lmbench3: measuring scalability

Carl Staelin
HP Laboratories Israel
HPL-2002-313
November 8th , 2002*

lmbench, benchmarking, distributed systems, parallel systems, multi-processor systems

`lmbench3` extends the `lmbench2` system to measure a system's performance under scalable load to make it possible to assess parallel and distributed computer performance with the same power and flexibility that `lmbench2` brought to uni-processor performance analysis. There is a new timing harness, `benchmp`, designed to measure performance at specific levels of parallel (simultaneous) load, and most existing benchmarks have been converted to use the new harness.

`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. It is also designed to make it easy for users to create additional micro-benchmarks that can measure features, algorithms, or subsystems of particular interest to the user.

* Internal Accession Date Only
Technion City, Haifa 32000, Israel

lmbench3: measuring scalability

Carl Staelin

Hewlett-Packard Laboratories Israel

ABSTRACT

`lmbench3` extends the `lmbench2` system to measure a system's performance under scalable load to make it possible to assess parallel and distributed computer performance with the same power and flexibility that `lmbench2` brought to uni-processor performance analysis. There is a new timing harness, `benchmp`, designed to measure performance at specific levels of parallel (simultaneous) load, and most existing benchmarks have been converted to use the new harness.

`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. It is also designed to make it easy for users to create additional micro-benchmarks that can measure features, algorithms, or subsystems of particular interest to the user.

1. Introduction

`lmbench` is a widely used suite of micro-benchmarks that measures important aspects of computer system performance, such as memory latency and bandwidth. Crucially, the suite is written in portable ANSI-C using POSIX interfaces and is intended to run on a wide range of systems without modification.

The benchmarks included in the suite were chosen because in the `lmbench` developer's experience, they each represent an aspect of system performance which has been crucial to an application's performance. Using this multi-dimensional performance analysis approach, it is possible to better predict and understand application performance because key aspects of application performance can often be understood as linear combinations of the elements measured by `lmbench`[4].

`lmbench3` extends the `lmbench` suite to encompass parallel and distributed system performance by measuring system performance under scalable load. This means that the user can specify the number of processes that will be executing the benchmarked feature in parallel during the measurements. It is possible to utilize this framework to develop benchmarks to measure distributed application performance, but it is primarily intended to measure the performance of multiple processes using the same system resource at the same time.

In general the benchmarks report either the latency or bandwidth of an operation or data pathway. The exceptions are generally those benchmarks that report on a specific aspect of the hardware, such as the processor clock rate, which is reported in MHz and nanoseconds.

`lmbench` consists of three major components: a timing harness, the individual benchmarks built on top of the timing harness, and the various scripts and glue that build and run the benchmarks and process the results.

1.1. `lmbench` history

`lmbench1` was written by Larry McVoy while he was at Sun Microsystems. It focussed on two measures of system performance: latency and bandwidth. It measured a number of basic operating system functions, such as file system read/write bandwidth or file creation time. It also focussed a great deal of energy on measuring data transfer operations, such as `bcopy` and `pipe` latency and bandwidth as well as raw memory latency and bandwidth.

Shortly after the `lmbench1` paper [12] was published, Aaron Brown examined the `lmbench` benchmark suite and published a detailed critique of its strengths and weaknesses[4]. Largely in response to these remarks, development of `lmbench2` began with a focus on improving the experimental design and statistical data analysis. The primary change was the development and adoption across all the benchmarks of a timing harness that incorporated loop-autosizing and clock resolution detection. In addition, each experiment was typically repeated eleven times with the median result reported to the user.

The `lmbench2`[21] timing harness was implemented through a new macro, BENCH(), that automatically manages nearly all aspects of accurately timing operations. For example, it automatically detects the minimal timing interval necessary to provide timing results within 1% accuracy, and it automatically repeats most experiments eleven times and reports the median result.

lmbench3 focussed on extending lmbench's functionality along two dimensions: measuring multi-processor scalability and measuring basic aspects of processor micro-architecture.

An important feature of multi-processor systems is their ability to scale their performance. While lmbench1 and lmbench2 measure various important aspects of system performance, they cannot measure performance with more than one client process active at a time. Consequently, measuring performance of multi-processor and clustered systems as a function of scalable load was impossible using those tools.

lmbench3 took the ideas and techniques developed in the earlier versions and extended them to create a new timing harness which can measure system performance under parallel, scalable loads.

lmbench3 also includes a version of John McCalpin's STREAM benchmarks. Essentially the STREAM kernels were placed in the new lmbench timing harness. Since the new timing harness also measures scalability under parallel load, the lmbench3 STREAM benchmarks include this capability automatically.

Finally, lmbench3 includes a number of new benchmarks which measure various aspects of the processor architecture, such as basic operation latency and parallelism, to provide developers with a better understanding of system capabilities. The hope is that better informed developers will be able to better design and evaluate performance critical software in light of their increased understanding of basic system performance.

2. Prior Work

Benchmarking is not a new field of endeavor. There are a wide variety of approaches to benchmarking, many of which differ greatly from that taken by lmbench.

One common form of benchmark is to take an important application or application and worklist, and to measure the time required to complete the entire task. This approach is particularly useful when evaluating the utility of systems for a single and well-known task.

Other benchmarks, such as SPECint, use a variation on this approach by measuring several applications and combining the results to predict overall performance. SPEChpc96 [22] extends this approach to the parallel and distributed domain by measuring the performance of a selected parallel applications built on top of MPI and/or PVM.

Another variation takes the "kernel" of an important application and measures its performance, where the "kernel" is usually a simplification of the most expensive portion of a program. Dhrystone [23] is an example of this type of benchmark as it measures the performance of important matrix operations and was often used to predict system performance for numerical operations.

Banga developed a benchmark to measure HTTP server performance which can accurately measure server performance under high load[1]. Due to the idiosyncracies of the HTTP protocol and TCP design and implementation, there are generally operating system limits on the rate at which a single system can generate independent HTTP requests. However, Banga developed a system which can scalably present load to HTTP servers in spite of this limitation[2].

John McCalpin's STREAM benchmark measures memory bandwidth during four common vector operations[11]. It does not measure memory latency, and strictly speaking it does not measure raw memory bandwith although memory bandwidth is crucial to STREAM performance. More recently, STREAM has been extended to measure distributed application performance using MPI to measure scalable memory subsystem performance, particularly for multi-processor machines.

Prestor[16] and Saavedra[17] have developed benchmarks which analyze memory subsystem performance.

Micro-benchmarking extends the "kernel" approach, by measuring the performance of operations or resources in isolation. lmbench and many other benchmarks, such as nfsstone[20], measure the performance of key operations so users can predict performance for certain workloads and applications by combining the performance of these operations in the right mixture.

Saavedra[18] takes the micro-benchmark approach and applies it to the problem of predicting application performance. They analyze applications or other benchmarks in terms of their ‘‘narrow spectrum benchmarks’’ to create a linear model of the application's computing requirements. They then measure the computer system's performance across this set of micro-benchmarks and use a linear model to predict the application's performance on the computer system. Seltzer[19] applied this technique using the features measured by lmbench as the basis for application prediction.

Benchmarking I/O systems has proven particularly troublesome over the years, largely due to the strong non-linearities exhibited by disk systems. Sequential I/O provides much higher bandwidth than non-sequential I/O, so performance is highly dependent on the workload characteristics as well as the file system's ability to capitalize on available sequentiality by laying out data contiguously on disk.

I/O benchmarks have a tendency to age poorly. For example, IOStone[15], IOBench[24], and the Andrew benchmark[8] used fixed size datasets, whose size was significant at the time, but which no longer measure I/O performance as the data can now fit in the processor cache of many modern machines.

The Andrew benchmark attempts to separately measure the time to create, write, re-read, and then delete a large number of files in a hierarchical file system.

Bonnie[3] measures sequential, streaming I/O bandwidth for a single process, and random I/O latency for multiple processes.

Peter Chen developed an adaptive harness for I/O benchmarking[6][5], which defines I/O load in terms of five parameters, uniqueBytes, sizeMean, readFrac, seqFrac, and processNum. The benchmark then explores the parameter space to measure file system performance in a scalable fashion.

Parkbench[14] is a benchmark suite that can analyze parallel and distributed computer performance. It contains a variety of benchmarks that measure both aspects of system performance, such as communication overheads, and distributed application kernel performance. Parkbench contains benchmarks from both NAS[13] and Genesis[7].

3. Timing Harness

The first, and most crucial element in extending `lmbench2` so that it could measure scalable performance, was to develop a new timing harness that could accurately measure performance for any given load. Once this was done, then each benchmark would be migrated to the new timing harness.

The harness is designed to accomplish a number of goals:

1. during any timing interval of any child it is guaranteed that all other child processes are also running the benchmark
2. the timing intervals are long enough to average out most transient OS scheduler affects
3. the timing intervals are long enough to ensure that error due to clock resolution is negligible
4. timing measurements can be postponed to allow the OS scheduler to settle and adjust to the load
5. the reported results should be representative and the data analysis should be robust
6. timing intervals should be as short as possible while ensuring accurate results

Developing an accurate timing harness with a valid experimental design is more difficult than is generally supposed. Many programs incorporate elementary timing harnesses which may suffer from one or more defects, such as insufficient care taken to ensure that the benchmarked operation is run long enough to ensure that the error introduced by the clock resolution is insignificant. The basic elements of a good timing harness are discussed in Staelin[21].

The new timing harness must also collect and process the timing results from all the child processes so that it can report the representative performance. It currently reports the median performance over all timing intervals from all child processes. It might perhaps be argued that it should report the median of the medians.

When running benchmarks with more than one child, the harness must first get a baseline estimate of performance by running the benchmark in only one process using the standard `lmbench` timing interval, which is often 5,000 microseconds. Using this information, the harness can compute the average time per iteration for a single process, and it uses this figure to compute the number of iterations necessary to ensure that each child runs for at least one second.

3.1. Clock resolution

`lmbench` uses the `gettimeofday` clock, whose interface resolves time down to 1 microsecond. However, many system clock's resolution is only 10 milliseconds, and there is no portable way to query the system to discover the true clock resolution.

The problem is that the timing intervals must be substantially larger than the clock resolution in order to ensure that the timing error doesn't impact the results. For example, the true duration of an event measured with a 10 milli-second clock can vary ±10 milli-seconds from the true time, assuming that the reported time is always a truncated version of the true time. If the clock itself is not updated precisely, the true error can be even larger. This implies that timing intervals on these systems should be at least 1 second.

However, the `gettimeofday` clock resolution in most modern systems is 1 microsecond, so timing intervals can as small as a few milli-seconds without incurring significant timing errors related to clock resolution.

Since there is no standard interface to query the operating system for the clock resolution, `lmbench` must experimentally determine the appropriate timing interval duration which provides results in a timely fashion with a negligible clock resolution error.

3.2. Coordination

Developing a timing harness that correctly manages N processes and accurately measures system performance over those same N processes is significantly more difficult than simply measuring system performance with a single process because of the asynchronous nature of parallel programming.

In essence, the new timing harness needs to create N jobs, and measure the average performance of the target subsystem while all N jobs are running. This is a standard problem for parallel and distributed programming, and involves starting the child processes and then stepping through a handshaking process to ensure that all children have started executing the benchmarked operation before any child starts taking measurements.

Parent	Child
• start up P child processes	
• wait for P *ready* signals	• run benchmark operation for a little while
↓	• send a *ready* signal
• on reciept of *ready* signals, sleep for *warmup* `micro`s	• run benchmark operation while polling for a *go* signal
• send *go* signal to P children	↓
• wait for P *done* signals	• on receipt of *go* signal, begin timing benchmark operation
↓	• send a *done* signal
• one receipt of *done* signals, iterate through children sending *results* signal and gathering results	• run benchmark operation while polling for a *results* signal
• collate results	• on receipt of *results* signal, send timing results and wait for *exit* signal
• send *exit* signal	↓
	• exit

Table 1. Timing harness sequencing

Table 1 shows how the parent and child processes coordinate their activities to ensure that all children are actively running the benchmark activity while any child could be taking timing measurements.

The reason for the separate "exit" signal is to ensure that all properly managed children are alive until the parent allows them to die. This means that any SIGCHLD events that occur before the "exit" signal indicate a child failure.

3.3. Accuracy

The new timing harness also needs to ensure that the timing intervals are long enough for the results to be representative. The previous timing harness assumed that only single process results were important, and it was able to use timing intervals as short as possible while ensuring that errors introduced by the clock resolution were negligible. In many instances this meant that the timing intervals were smaller than a single scheduler time slice. The new timing harness must run benchmarked operations long enough to ensure that timing intervals are longer than a single scheduler time slice. Otherwise, you can get results which are complete nonsense. For example, running several copies of an `lmbench2` benchmark on a uni-processor machine will often report that the per-process performance with *N* jobs running in parallel is equivalent to the performance with a single job running![1]

In addition, since the timing intervals now have to be longer than a single scheduler time slice, they also need to be long enough so that a single scheduler time slice is insignificant compared to the timing interval. Otherwise the timing results can be dramatically affected by small variations in the scheduler's behavior.

Currently `lmbench` does not measure the scheduler timeslice; the design blithely assumes that timeslices are generally on the order of 10-20ms, so one second timing intervals are sufficient. Some schedulers may utilize longer time slices, but this has not (yet) been a problem.

3.4. Resource consumption

One important design goal was that resource consumption be constant with respect to the number of child processes. This is why the harness uses shared pipes to communicate with the children, rather than having a separate set of pipes to communicate with each child. An early design of the system utilized a pair of pipes per child for communication and synchronization between the master and slave processes. However, as the number of child processes grew, the fraction of system resources consumed by the harness grew and the additional system overhead could start to interfere with the accuracy of the measurements.

Additionally, if the master has to poll (`select`) *N* pipes, then the system overhead of that operation also scales with the number of children.

3.5. Pipe atomicity

Since all communication between the master process and the slave (child) processes is done via a set of shared pipes, we have to ensure that we never have a situation where the message can be garbled by the intermingling of two separate messages from two separate children. This is ensured by either using pipe operations that are guaranteed to be atomic on all machines, or by coordinating between processes so that at most one process is writing at a time.

The atomicity guarantees are provided by having each client communicate synchronization states in one-byte messages. For example, the signals from the master to each child are one-byte messages, so each child only reads a single byte from the pipe. Similarly, the responses from the children back to the master are also one-byte messages. In this way no child can receive partial messages, and no message can be interleaved with any other message.

However, using this design means that we need to have a separate pipe for each *barrier* in the process, so

[1] This was discovered by someone who naively attempted to parallelize `lmbench2` in this fashion, and I received a note from the dismayed developer describing the failed experiment.

the master uses three pipes to send messages to the children, namely: *start_signal*, *result_signal*, and *exit_signal*. If a single pipe was used for all three barrier events, then it is possible for a child to miss a signal, or if the signal is encoded into the message, then it is possible for a child to infinite loop pulling a signal off the pipe, recognizing that it has already received that signal so that it needs to push it back into the pipe, and then then re-receiving the same message it just re-sent.

However, all children share a single pipe to send data back to the master process. Usually the messages on this pipe are single-byte signals, such as *ready* or *done*. However, the timing data results need to be sent from the children to the master and they are (much) larger than a single-byte message. In this case, the timing harness sends a single-byte message on the *result_signal* channel, which can be received by at most one child process. This child then knows that it has sole ownership of the response pipe, and it writes its entire set of timing results to this pipe. Once the master has received all of the timing results from a single child, it sends the next one-byte message on the *result_signal* channel to gather the next set of timing results.

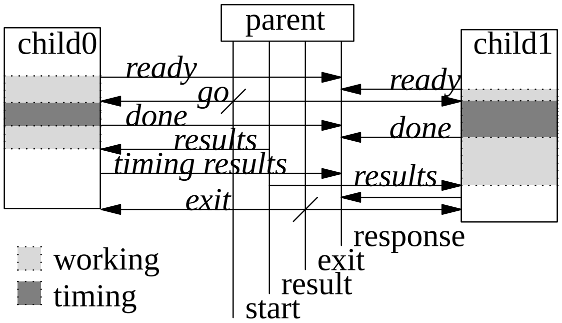

Figure 1. Control signals

The design of the signals is shown in Figure 1.

3.6. Benchmark initialization

By allowing the benchmark to specify an initialization routine that is run in the child processes, the new timing harness allows benchmarks to do either or both global initializations that are shared by all children and specific per-child initializations that are done independently by each child. Global initialization is done in the master process before the `benchmp` harness is called, so the state is preserved across the `fork` operations. Per-child initialization is done inside the `benchmp` harness by the optional initialization routine and is done after the `fork` operation.

Similarly, each benchmark is allowed to specify a cleanup routine that is run by the child processes just before exiting. This allows the benchmark routines to release any resources that they may have used during the benchmark. Most system resources would be automatically released on process exit, such as file descriptors and shared memory segments, but some resources such as temporary files might need to be explicitly released by the benchmark.

3.7. Scheduler transients

Particularly on multi-processor systems, side-effects of process migration can dramatically affect program runtimes. For example, if the processes are all initially assigned to the same processor as the parent process, and the timing is done before the scheduler migrates the processes to other available processors, then the system performance will appear to be that of a uniprocessor. Similarly, if the scheduler is over-enthusiastic about re-assigning processes to processors, then performance will be worse than necessary because the processes will keep encountering cold caches and will pay exhorbitant memory access costs.

The first case is a scheduler transient, and users may not want to measure such transient phenomena if their primary interest is in predicting performance for long-running programs. Conversely, that same user would be extraordinarily interested in the second phenomena. The harness was designed to allow users to specify that the benchmarked processes are run for long enough to (hopefully) get the scheduler past the transient startup phase, so it can measure the steady-state behavior.

3.8. Data analysis

Analyzing the data to produce representative results is a crucial step in the benchmarking process. `lmbench` generally reports the *median* result for 11 measurements. Most benchmarks report the results of a single measurement[8], an average of several results[11], or a trimmed mean[4].

Since `lmbench` is able to use timing intervals that are often smaller than a scheduler time slice when measuring single-process performance, the raw timing results are often severely skewed. Often most results cluster around a single value a small number of outliers with significantly larger values. The median is preferable to the mean when the data can be very skewed[9]. Since the timing intervals are significantly longer when the desired load is larger than a single process, the results tend not to be as badly skewed. In these cases we could use the *mean* instead, but we decide to use a uniform statistical framework, so we usually use the median.

In some instances, however, `lmbench` internally uses the *minimum* rather than the median, such as in `mhz`. In those instances, we are not trying to find the *representative* value, but rather the *minimum* value. There are only a few sources of error which could cause a the measured timing result to be shorter than the true elapsed time: the system clock is adjusted, or round-off error in the clock resolution. The timing interval duration is set to ensure that the round-off error is bounded to 1% of the timing interval, and we blithely assume that people don’t reset their system clocks while benchmarking their systems.

`lmbench` does not currently report any statistics representing measurement variation, such as the difference between the first and third quartiles. This is an enhancement under active consideration.

4. Interface

Unfortunately we had to move away from the macro-based timing harness used in `lmbench2` and migrate to a function-based system because the macros were too large for some C pre-processors.

```
typedef void (*bench_f)(iter_t iters,
             void* cookie);
typedef void (*support_f)(void* cookie);

extern void benchmp(support_f initialize,
        bench_f benchmark,
        support_f cleanup,
        int enough,
        int parallel,
        int warmup,
        int repetitions,
        void* cookie);

extern uint64 gettime();
extern uint64 get_n();
extern void nano(char* s, uint64 n);
extern void micro(char* s, uint64 n);
extern void mb(uint64 bytes);
```

Figure 2. Programming interface

Figure 2 shows the key elements of the new timing harness and result reporting interface. A brief description of the `benchmp` parameters:

enough

Enough can be used to ensure that a timing interval is at least ’enough’ microseconds in duration. For most benchmarks this should be zero, but some benchmarks have to run for more time due to startup effects or other transient behavior.

parallel

is simply the number of instances of the benchmark that will be run in parallel on the system.

warmup

can be used to force the benchmark to run for warmup microseconds before the system starts making timing measurements. Note that it is a lower bound, not a fixed value, since it is simply the time that the parent sleeps after receiving the last "ready" signal from each child (and before it sends the "go" signal to the children).

repetitions

is the number of times the experiment should be repeated. The default is eleven.

cookie

is a pointer that can be used by the benchmark writer to pass in configuration information, such as buffer size or other parameters needed by the inner loop. In `lmbench3` it is generally used to point to a structure containing the relevant configuration information.

`gettime` returns the median timing interval duration, while `get_n` returns the number of iterations executed during that timing interval.

`nano` and `micro` print the passed string latency followed by the latency in terms of nanoseconds and microseconds respectively. The latency is computed as *gettime*()/*n*, where *n* is the passed parameter. The reason *n* is passed as a parameter is because the benchmark can actually execute the operation of interest multiple times during a single iteration. For example, the memory latency benchmarks typically repeat the memory load operation a hundred times inside the loop, so the actual number of operations is 100 × *get_n*(), and it is this value that should be passed to `nano` or `micro`.

`mb` reports the bandwidth in MB/s when given the total number of bytes processed during the timing interval. Note that for scalable benchmarks that process *size* bytes per iteration, the total number of bytes processed is *get_n*() × *parallel* × *size*.

```
#include "bench.h"

void
bench(iter_t iters, void* cookie)
{
    while (iters-- > 0) {
        getppid();
    }
}

int
main(int argc, char* argv[])
{
    benchmp(NULL, bench, NULL,
        0, 1, 0, TRIES, NULL);
    nano("getppid", get_n());
    return(0);
}
```

Figure 3. A sample benchmark

Figure 3 shows a sample benchmark that measures the latency of the `getppid` system call using this timing harness. Since there is no setup or cleanup needed for this benchmark, the *initialize* and *cleanup* parameters are NULL. The *bench* routine simply calls `getppid` as many times as requested, and the rest of the parameters, *enough*, *parallel*, *warmup*, *repetitions*, and *cookie* are given with the default values.

5. Benchmarks

`lmbench` contains a large number of micro-benchmarks that measure various aspects of hardware and operating system performance. The benchmarks generally measure latency or bandwidth, but some new benchmarks also measure instruction-level parallelism.

Table 2 contains the full list of micro-benchmarks in `lmbench3`. Benchmarks that were converted to measure performance under scalable load are shown in italics, while the remaining benchmarks are shown with normal typeface. A detailed description of most benchmarks can be found in[12].

6. Scaling Benchmarks

There are a number of issues associated with converting single-process benchmarks with a single process to scalable benchmarks with several independent processes, in addition to the various issues addressed by the timing harness. Many of the benchmarks consume or utilize system resources, such as memory or network bandwidth, and a careful assessment of the likely resource contention issues is necessary to ensure that the benchmarks measure important aspects of system performance and not artifacts of artificial resource contention.

For example, the Linux 2.2 and 2.4 kernels use a single lock to control access to the kernel data structures for a file. This means that multiple processes accessing that file will have their operations serialized by that lock. If one is interested in how well a system can handle multiple independent accesses to separate files and if the child processes all access the same file, then this file sharing is an artificial source of contention with potentially dramatic effects on the benchmark results.

6.1. File System

A number of the benchmarks measure aspects of file system performance, such as `bw_file_rd`, `bw_mmap_rd`, `lat_mmap`, and `lat_pagefault`. It is not immediately apparent how these benchmarks should be extended to the parallel domain. For example, it may be important to know how file system performance scales when multiple processes are reading the same file, or when multiple processes are reading different files. The first case might be important for large, distributed scientific calculations, while the second might be more important for a web server.

However, for the operating system, the two cases are significantly different. When multiple processes access the same file, access to the kernel data structures for that file must be coordinated and so contention and locking of those structures can impact performance, while this is less true when multiple processes access different files.

Name	Measures
	Bandwidth
bw_file_rd	`read` and then load into processor
bw_mem	read, write, and copy data to/from memory
bw_mmap_rd	read from `mmap`'ed memory
bw_pipe	`pipe` inter-process data copy
bw_tcp	TCP inter-process data copy
bw_unix	UNIX inter-process
	Latency
lat_connect	TCP connection
lat_ctx	context switch via `pipe`-based ‘‘hot-potato’’ token passing
lat_dram_page	DRAM page open
lat_fcntl	`fcntl` file locking ‘‘hot-potato’’ token passing
lat_fifo	FIFO ‘‘hot-potato’’ token passing
lat_fs	file creation and deletion
lat_http	http GET request latency
lat_mem_rd	memory read
lat_mmap	`mmap` operation
lat_ops	basic operations (*xor*, *add*, *mul*, *div*, *mod*) on (relevant) basic data types (*int*, *int64*, *float*, *double*)
lat_pagefault	page fault handler
lat_pipe	`pipe` ‘‘hot-potato’’ token passing
lat_pmake	time to complete *N* parallel jobs that do *usecs*-worth of work
lat_proc	procedure call overhead and process creation using `fork`, `fork` and `execve`, and `fork` and `/bin/sh`
lat_rpc	SUN RPC procedure call
lat_select	`select` operation
lat_sem	semaphore ‘‘hot-potato’’ token passing
lat_sig	signal handle installation and handling
lat_syscall	`open`, `close`, `getppid`, `write`, `stat`, `fstat`
lat_tcp	TCP ‘‘hot-potato’’ token passing
lat_udp	UDP ‘‘hot-potato’’ token passing
lat_unix	UNIX ‘‘hot-potato’’ token passing
lat_unix_connect	UNIX socket connection
lat_usleep	`usleep`, `select`, `pselect`, `nanosleep`, `setitimer` timer resolution
	Other
disk	zone bandwidths and seek times
line	cache line size
lmdd	*dd* clone
par_mem	memory subsystem ILP
par_ops	basic operation ILP
stream	STREAM clones
tlb	TLB size

Table 2. `lmbench` micro-benchmarks

In addition, there are any number of issues associated with ensuring that the benchmarks are either measuring operating system overhead (e.g., that no I/O is actually done to disk), or actually measuring the system's I/O performance (e.g., that the data cannot be resident in the buffer cache). Especially with file system related benchmarks, it is very easy to develop benchmarks that compare apples and oranges (e.g., the benchmark includes the time to flush data to disk on one system, but only includes the time to flush a portion of data to disk on another system).

`lmbench3` allows the user to measure either case as controlled by a command-line switch. When measuring accesses to independent files, the benchmarks first create their own private copies of the file, one for each child process. Then each process accesses its private file. When measuring accesses to a single file, each child simply uses the designated file directly.

6.2. Context Switching

Measuring context switching accurately is a difficult task. `lmbench1` and `lmbench2` measured context switch times via a "hot-potato" approach using pipes connected in a ring. However, this experimental design heavily favors schedulers that do "hand-off" scheduling, since at most one process is active at a time. Consequently, it is not really a good benchmark for measuring scheduler overhead in multi-processor machines.

The design currently used in `lmbench3` is to create *N* `lmbench2`-style process rings and to measure the context switch times with all *N* rings running in parallel. This does extend the `lmbench2` context switch benchmark to a scalable form, but it still suffers from the same weaknesses.

One approach that was considered was to replace the ring with a star formation, so the master process would send tokens to each child and then wait for them all to be returned. This has the advantage that more than one process is active at a time, reducing the sensitivity to "hand-off" scheduling. However, this same feature can cause problems on a multi-processor system because several of the context switches and working set accesses can occur in parallel.

The design and methodology for measuring context switching and scheduler overhead need to be revisited so that it can more accurately measure performance for multi-processor machines.

7. Stream

`lmbench3` includes a new micro-benchmark, `stream` which measures the performance of John McCalpin's STREAM benchmark kernels for both STREAM version 1 [11] and version 2[10]. This benchmark faithfully recreates each of the kernel operations from both STREAM benchmarks, and because of the powerful new timing harness it can easily measure memory system scalability.

Table 3 is based on McCalpin's tables[11][10] and shows the four kernels for each version of the `stream` benchmark. Note that the *read* columns include numbers in parentheses, which represent the average number of bytes read into the cache as a result of the write to that variable[2]. Cache lines are almost invariably bigger than a single double, and so when a write miss occurs the cache will read the line from memory and then modify the selected bytes. Sometimes vector instructions such as SSE and 3DNow can avoid this load by writing an entire cache line at once.

Stream				
Kernel	Code	Bytes		FL
		rd	wr	OPS
COPY	$a[i] = b[i]$	8(+8)	8	0
SCALE	$a[i] = q \times b[i]$	8(+8)	8	1
ADD	$a[i] = b[i] + c[i]$	16(+8)	8	1
TRIAD	$a[i] = b[i] + q \times c[i]$	16(+8)	8	2(-1)

Stream2				
Kernel	Code	Bytes		FL
		rd	wr	OPS
FILL	$a[i] = q$	0(+8)	8	0
COPY	$a[i] = b[i]$	8(+8)	8	0
DAXPY	$a[i] = a[i] + q \times b[i]$	16	8	2(-1)
SUM	$sum = sum + a[i]$	8	0	1

Table 3. Stream operations

In addition, some architectures support multiply-add instructions which can do both the multiply and add operations for TRIAD and DAXPY in a single operation, so the physical FLOPS count would be 1 for these architectures on these instructions. The numbers in parenthesis in the *FLOPS* column reflect this reduction in FLOPS count.

Following the STREAM bandwidth reporting conventions, the `lmbench` STREAM benchmarks report their results as bandwidth results (MB/s) computed as a function of the amount of data explicitly read or written by the benchmark. For example, *copy* and *scale* copy data from one array to the other, so the bandwidth is measured as a function of the amount of data read plus the amount of data written, or the sum of the two array sizes. Similarly, *sum*, *triad*, and *daxpy* operate on three arrays, so the amount of data transferred is the sum of the sizes of the three arrays. Note that the actual amount of data that is transferred by the system may be larger because in the write path the cache may need to fetch (read) the cache line before a portion of it is overwritten by dirty data.

8. Unscalable benchmarks

There are a number of benchmarks which either did not make sense for scalable load, such as `mhz`, or which could not be extended to measure scalable load due to other constraints, such as `lat_connect`.

[2] This number is independent of the cache line size because the STREAM uses dense arrays, so the cost is amortized over the subsequent operations on the rest of the line.

mhz measures the processor clock speed, which is not a scalable feature of the system, so it doesn't make any sense to create a version of it that measures scalable performance.

More specifically, lat_connect measures the latency of connecting to a TCP socket. TCP implementations have a timeout on sockets and there is generally a fixed size queue for sockets in the TIMEOUT state. This means that once the queue has been filled by a program connecting and closing sockets as fast as possible, then all new socket connections have to wait TIMEOUT seconds. Needless to say, this gives no insight into the latency of socket creation per se, but is rather a boring artifact. Since the lmbench2 version of the benchmark can run for very short periods of time, it generally does not run into this problem and is able to correctly measure TCP connection latency.

Any scalable version of the benchmark needs each copy to run for at least a second, and there are *N* copies creating connections as fast as possible, so it would essentially be guaranteed to run into the TIMEOUT problem. Consequently, lat_connect was not enhanced to measure scalable performance.

9. Results

The results presented here were obtained using lmbench version 3.0-a2 under Linux 2.4.18-6mdk on a two processor 450MHz PIII running a stock Mandrake 8.2 Linux 2.4.18 kernel.

Benchmark	Latency (μs)	
	1 process	2 processes
null call	0.79	0.81
null I/O	1.39	2.39
stat	9.26	25.9
open/close	11.7	27.1
select (TCP)	55.3	58.6
signal install	1.89	1.95
signal handler	6.34	7.21
fork process	793.	868.
exec process	2474	2622
sh process	24.K	25.K
pipe	17.7	23.3
unix socket	51.6	37.6
UDP	70.2	70.6
TCP	91.2	92.3
rpc (UDP)	120.0	120.4
rpc (TCP)	157.1	159.1

Table 4. Latency results

Benchmark	Load		
	1	2	2clone
bw_file_rd	151.04	266.74	273.51
bw_mmap_rd	316.08	480.02	482.57
lat_mmap	615	878	786
lat_pagefault	2.9802	3.9159	3.4589

Table 6. File bandwidth results

Benchmark	Bandwidth (MB/s)	
	1 process	2 processes
pipe	155	268
unix socket	142	179
TCP	57.5	57.8
bcopy(libc)	134	175
bcopy(hand)	144	174
memory read	319	486
memory write	199	202
STREAM copy	288.68	367.99
STREAM scale	290.39	369.08
STREAM sum	337.75	415.54
STREAM triad	246.90	380.09
STREAM2 fill	198.96	276.28
STREAM2 copy	288.55	359.93
STREAM2 daxpy	318.98	493.79
STREAM2 sum	354.03	512.05

Table 5. Bandwidth results

Table 4 shows the latency of various system and communication operations for both 1 and 2 process loads, while Table 5 shows the bandwidth of various data operations and Table 6 shows how various file system operations scale. Table 6 shows system performance with one process, two processes sharing the same file, and two processes accessing their own files.

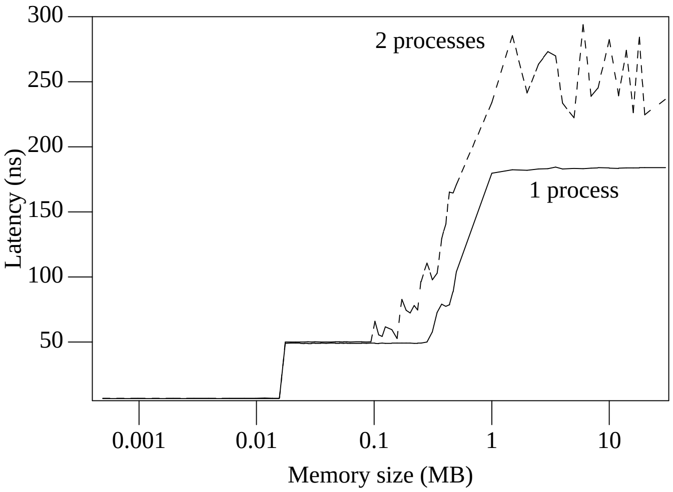

Figure 4. Memory subsystem performance

Figure 4 shows the memory latency curves with 32 byte strides for one and two process loads versus memory size.

10. Conclusion

lmbench is a useful, portable micro-benchmark suite designed to measure important aspects of system performance. lmbench3 adds a number of important extensions, such as the ability to measure system scalability.

The benchmarks are available via ftp from:

http://ftp.bitmover.com/lmbench

11. Acknowledgments

Many people have provided invaluable help and insight into the benchmarks. We especially thank: Eric Anderson (HP), Bruce Chapman (SUN), Larry McVoy (BitMover), David Mosberger (HP), Wayne Scott (BitMover), John Wilkes (HP), and Mitch Wright (HP).

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.

References

[1] Guarav Banga and Peter Druschel, ‘‘Measuring the capacity of a web server’’ in *Proceedings USENIX Symposium on Internet Technologies and Systems,* Monterey, CA, December 1997.

[2] Guarav Banga and Jeffrey C. Mogul, ‘‘Scalable kernel performance for internet servers under realistic loads’’ in *Proceedings of the 1998 USENIX Annual Technical Conference,* New Orleans, LA, June 1998.

[3] Tim Bray, ‘‘Bonnie benchmark,’’ 1990.

[4] Aaron Brown and Margo Seltzer, ‘‘Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture’’ in *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* pp. 214-224, Seattle, WA, June 1997.

[5] 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.

[6] Peter M. Chen and David Patterson, ‘‘Storage performance − metrics and benchmarks,’’ *Proceedings of the IEEE,* 81 (8), pp. 1151-1165, August 1993.

[7] Ian Glendinning, ‘‘GENESIS distributed memory benchmark suite.’’ http://wotug.ukc.ac.uk/parallel/performance/benchmarks/genesis.

[8] J. Howard, M. Kazar, S. Menees, S. Nichols, M. Satyanrayanan, R. Sidebotham, and M. West, ‘‘Scale and performance in a distributed system,’’ *ACM Transactions on Computer Systems,* 6 (1), pp. 51-81, February 1988.

[9] Raj Jain, ‘‘The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,’’ Wiley-Interscience, New York, NY, April 1991.

[10] John D. McCalpin, ‘‘The STREAM2 home page.’’

[11] John D. McCalpin, ‘‘Memory bandwidth and machine balance in current high performance computers,’’ *IEEE Technical Committee on Computer Architecture newsletter,* December 1995.

[12] Larry McVoy and Carl Staelin, ‘‘lmbench: Portable tools for performance analysis’’ in *Proceedings USENIX Winter Conference,* pp. 279-284, San Diego, CA, January 1996.

[13] NASA Advanced Supercomputing Division, NASA Ames Research Center, ‘‘NAS parallel benchmarks.’’ http://www.nas.nasa.gov/NAS/NPB.

[14] PARallel Kernels and BENCHmarks committee, ‘‘PARKBENCH,’’ 2002. http://www.netlib.org/parkbench/.

[15] Arvin Park and J. C. Becker, ‘‘IOStone: a synthetic file system benchmark,’’ *Computer Architecture News,* 18 (2), pp. 45-52, June 1990.

[16] Uros Prestor, ‘‘Evaluating the memory performance of a ccNUMA system,’’ Department of Computer Science, University of Utah, May 2001.

[17] 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.

[18] Rafael H. Saavedra-Barrera, ‘‘CPU Performance evaluation and execution time prediction using narrow spectrum benchmarking,’’ Department of Computer Science, University of California at Berkeley, 1992.

[19] Margo Seltzer, David Krinsky, Keith Smith, and Xiolan Zhang, ‘‘The case for application-specific benchmarking’’ in *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems,* Rico, AZ, 1999.

[20] Barry Shein, Mike Callahan, and Paul Woodbury, ‘‘NFSSTONE: A network file server performance benchmark’’ in *Proceedings USENIX Summer Conference,* pp. 269-275, Baltimore, MD, June 1989.

[21] Carl Staelin and Larry McVoy, ‘‘mhz: Anatomy of a microbenchmark’’ in *Proceedings USENIX Annual Technical Conference,* pp. 155-166, New Orleans, LA, June 1998.

[22] Standard Performance Evaluation Corporation, ‘‘SPEC HPC96 benchmark,’’ 1996. http://www.specbench.org/hpg/hpc96/.

[23] R.P. Weicker, ‘‘Dhrystone: A synthetic systems programming benchmark,’’ *CACM,* 27 (10), pp. 1013-1030, 1984.

[24] Barry L. Wolman and Thomas M. Olson, ‘‘IOBENCH: a system independent IO benchmark,’’ *Computer Architecture News,* 17 (5), pp. 55-70, September 1989.