MB/s)

Memory writing is measured by an unrolled loop that stores a value into an integer (typically a 4 byte integer) and then increments the pointer. The processor cost of each memory operation is approximately the same as the cost in the read case.

The numbers reported in Table 2 are not the raw hardware speed in some cases. The Power2² is capable of up to 800M/sec read rates [McCalpin95] and HP PA RISC (and other prefetching) systems also do better if higher levels of code optimization used and/or the code is hand tuned.

The Sun libc `bcopy` in Table 2 is better because they use a hardware specific `bcopy` routine that uses instructions new in SPARC V9 that were added specifically for memory movement.

The Pentium Pro read rate in Table 2 is much higher than the write rate because, according to Intel,

² Someone described this machine as a \$1,000 processor on a \$99,000 memory subsystem.

the write transaction turns into a read followed by a write to maintain cache consistency for MP systems.

5.2. IPC bandwidth

Interprocess communication bandwidth is frequently a performance issue. Many Unix applications are composed of several processes communicating through pipes or TCP sockets. Examples include the `groff` documentation system that prepared this paper, the X Window System, remote file access, and World Wide Web servers.

Unix pipes are an interprocess communication mechanism implemented as a one-way byte stream. Each end of the stream has an associated file descriptor; one is the write descriptor and the other the read descriptor. TCP sockets are similar to pipes except they are bidirectional and can cross machine boundaries.

Pipe bandwidth is measured by creating two processes, a writer and a reader, which transfer 50M of data in 64K transfers. The transfer size was chosen so that the overhead of system calls and context switching would not dominate the benchmark time. The reader prints the timing results, which guarantees that all data has been moved before the timing is finished.

TCP bandwidth is measured similarly, except the data is transferred in 1M page aligned transfers instead of 64K transfers. If the TCP implementation supports it, the send and receive socket buffers are enlarged to 1M, instead of the default 4-60K. We have found that setting the transfer size equal to the socket buffer size produces the greatest throughput over the most implementations.

| System | Libc bcopy | pipe | TCP |
|---------------|------------|------|-----|
| HP K210 | 57 | 93 | 34 |
| IBM Power2 | 171 | 84 | 10 |
| Linux/i686 | 57 | 73 | 15 |
| Linux/Alpha | 39 | 73 | 9 |
| Unixware/i686 | 55 | 63 | -1 |
| Sun Ultra1 | 167 | 61 | 51 |
| DEC Alpha@300 | 80 | 46 | 11 |
| Solaris/i686 | 48 | 38 | 20 |
| DEC Alpha@150 | 45 | 35 | 9 |
| SGI Indigo2 | 32 | 34 | 22 |
| Linux/i586 | 42 | 34 | 7 |
| IBM PowerPC | 21 | 30 | 17 |
| FreeBSD/i586 | 42 | 23 | 13 |
| SGI Challenge | 36 | 17 | 31 |
| Sun SC1000 | 15 | 9 | 11 |

Table 3. Pipe and local TCP bandwidth (MB/s)

`bcopy` is important to this test because the pipe write/read is typically implemented as a `bcopy` into the kernel from the writer and then a `bcopy` from the kernel to the reader. Ideally, these results would be approximately one-half of the `bcopy` results. It is possible for the kernel `bcopy` to be faster than the C library `bcopy` since the kernel may have access to

`bcopy` hardware unavailable to the C library.

It is interesting to compare pipes with TCP because the TCP benchmark is identical to the pipe benchmark except for the transport mechanism. Ideally, the TCP bandwidth would be as good as the pipe bandwidth. It is not widely known that the majority of the TCP cost is in the `bcopy`, the checksum, and the network interface driver. The checksum and the driver may be safely eliminated in the loopback case and if the costs have been eliminated, then TCP should be just as fast as pipes. From the pipe and TCP results in Table 3, it is easy to see that Solaris and HP-UX have done this optimization.

Bcopy rates in Table 3 can be lower than pipe rates because the pipe transfers are done in 64K buffers, a size that frequently fits in caches, while the `bcopy` is typically an 8M-to-8M copy, which does not fit in the cache.

In Table 3, the SGI Indigo2, a uniprocessor, does better than the SGI MP on pipe bandwidth because of caching effects - in the UP case, both processes share the cache; on the MP, each process is communicating with a different cache.

All of the TCP results in Table 3 are in loopback mode — that is both ends of the socket are on the same machine. It was impossible to get remote networking results for all the machines included in this paper. We are interested in receiving more results for identical machines with a dedicated network connecting them. The results we have for over the wire TCP bandwidth are shown below.

| System | Network | TCP bandwidth |
|--------------------|----------|---------------|
| SGI PowerChallenge | hippi | 79.3 |
| Sun Ultra1 | 100baseT | 9.5 |
| HP 9000/735 | fddi | 8.8 |
| FreeBSD/i586 | 100baseT | 7.9 |
| SGI Indigo2 | 10baseT | .9 |
| HP 9000/735 | 10baseT | .9 |
| Linux/i586@90Mhz | 10baseT | .7 |

Table 4. Remote TCP bandwidth (MB/s)

The SGI using 100MB/s Hippi is by far the fastest in Table 4. The SGI Hippi interface has hardware support for TCP checksums and the IRIX operating system uses virtual memory tricks to avoid copying data as much as possible. For larger transfers, SGI Hippi has reached 92MB/s over TCP.

100baseT is looking quite competitive when compared to FDDI in Table 4, even though FDDI has packets that are almost three times larger. We wonder how long it will be before we see gigabit ethernet interfaces.

5.3. Cached I/O bandwidth

Experience has shown us that reusing data in the file system page cache can be a performance issue. This section measures that operation through two

interfaces, `read` and `mmap`. The benchmark here is not an I/O benchmark in that no disk activity is involved. We wanted to measure the overhead of reusing data, an overhead that is CPU intensive, rather than disk intensive.

The `read` interface copies data from the kernel's file system page cache into the process's buffer, using 64K buffers. The transfer size was chosen to minimize the kernel entry overhead while remaining realistically sized.

The difference between the `bcopy` and the `read` benchmarks is the cost of the file and virtual memory system overhead. In most systems, the `bcopy` speed should be faster than the `read` speed. The exceptions usually have hardware specifically designed for the `bcopy` function and that hardware may be available only to the operating system.

The `read` benchmark is implemented by rereading a file (typically 8M) in 64K buffers. Each buffer is summed as a series of integers in the user process. The summing is done for two reasons: for an apples-to-apples comparison the memory-mapped benchmark needs to touch all the data, and the file system can sometimes transfer data into memory faster than the processor can read the data. For example, SGI's XFS can move data into memory at rates in excess of 500M per second, but it can move data into the cache at only 68M per second. The intent is to measure performance delivered to the application, not DMA performance to memory.

| System | Libc bcopy | File read | Memory read | File mmap |
|---------------|---------------|--------------|----------------|--------------|
| IBM Power2 | 171 | 187 | 205 | 106 |
| HP K210 | 57 | 88 | 117 | 52 |
| Sun Ultra1 | 167 | 85 | 129 | 101 |
| DEC Alpha@300 | 80 | 67 | 120 | 78 |
| Unixware/i686 | 55 | 53 | 214 | 198 |
| Solaris/i686 | 48 | 52 | 159 | 94 |
| Linux/i686 | 57 | 46 | 205 | 34 |
| DEC Alpha@150 | 45 | 40 | 79 | 50 |
| IBM PowerPC | 21 | 40 | 63 | 51 |
| SGI Challenge | 36 | 36 | 65 | 56 |
| SGI Indigo2 | 32 | 32 | 69 | 44 |
| FreeBSD/i586 | 42 | 30 | 73 | 53 |
| Linux/Alpha | 39 | 24 | 73 | 18 |
| Linux/i586 | 42 | 23 | 74 | 9 |
| Sun SC1000 | 15 | 20 | 38 | 28 |

Table 5. File vs. memory bandwidth (MB/s)

The `mmap` interface provides a way to access the kernel's file cache without copying the data. The `mmap` benchmark is implemented by mapping the entire file (typically 8M) into the process's address space. The file is then summed to force the data into the cache.

In Table 5, a good system will have *File read* as fast as (or even faster than) *Libc bcopy* because as the file system overhead goes to zero, the file reread case

is virtually the same as the library `bcopy` case. However, file reread can be faster because the kernel may have access to `bcopy` assist hardware not available to the C library. Ideally, *File mmap* performance should approach *Memory read* performance, but `mmap` is often dramatically worse. Judging by the results, this looks to be a potential area for operating system improvements.

In Table 5 the Power2 does better on file reread than `bcopy` because it takes full advantage of the memory subsystem from inside the kernel. The `mmap` reread is probably slower because of the lower clock rate; the page faults start to show up as a significant cost.

It is surprising that the Sun Ultra1 was able to `bcopy` at the high rates shown in Table 2 but did not show those rates for file reread in Table 5. HP has the opposite problem, they get file reread faster than `bcopy`, perhaps because the kernel `bcopy` has access to hardware support.

The Unixware system has outstanding `mmap` reread rates, better than systems of substantially higher cost. Linux needs to do some work on the `mmap` code.

6. Latency measurements

Latency is an often-overlooked area of performance problems, possibly because resolving latency issues is frequently much harder than resolving bandwidth issues. For example, memory bandwidth may be increased by making wider cache lines and increasing memory "width" and interleave, but memory latency can be improved only by shortening paths or increasing (successful) prefetching. The first step toward improving latency is understanding the current latencies in a system.

The latency measurements included in this suite are memory latency, basic operating system entry cost, signal handling cost, process creation times, context switching, interprocess communication, file system latency, and disk latency.

6.1. Memory read latency background

In this section, we expend considerable effort to define the different memory latencies and to explain and justify our benchmark. The background is a bit tedious but important, since we believe the memory latency measurements to be one of the most thought-provoking and useful measurements in `lmbench`.

The most basic latency measurement is memory latency since most of the other latency measurements can be expressed in terms of memory latency. For example, context switches require saving the current process state and loading the state of the next process. However, memory latency is rarely accurately measured and frequently misunderstood.

Memory read latency has many definitions; the most common, in increasing time order, are memory chip cycle time, processor-pins-to-memory-and-back time, load-in-a-vacuum time, and back-to-back-load time.

- **Memory chip cycle latency:** Memory chips are rated in nanoseconds; typical speeds are around 60ns. A general overview on DRAM architecture may be found in [Hennessy96]. The specific information we describe here is from [Toshiba94] and pertains to the THM361020AS-60 module and TC514400AJS DRAM used in SGI workstations. The 60ns time is the time from RAS assertion to the when the data will be available on the DRAM pins (assuming $\overline{\text{CAS}}$ access time requirements were met). While it is possible to get data out of a DRAM in 60ns, that is not all of the time involved. There is a precharge time that must occur after every access. [Toshiba94] quotes 110ns as the random read or write cycle time and this time is more representative of the cycle time.

- **Pin-to-pin latency:** This number represents the time needed for the memory request to travel from the processor's pins to the memory subsystem and back again. Many vendors have used the pin-to-pin definition of memory latency in their reports. For example, [Fenwick95] while describing the DEC 8400 quotes memory latencies of 265ns; a careful reading of that paper shows that these are pin-to-pin numbers. In spite of the historical precedent in vendor reports, this definition of memory latency is misleading since it ignores actual delays seen when a load instruction is immediately followed by a use of the data being loaded. The number of additional cycles inside the processor can be significant and grows more significant with today's highly pipelined architectures.

It is worth noting that the pin-to-pin numbers include the amount of time it takes to charge the lines going to the SIMMs, a time that increases with the (potential) number of SIMMs in a system. More SIMMs mean more capacitance which requires in longer charge times. This is one reason why personal computers frequently have better memory latencies than workstations: the PCs typically have less memory capacity.

- **Load-in-a-vacuum latency:** A load in a vacuum is the time that the processor will wait for one load that must be fetched from main memory (i.e., a cache miss). The "vacuum" means that there is no other activity on the system bus, including no other loads. While this number is frequently used as the memory latency, it is not very useful. It is basically a "not to exceed" number important only for marketing reasons. Some architects point out that since most processors implement nonblocking loads (the load does not cause a stall until the data is used), the perceived load latency may be much less than the real latency. When pressed, however, most will admit that cache misses occur in bursts, resulting in perceived latencies

of at least the load-in-a-vacuum latency.

- **Back-to-back-load latency:** Back-to-back-load latency is the time that each load takes, assuming that the instructions before and after are also cache-missing loads. Back-to-back loads may take longer than loads in a vacuum for the following reason: many systems implement something known as *critical word first*, which means that the subblock of the cache line that contains the word being loaded is delivered to the processor before the entire cache line has been brought into the cache. If another load occurs quickly enough after the processor gets restarted from the current load, the second load may stall because the cache is still busy filling the cache line for the previous load. On some systems, such as the current implementation of UltraSPARC, the difference between back to back and load in a vacuum is about 35%.

lmbench measures back-to-back-load latency because it is the only measurement that may be easily measured from software and because we feel that it is what most software developers consider to be memory latency. Consider the following C code fragment:

```
p = head;
while (p->p_next)
    p = p->p_next;
```

On a DEC Alpha, the loop part turns into three instructions, including the load. A 300 Mhz processor has a 3.33ns cycle time, so the loop could execute in slightly less than 10ns. However, the load itself takes 400ns on a 300 Mhz DEC 8400. In other words, the instructions cost 10ns but the load stalls for 400. Another way to look at it is that 400/3.3, or 121, nondependent, nonloading instructions following the load would be needed to hide the load latency. Because superscalar processors typically execute multiple operations per clock cycle, they need even more useful operations between cache misses to keep the processor from stalling.

This benchmark illuminates the tradeoffs in processor cache design. Architects like large cache lines, up to 64 bytes or so, because the prefetch effect of gathering a whole line increases hit rate given reasonable spatial locality. Small stride sizes have high spatial locality and should have higher performance, but large stride sizes have poor spatial locality causing the system to prefetch useless data. So the benchmark provides the following insight into negative effects of large line prefetch:

- Multi-cycle fill operations are typically atomic events at the caches, and sometimes block other cache accesses until they complete.
- Caches are typically single-ported. Having a large line prefetch of unused data causes extra bandwidth demands at the cache, and can cause increased access latency for normal cache accesses.

| System | Level 1 cache | | Level 2 cache | | Memory latency |
|---------------|---------------|-----------|---------------|--------|----------------|
| | Clk. | lat. size | lat. size | size | |
| HP K210 | 8 | 8 256K | -- | -- | 349 |
| IBM Power2 | 14 | 13 256K | -- | -- | 260 |
| Unixware/i686 | 5 | 5 8K | 25 | 256K | 175 |
| Linux/i686 | 5 | 10 8K | 30 | 256K | 179 |
| Sun Ultra1 | 6 | 6 16K | 42 | 512K | 270 |
| Linux/Alpha | 3 | 6 8K | 46 | 96K | 357 |
| Solaris/i686 | 7 | 14 8K | ? 48 | 256K | 281 |
| SGI Indigo2 | 5 | 8 16K | 64 | 2M | 1170 |
| SGI Challenge | 5 | 8 16K | 64 | 4M | 1189 |
| DEC Alpha@300 | 3 | 3 8K | 66 | 4M | 400 |
| DEC Alpha@150 | 6 | 12 8K | 67 | 512K | 291 |
| FreeBSD/i586 | 7 | 7 8K | 95 | 512K | 182 |
| Linux/i586 | 8 | 8 8K | 107 | 256K | 150 |
| Sun SC1000 | 20 | 20 8K | 140 | 1M | 1236 |
| IBM PowerPC | 7 | 6 16K | 164 | ? 512K | 394 |

Table 6. Cache and memory latency (ns)

set associative cache, accessible in one clock (16ns). That system is primarily a database server.

The IBM focus is on low latency, high bandwidth memory. The IBM memory subsystem is good because all of memory is close to the processor, but has the weakness that it is extremely difficult to evolve the design to a multiprocessor system.

The 586 and PowerPC motherboards have quite poor second level caches, the caches are not substantially better than main memory.

The Pentium Pro and Sun Ultra second level caches are of medium speed at 5-6 clocks latency each. 5-6 clocks seems fast until it is compared against the HP and IBM one cycle latency caches of similar size. Given the tight integration of the Pentium Pro level 2 cache, it is surprising that it has such high latencies.

The 300Mhz DEC Alpha has a rather high 22 clock latency to the second level cache which is probably one of the reasons that they needed a 96K level 1.5 cache. SGI and DEC have used large second level caches to hide their long latency from main memory.

6.3. Operating system entry

Entry into the operating system is required for many system facilities. When calculating the cost of a facility, it is useful to know how expensive it is to perform a nontrivial entry into the operating system.

We measure nontrivial entry into the system by repeatedly writing one word to `/dev/null`, a pseudo device driver that does nothing but discard the data. This particular entry point was chosen because it has never been optimized in any system that we have measured. Other entry points, typically `getpid` and `gettimeofday`, are heavily used, heavily optimized, and sometimes implemented as user-level library routines rather than system calls. A write to the `/dev/null` driver will go through the system

call table to write, verify the user area as readable, look up the file descriptor to get the vnode, call the vnode's write function, and then return.

| System | system call |
|---------------|-------------|
| Linux/Alpha | 2 |
| Linux/i586 | 2 |
| Linux/i686 | 4 |
| Sun Ultra1 | 5 |
| Unixware/i686 | 5 |
| FreeBSD/i586 | 6 |
| Solaris/i686 | 7 |
| DEC Alpha@300 | 9 |
| Sun SC1000 | 9 |
| HP K210 | 10 |
| SGI Indigo2 | 11 |
| DEC Alpha@150 | 11 |
| IBM PowerPC | 12 |
| IBM Power2 | 16 |
| SGI Challenge | 24 |

Table 7. Simple system call time (microseconds)

Linux is the clear winner in the system call time. The reasons are twofold: Linux is a uniprocessor operating system, without any MP overhead, and Linux is a small operating system, without all of the "features" accumulated by the commercial offers.

Unixware and Solaris are doing quite well, given that they are both fairly large, commercially oriented operating systems with a large accumulation of "features."

6.4. Signal handling cost

Signals in Unix are a way to tell another process to handle an event. They are to processes as interrupts are to the CPU.

Signal handling is often critical to layered systems. Some applications, such as databases, software development environments, and threading libraries provide an operating system-like layer on top of the operating system, making signal handling a critical path in many of these applications.

`lmbench` measure both signal installation and signal dispatching in two separate loops, within the context of one process. It measures signal handling by installing a signal handler and then repeatedly sending itself the signal.

Table 8 shows the signal handling costs. Note that there are no context switches in this benchmark; the signal goes to the same process that generated the signal. In real applications, the signals usually go to another process, which implies that the true cost of sending that signal is the signal overhead plus the context switch overhead. We wanted to measure signal and context switch overheads separately since context switch times vary widely among operating systems.

SGI does very well on signal processing, especially since their hardware is of an older generation

| System | sigaction | sig handler |
|---------------|-----------|-------------|
| SGI Indigo2 | 5 | 8 |
| SGI Challenge | 4 | 9 |
| HP K210 | 4 | 13 |
| Linux/i686 | 4 | 22 |
| FreeBSD/i586 | 5 | 25 |
| Unixware/i686 | 6 | 25 |
| IBM Power2 | 10 | 27 |
| Solaris/i686 | 9 | 45 |
| IBM PowerPC | 10 | 52 |
| Linux/i586 | 7 | 52 |
| DEC Alpha@300 | 6 | 59 |
| Linux/Alpha | 13 | 138 |

Table 8. Signal times (microseconds)

than many of the others.

The Linux/Alpha signal handling numbers are so poor that we suspect that this is a bug, especially given that the Linux/x86 numbers are quite reasonable.

6.5. Process creation costs

Process benchmarks are used to measure the basic process primitives, such as creating a new process, running a different program, and context switching. Process creation benchmarks are of particular interest in distributed systems since many remote operations include the creation of a remote process to shepherd the remote operation to completion. Context switching is important for the same reasons.

- **Simple process creation.** The Unix process creation primitive is `fork`, which creates a (virtually) exact copy of the calling process. Unlike VMS and some other operating systems, Unix starts any new process with a `fork`. Consequently, `fork` and/or `execve` should be fast and “light,” facts that many have been ignoring for some time.

`lmbench` measures simple process creation by creating a process and immediately exiting the child process. The parent process waits for the child process to exit. The benchmark is intended to measure the overhead for creating a new thread of control, so it includes the `fork` and the `exit` time.

The benchmark also includes a `wait` system call in the parent and context switches from the parent to the child and back again. Given that context switches of this sort are on the order of 20 microseconds and a system call is on the order of 5 microseconds, and that the entire benchmark time is on the order of a millisecond or more, the extra overhead is insignificant. Note that even this relatively simple task is very expensive and is measured in milliseconds while most of the other operations we consider are measured in microseconds.

- **New process creation.** The preceding benchmark did not create a new application; it created a copy of the old application. This benchmark measures the cost of creating a new process and changing that

process into a new application, which forms the basis of every Unix command line interface, or shell. `lmbench` measures this facility by forking a new child and having that child execute a new program — in this case, a tiny program that prints “hello world” and exits.

The startup cost is especially noticeable on (some) systems that have shared libraries. Shared libraries can introduce a substantial (tens of milliseconds) startup cost.

| System | fork & exit | fork, exec & exit | fork, exec sh -c & exit |
|---------------|-------------|-------------------|-------------------------|
| Linux/Alpha | 0.7 | 3 | 12 |
| Linux/i686 | 0.5 | 5 | 17 |
| Linux/i586 | 0.9 | 5 | 16 |
| DEC Alpha@300 | 2.0 | 6 | 16 |
| Unixware/i686 | 1.0 | 6 | 10 |
| IBM PowerPC | 2.9 | 8 | 50 |
| SGI Indigo2 | 3.1 | 8 | 19 |
| IBM Power2 | 1.2 | 8 | 16 |
| FreeBSD/i586 | 2.0 | 11 | 19 |
| HP K210 | 3.1 | 11 | 20 |
| DEC Alpha@150 | 4.6 | 13 | 39 |
| SGI Challenge | 4.0 | 14 | 24 |
| Sun Ultra1 | 3.7 | 20 | 37 |
| Solaris/i686 | 4.5 | 22 | 46 |
| Sun SC1000 | 14.0 | 69 | 281 |

Table 9. Process creation time (milliseconds)

- **Complicated new process creation.** When programs start other programs, they frequently use one of three standard interfaces: `popen`, `system`, and/or `execvp`. The first two interfaces start a new process by invoking the standard command interpreter, `/bin/sh`, to start the process. Starting programs this way guarantees that the shell will look for the requested application in all of the places that the user would look — in other words, the shell uses the user’s `$PATH` variable as a list of places to find the application. `execvp` is a C library routine which also looks for the program using the user’s `$PATH` variable.

Since this is a common way of starting applications, we felt it was useful to show the costs of the generality.

We measure this by starting `/bin/sh` to start the same tiny program we ran in the last case. In Table 9 the cost of asking the shell to go look for the program is quite large, frequently ten times as expensive as just creating a new process, and four times as expensive as explicitly naming the location of the new program.

The results that stand out in Table 9 are the poor Sun Ultra 1 results. Given that the processor is one of the fastest, the problem is likely to be software. There is room for substantial improvement in the Solaris process creation code.

6.6. Context switching

Context switch time is defined here as the time needed to save the state of one process and restore the state of another process.

Context switches are frequently in the critical performance path of distributed applications. For example, the multiprocessor versions of the IRIX operating system use processes to move data through the networking stack. This means that the processing time for each new packet arriving at an idle system includes the time needed to switch in the networking process.

Typical context switch benchmarks measure just the minimal context switch time — the time to switch between two processes that are doing nothing but context switching. We feel that this is misleading because there are frequently more than two active processes, and they usually have a larger working set (cache footprint) than the benchmark processes.

Other benchmarks frequently include the cost of the system calls needed to force the context switches. For example, Ousterhout's context switch benchmark measures context switch time plus a read and a write on a pipe. In many of the systems measured by `lmbench`, the pipe overhead varies between 30% and 300% of the context switch time, so we were careful to factor out the pipe overhead.

- **Number of processes.** The context switch benchmark is implemented as a ring of two to twenty processes that are connected with Unix pipes. A token is passed from process to process, forcing context switches. The benchmark measures the time needed to pass the token two thousand times from process to process. Each transfer of the token has two costs: the context switch, and the overhead of passing the token. In order to calculate just the context switching time, the benchmark first measures the cost of passing the token through a ring of pipes in a single process. This overhead time is defined as the cost of passing the token and is not included in the reported context switch time.

- **Size of processes.** In order to measure more realistic context switch times, we add an artificial variable size “cache footprint” to the switching processes. The cost of the context switch then includes the cost of restoring user-level state (cache footprint). The cache footprint is implemented by having the process allocate an array of data⁴ and sum the array as a series of integers after receiving the token but before passing the token to the next process. Since most systems will cache data across context switches, the working set for the benchmark is slightly larger than the number of processes times the array size.

It is worthwhile to point out that the overhead mentioned above also includes the cost of accessing the data, in the same way as the actual benchmark.

⁴ All arrays are at the same virtual address in all processes.

However, because the overhead is measured in a single process, the cost is typically the cost with “hot” caches. In the Figure 2, each size is plotted as a line, with context switch times on the Y axis, number of processes on the X axis, and the process size as the data set. The process size and the hot cache overhead costs for the pipe read/writes and any data access is what is labeled as `size=0KB overhead=10`. The size is in kilobytes and the overhead is in microseconds.

The context switch time does not include anything other than the context switch, provided that all the benchmark processes fit in the cache. If the total size of all of the benchmark processes is larger than the cache size, the cost of each context switch will include cache misses. We are trying to show realistic context switch times as a function of both size and number of processes.

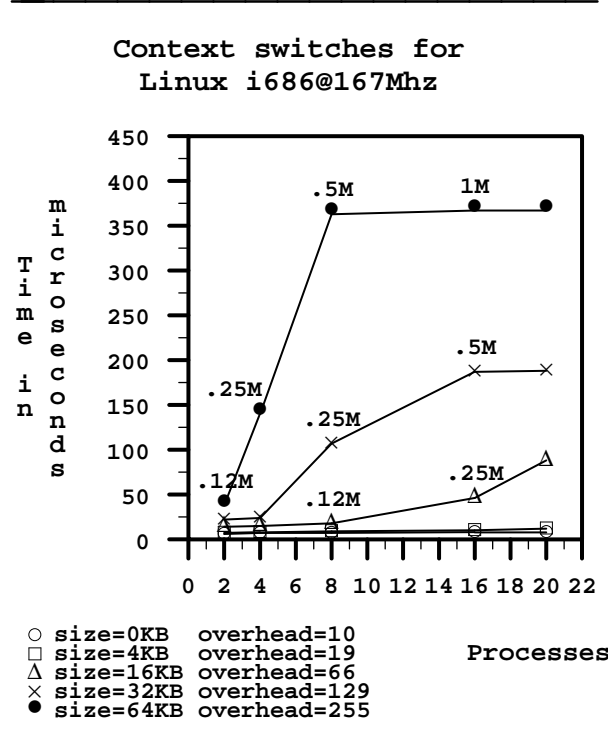


Figure 2. Context switch times

Results for an Intel Pentium Pro system running Linux at 167 MHz are shown in Figure 2. The data points on the figure are labeled with the working set due to the sum of data in all of the processes. The actual working set is larger, as it includes the process and kernel overhead as well. One would expect the context switch times to stay constant until the working set is approximately the size of the second level cache. The Intel system has a 256K second level cache, and the context switch times stay almost constant until about 256K (marked as .25M in the graph).

- **Cache issues** The context switch benchmark is a deliberate measurement of the effectiveness of the caches across process context switches. If the cache does not include the process identifier (PID, also sometimes called an address space identifier) as part of the address, then the cache must be flushed on every context switch. If the cache does not map the same virtual addresses from different processes to different cache lines, then the cache will appear to be flushed on every context switch.

If the caches do not cache across context switches there would be no grouping at the lower left corner of Figure 2, instead, the graph would appear as a series of straight, horizontal, parallel lines. The number of processes will not matter, the two process case will be just as bad as the twenty process case since the cache would not be useful across context switches.

| System | 2 processes | | 8 processes | |
|---------------|-------------|------|-------------|------|
| | 0KB | 32KB | 0KB | 32KB |
| Linux/i686 | 6 | 22 | 7 | 107 |
| Linux/i586 | 10 | 163 | 13 | 215 |
| Linux/Alpha | 11 | 70 | 13 | 78 |
| IBM Power2 | 13 | 16 | 18 | 43 |
| Sun Ultra1 | 14 | 31 | 20 | 102 |
| DEC Alpha@300 | 14 | 17 | 22 | 41 |
| IBM PowerPC | 16 | 87 | 26 | 144 |
| HP K210 | 17 | 17 | 18 | 99 |
| Unixware/i686 | 21 | 27 | 22 | 73 |
| FreeBSD/i586 | 27 | 34 | 33 | 102 |
| Solaris/i686 | 36 | 54 | 43 | 118 |
| SGI Indigo2 | 40 | 47 | 38 | 104 |
| DEC Alpha@150 | 53 | 68 | 59 | 134 |
| SGI Challenge | 63 | 80 | 69 | 93 |
| Sun SC1000 | 107 | 142 | 104 | 197 |

Table 10. Context switch time (microseconds)

We picked four points on the graph and extracted those values for Table 10. The complete set of values, as well as tools to graph them, are included with `lmbench`.

Note that multiprocessor context switch times are frequently more expensive than uniprocessor context switch times. This is because multiprocessor operating systems tend to have very complicated scheduling code. We believe that multiprocessor context switch times can be, and should be, within 10% of the uniprocessor times.

Linux does quite well on context switching, especially on the more recent architectures. By comparing the Linux 2 0K processes to the Linux 2 32K processes, it is apparent that there is something wrong with the Linux/i586 case. If we look back to Table 6, we can find at least part of the cause. The second level cache latency for the i586 is substantially worse than either the i686 or the Alpha.

Given the poor second level cache behavior of the PowerPC, it is surprising that it does so well on context switches, especially the larger sized cases.

The Sun Ultra1 context switches quite well in part because of enhancements to the register window handling in SPARC V9.

6.7. Interprocess communication latencies

Interprocess communication latency is important because many operations are control messages to another process (frequently on another system). The time to tell the remote process to do something is pure overhead and is frequently in the critical path of important functions such as distributed applications (e.g., databases, network servers).

The interprocess communication latency benchmarks typically have the following form: pass a small message (a byte or so) back and forth between two processes. The reported results are always the microseconds needed to do one round trip. For one way timing, about half the round trip is right. However, the CPU cycles tend to be somewhat asymmetric for one trip: receiving is typically more expensive than sending.

- **Pipe latency.** Unix pipes are an interprocess communication mechanism implemented as a one-way byte stream. Each end of the stream has an associated file descriptor; one is the write descriptor and the other the read descriptor.

Pipes are frequently used as a local IPC mechanism. Because of the simplicity of pipes, they are frequently the fastest portable communication mechanism.

Pipe latency is measured by creating a pair of pipes, forking a child process, and passing a word back and forth. This benchmark is identical to the two-process, zero-sized context switch benchmark, except that it includes both the context switching time and the pipe overhead in the results. Table 11 shows the round trip latency from process A to process B and back to process A.

| System | Pipe latency |
|---------------|--------------|
| Linux/i686 | 31 |
| Linux/i586 | 33 |
| Linux/Alpha | 34 |
| Sun Ultra1 | 62 |
| IBM PowerPC | 65 |
| DEC Alpha@300 | 71 |
| HP K210 | 78 |
| Unixware/i686 | 86 |
| IBM Power2 | 91 |
| Solaris/i686 | 101 |
| FreeBSD/i586 | 104 |
| SGI Indigo2 | 131 |
| DEC Alpha@150 | 179 |
| SGI Challenge | 251 |
| Sun SC1000 | 278 |

Table 11. Pipe latency (microseconds)

The time can be broken down to two context switches plus four system calls plus the pipe overhead.

The context switch component is two of the small processes in Table 10. This benchmark is identical to the context switch benchmark in [Ousterhout90].

- **TCP and RPC/TCP latency.** TCP sockets may be viewed as an interprocess communication mechanism similar to pipes with the added feature that TCP sockets work across machine boundaries.

TCP and RPC/TCP connections are frequently used in low-bandwidth, latency-sensitive applications. The default Oracle distributed lock manager uses TCP sockets, and the locks per second available from this service are accurately modeled by the TCP latency test.

| System | TCP | RPC/TCP |
|---------------|-----|---------|
| Sun Ultra1 | 162 | 346 |
| DEC Alpha@300 | 267 | 371 |
| Linux/i686 | 263 | 427 |
| FreeBSD/i586 | 256 | 440 |
| Solaris/i686 | 305 | 528 |
| Linux/Alpha | 429 | 602 |
| HP K210 | 146 | 606 |
| SGI Indigo2 | 278 | 641 |
| IBM Power2 | 332 | 649 |
| IBM PowerPC | 299 | 698 |
| Linux/i586 | 467 | 713 |
| DEC Alpha@150 | 485 | 788 |
| SGI Challenge | 546 | 900 |
| Sun SC1000 | 855 | 1386 |

Table 12. TCP latency (microseconds)

Sun's RPC is layered either over TCP or over UDP. The RPC layer is responsible for managing connections (the port mapper), managing different byte orders and word sizes (XDR), and implementing a remote procedure call abstraction. Table 12 shows the same benchmark with and without the RPC layer to show the cost of the RPC implementation.

TCP latency is measured by having a server process that waits for connections and a client process that connects to the server. The two processes then exchange a word between them in a loop. The latency reported is one round-trip time. The measurements in Table 12 are local or loopback measurements, since our intent is to show the overhead of the software. The same benchmark may be, and frequently is, used to measure host-to-host latency.

Note that the RPC layer frequently adds hundreds of microseconds of additional latency. The problem is not the external data representation (XDR) layer — the data being passed back and forth is a byte, so there is no XDR to be done. There is no justification for the extra cost; it is simply an expensive implementation. DCE RPC is worse.

- **UDP and RPC/UDP latency.** UDP sockets are an alternative to TCP sockets. They differ in that UDP sockets are unreliable messages that leave the retransmission issues to the application. UDP sockets have a

| System | UDP | RPC/UDP |
|---------------|-----|---------|
| Linux/i686 | 112 | 217 |
| Sun Ultra1 | 197 | 267 |
| Linux/Alpha | 180 | 317 |
| DEC Alpha@300 | 259 | 358 |
| Linux/i586 | 187 | 366 |
| FreeBSD/i586 | 212 | 375 |
| Solaris/i686 | 348 | 454 |
| IBM Power2 | 254 | 531 |
| IBM PowerPC | 206 | 536 |
| HP K210 | 152 | 543 |
| SGI Indigo2 | 313 | 671 |
| DEC Alpha@150 | 489 | 834 |
| SGI Challenge | 678 | 893 |
| Sun SC1000 | 739 | 1101 |

Table 13. UDP latency (microseconds)

few advantages, however. They preserve message boundaries, whereas TCP does not; and a single UDP socket may send messages to any number of other sockets, whereas TCP sends data to only one place.

UDP and RPC/UDP messages are commonly used in many client/server applications. NFS is probably the most widely used RPC/UDP application in the world.

Like TCP latency, UDP latency is measured by having a server process that waits for connections and a client process that connects to the server. The two processes then exchange a word between them in a loop. The latency reported is round-trip time. The measurements in Table 13 are local or loopback measurements, since our intent is to show the overhead of the software. Again, note that the RPC library can add hundreds of microseconds of extra latency.

| System | Network | TCP latency | UDP latency |
|--------------------|----------|-------------|-------------|
| Sun Ultra1 | 100baseT | 280 | 308 |
| FreeBSD/i586 | 100baseT | 365 | 304 |
| HP 9000/735 | fdi | 425 | 441 |
| SGI Indigo2 | 10baseT | 543 | 602 |
| HP 9000/735 | 10baseT | 592 | 603 |
| SGI PowerChallenge | hippi | 1068 | 1099 |
| Linux/i586@90Mhz | 10baseT | 2954 | 1912 |

Table 14. Remote latencies (microseconds)

- **Network latency.** We have a few results for over the wire latency included in Table 14. As might be expected, the most heavily used network interfaces (i.e., ethernet) have the lowest latencies. The times shown include the time on the wire, which is about 130 microseconds for 10Mbit ethernet, 13 microseconds for 100Mbit ethernet and FDDI, and less than 10 microseconds for Hippi.

- **TCP connection latency.** TCP is a connection-based, reliable, byte-stream-oriented protocol. As part of this reliability, a connection must be established before any data can be transferred. The connection is

accomplished by a “three-way handshake,” an exchange of packets when the client attempts to connect to the server.

Unlike UDP, where no connection is established, TCP sends packets at startup time. If an application creates a TCP connection to send one message, then the startup time can be a substantial fraction of the total connection and transfer costs. The benchmark shows that the connection cost is approximately half of the cost.

Connection cost is measured by having a server, registered using the port mapper, waiting for connections. The client figures out where the server is registered and then repeatedly times a connect system call to the server. The socket is closed after each connect. Twenty connects are completed and the fastest of them is used as the result. The time measured will include two of the three packets that make up the three way TCP handshake, so the cost is actually greater than the times listed.

| System | TCP connection |
|---------------|----------------|
| HP K210 | 238 |
| Linux/i686 | 316 |
| IBM Power2 | 339 |
| FreeBSD/i586 | 418 |
| Linux/i586 | 606 |
| SGI Indigo2 | 667 |
| SGI Challenge | 716 |
| Sun Ultra1 | 852 |
| Solaris/i686 | 1230 |
| Sun SC1000 | 3047 |

Table 15. TCP connect latency (microseconds)

Table 15 shows that if the need is to send a quick message to another process, given that most packets get through, a UDP message will cost a send and a reply (if positive acknowledgments are needed, which they are in order to have an apples-to-apples comparison with TCP). If the transmission medium is 10Mbit Ethernet, the time on the wire will be approximately 65 microseconds each way, or 130 microseconds total. To do the same thing with a short-lived TCP connection would cost 896 microseconds of wire time alone.

The comparison is not meant to disparage TCP; TCP is a useful protocol. Nor is the point to suggest that all messages should be UDP. In many cases, the difference between 130 microseconds and 900 microseconds is insignificant compared with other aspects of application performance. However, if the application is very latency sensitive and the transmission medium is slow (such as serial link or a message through many routers), then a UDP message may prove cheaper.

6.8. File system latency

File system latency is defined as the time required to create or delete a zero length file. We define it this way because in many file systems, such as the BSD fast file system, the directory operations are done synchronously in order to maintain on-disk integrity. Since the file data is typically cached and sent to disk at some later date, the file creation and deletion become the bottleneck seen by an application. This bottleneck is substantial: to do a synchronous update to a disk is a matter of tens of milliseconds. In many cases, this bottleneck is much more of a perceived performance issue than processor speed.

The benchmark creates 1,000 zero-sized files and then deletes them. All the files are created in one directory and their names are short, such as "a", "b", "c", ... "aa", "ab",

| System | FS | Create | Delete |
|---------------|--------|--------|--------|
| Linux/i686 | EXT2FS | 751 | 45 |
| HP K210 | HFS | 579 | 67 |
| Linux/i586 | EXT2FS | 1,114 | 95 |
| Linux/Alpha | EXT2FS | 834 | 115 |
| Unixware/i686 | UFS | 450 | 369 |
| SGI Challenge | XFS | 3,508 | 4,016 |
| DEC Alpha@300 | ADVFS | 4,255 | 4,184 |
| Solaris/i686 | UFS | 23,809 | 7,246 |
| Sun Ultra1 | UFS | 18,181 | 8,333 |
| Sun SC1000 | UFS | 25,000 | 11,111 |
| FreeBSD/i586 | UFS | 28,571 | 11,235 |
| SGI Indigo2 | EFS | 11,904 | 11,494 |
| DEC Alpha@150 | ? | 38,461 | 12,345 |
| IBM PowerPC | JFS | 12,658 | 12,658 |
| IBM Power2 | JFS | 13,333 | 12,820 |

Table 16. File system latency (microseconds)

The create and delete latencies are shown in Table 16. Notice that Linux does extremely well here, 2 to 3 orders of magnitude faster than the slowest systems. However, Linux does not guarantee anything about the disk integrity; the directory operations are done in memory. Other fast systems, such as SGI's XFS, use a log to guarantee the file system integrity. The slower systems, all those with ~10 millisecond file latencies, are using synchronous writes to guarantee the file system integrity. Unless Unixware has modified UFS substantially, they must be running in an unsafe mode since the FreeBSD UFS is much slower and both file systems are basically the 4BSD fast file system.

6.9. Disk latency

Included with lmbench is a small benchmarking program useful for measuring disk and file I/O. lmdc, which is patterned after the Unix utility dd, measures both sequential and random I/O, optionally generates patterns on output and checks them on input, supports flushing the data from the buffer cache on systems that support msync, and has a very flexible user interface. Many I/O benchmarks can be

trivially replaced with a perl script wrapped around `lmd`.

While we could have generated both sequential and random I/O results as part of this paper, we did not because those benchmarks are heavily influenced by the performance of the disk drives used in the test. We intentionally measure only the system overhead of a SCSI command since that overhead may become a bottleneck in large database configurations.

Some important applications, such as transaction processing, are limited by random disk IO latency. Administrators can increase the number of disk operations per second by buying more disks, until the processor overhead becomes the bottleneck. The `lmd` benchmark measures the processor overhead associated with each disk operation, and it can provide an upper bound on the number of disk operations the processor can support. It is designed for SCSI disks, and it assumes that most disks have 32-128K read-ahead buffers and that they can read ahead faster than the processor can request the chunks of data.⁵

The benchmark simulates a large number of disks by reading 512byte transfers sequentially from the raw disk device (raw disks are unbuffered and are not read ahead by Unix). Since the disk can read ahead faster than the system can request data, the benchmark is doing small transfers of data from the disk's track buffer. Another way to look at this is that the benchmark is doing memory-to-memory transfers across a SCSI channel. It is possible to generate loads of more than 1,000 SCSI operations/second on a single SCSI disk. For comparison, disks under database load typically run at 20-80 operations per second.

| System | Disk latency |
|---------------|--------------|
| SGI Challenge | 920 |
| SGI Indigo2 | 984 |
| HP K210 | 1103 |
| DEC Alpha@150 | 1436 |
| Sun SC1000 | 1466 |
| Sun Ultra1 | 2242 |

Table 17. SCSI I/O overhead (microseconds)

The resulting overhead number represents a **lower** bound on the overhead of a disk I/O. The real overhead numbers will be higher on SCSI systems because most SCSI controllers will not disconnect if the request can be satisfied immediately. During the benchmark, the processor simply sends the request and transfers the data, while during normal operation, the processor will send the request, disconnect, get

⁵ This may not always be true: a processor could be fast enough to make the requests faster than the rotating disk. If we take 6M/second to be disk speed, and divide that by 512 (the minimum transfer size), that is 12,288 IOs/second, or 81 microseconds/IO. We don't know of any processor/OS/IO controller combinations that can do an IO in 81 microseconds.

interrupted, reconnect, and transfer the data.

This technique can be used to discover how many drives a system can support before the system becomes CPU-limited because it can produce the overhead load of a fully configured system with just a few disks.

7. Future work

There are several known improvements and extensions that could be made to `lmbench`.

- **Memory latency.** The current benchmark measures clean-read latency. By clean, we mean that the cache lines being replaced are highly likely to be unmodified, so there is no associated write-back cost. We would like to extend the benchmark to measure dirty-read latency, as well as write latency. Other changes include making the benchmark impervious to sequential prefetching and measuring TLB miss cost.

- **MP benchmarks.** None of the benchmarks in `lmbench` is designed to measure any multiprocessor features directly. At a minimum, we could measure cache-to-cache latency as well as cache-to-cache bandwidth.

- **Static vs. dynamic processes.** In the process creation section, we allude to the cost of starting up processes that use shared libraries. When we figure out how to create statically linked processes on all or most systems, we could quantify these costs exactly.

- **McCalpin's stream benchmark.** We will probably incorporate part or all of this benchmark into `lmbench`.

- **Automatic sizing.** We have enough technology that we could determine the size of the external cache and autosize the memory used such that the external cache had no effect.

- **More detailed papers.** There are several areas that could yield some interesting papers. The memory latency section could use an in-depth treatment, and the context switching section could turn into an interesting discussion of caching technology.

8. Conclusion

`lmbench` is a useful, portable micro-benchmark suite designed to measure important aspects of system performance. We have found that a good memory subsystem is at least as important as the processor speed. As processors get faster and faster, more and more of the system design effort will need to move to the cache and memory subsystems.

9. Acknowledgments

Many people have provided invaluable help and insight into both the benchmarks themselves and the paper. The USENIX reviewers were especially helpful. We thank all of them and especially thank: Ken Okin (SUN), Kevin Normoyle (SUN), Satya Nishtala (SUN),

Greg Chesson (SGI), John Mashey (SGI), Neal Nuckolls (SGI), John McCalpin (Univ. of Delaware), Ron Minnich (Sarnoff), Chris Ruemmler (HP), Tom Rokicki (HP), and John Weitz (Digidesign).

We would also like to thank all of the people that have run the benchmark and contributed their results; none of this would have been possible without their assistance.

Our thanks to all of the free software community for tools that were used during this project. `lmbench` is currently developed on Linux, a copylefted Unix written by Linus Torvalds and his band of happy hackers. This paper and all of the `lmbench` documentation was produced using the `groff` suite of tools written by James Clark. Finally, all of the data processing of the results is done with `perl` written by Larry Wall.

Sun Microsystems, and in particular Paul Borrill, supported the initial development of this project. Silicon Graphics has supported ongoing development that turned into far more time than we ever imagined. We are grateful to both of these companies for their financial support.

10. Obtaining the benchmarks

The benchmarks are available at http://reality.sgi.com/employees/lm_engr/lmbench.tgz as well as via a mail server. You may request the latest version of `lmbench` by sending email to archives@slovak.engr.sgi.com with `lmbench-current*` as the subject.

References

- [Chen94] P. M. Chen and D. A. Patterson, "A new approach to I/O performance evaluation – self-scaling I/O benchmarks, predicted I/O performance," *Transactions on Computer Systems*, 12 (4), pp. 308-339, November 1994.
- [Chen93] Peter M. Chen and David Patterson, "Storage performance – metrics and benchmarks," *Proceedings of the IEEE*, 81 (8), pp. 1151-1165, August 1993.
- [Fenwick95] David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Danial Wissell, "The AlphaServer 8000 series: high-end server platform development," *Digital Technical Journal*, 7 (1), pp. 43-65, August 1995.
- [Hennessy96] John L. Hennessy and David A. Patterson, "Computer Architecture A Quantitative Approach, 2nd Edition," Morgan Kaufman, 1996.
- [McCalpin95] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture newsletter*, to appear, December 1995.
- [McVoy91] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a Unix File System," pp. 33-43, Proceedings USENIX Winter Conference, January 1991.
- [Ousterhout90] John K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?," pp. 247-256, Proceedings USENIX Summer Conference, June 1990.
- [Park90] Arvin Park and J. C. Becker, "IOStone: a synthetic file system benchmark," *Computer Architecture News*, 18 (2), pp. 45-52, June 1990.
- [Saavedra95] R.H. Saavedra and A.J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, 44 (10), pp. 1223-1235, October 1995.
- [Stallman89] Free Software Foundation, Richard Stallman, "General Public License," 1989. Included with `lmbench`.
- [Toshiba94] Toshiba, "DRAM Components and Modules," pp. A59-A77, C37-C42, Toshiba America Electronic Components, Inc., 1994.
- [Wolman89] Barry L. Wolman and Thomas M. Olson, "IOBENCH: a system independent IO benchmark," *Computer Architecture News*, 17 (5), pp. 55-70, September 1989.

Biographical information

Larry McVoy currently works Silicon Graphics in the Networking Systems Division on high performance networked file systems and networking architecture. His computer interests include hardware architecture, software implementation and architecture, performance issues, and free (GPLed) software issues. Previously at Sun, he was the architect for the SPARC Cluster product line, redesigned and wrote an entire source management system (now productized as TeamWare), implemented UFS clustering, and implemented all of the Posix 1003.1 support in SunOS 4.1. Concurrent with Sun work, he lectured at Stanford University on Operating Systems. Before Sun, he worked on the ETA systems supercomputer Unix port. He may be reached by electronically via lm@sgi.com or by phone at (415) 933-1804.

Carl Staelin works for Hewlett-Packard Laboratories in the External Research Program. His research interests include network information infrastructures and high performance storage systems. He worked for HP at U.C. Berkeley on the 4.4BSD LFS port, the HighLight hierarchical storage file system, the Mariposa distributed database, and the NOW project. He received his PhD in Computer Science from Princeton University in 1991 in high performance file system design. He may be reached electronically via staelin@hpl.hp.com.