lmbench – an extensible micro-benchmark suite

Carl Staelin
HP Laboratories Israel
HPL-2004-213
November 22, 2004*

micro-benchmarking, performance analysis, measurement

lmbench is a powerful and extensible suite of micro-benchmarks that measure a variety of important aspects of system performance. It has a powerful timing harness that manages most of the "housekeeping" chores associated with benchmarking, making it easy to create new benchmarks that analyze systems or components of specific interest to the user. In many ways *lmbench* is a Swiss army knife for performance analysis. It includes an extensive suite of micro-benchmarks that give powerful insights into system performance. For those aspects of system or application performance not covered by the suite, it is generally a simple task to create new benchmarks using the timing harness. *lmbench* is written in ANSI-C and uses POSIX interfaces, so it is portable across a wide variety of systems and architectures. It also includes powerful new tools that measure performance under scalable loads to analyze SMP and clustered system performance.

* Internal Accession Date Only
Published in *Software – Practice and Experience*
Approved for External Publication
© Copyright 2004 John Wiley & Sons Ltd.

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

lmbench — an extensible micro-benchmark suite

Carl Staelin[1]

[1] *Hewlett-Packard Laboratories Israel, Technion City, Haifa, 32000, ISRAEL*

SUMMARY

***lmbench* is a powerful and extensible suite of micro-benchmarks that measure a variety of important aspects of system performance. It has a powerful timing harness that manages most of the "housekeeping" chores associated with benchmarking, making it easy to create new benchmarks that analyze systems or components of specific interest to the user.**

In many ways *lmbench* is a Swiss army knife for performance analysis. It includes an extensive suite of micro-benchmarks that give powerful insights into system performance. For those aspects of system or application performance not covered by the suite, it is generally a simple task to create new benchmarks using the timing harness.

***lmbench* is written in ANSI-C and uses POSIX interfaces, so it is portable across a wide variety of systems and architectures. It also includes powerful new tools that measure performance under scalable loads to analyze SMP and clustered system performance.**

Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: micro-benchmarking, performance analysis, measurement

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 experience of the *lmbench* developers, Larry McVoy and Carl Staelin, 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* [1].

*Correspondence to: Hewlett-Packard, Technion City, Haifa, 32000, ISRAEL

Copyright © 2004 John Wiley & Sons, Ltd.

However, *lmbench* does not include benchmarks that measure all possible operations or combinations of operations, but it is extensible. This allows the user to create benchmarks that measure performance of particular subsystems, combinations of operations, or application components. For example, one of the *lmbench* developers was developing an image processing application where the bottleneck was a convolution operation between the input image and a fixed-size mask, so he developed a few different convolution implementations and measured their performance on different combinations of hardware and compilers and discovered that there was no single optimal implementation for all platforms.

lmbench analyzes 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.

The timing harness is the heart of the system. It manages the actual benchmarking process: starting the benchmarked activity, repeating the benchmarked activity as long as necessary to ensure accurate results, and finally managing the statistical analysis to report representative results. Great care is taken in the design and implementation of this timing harness to ensure that as many sources of experimental and measurement error are reduced or eliminated. For example, the timing harness manages the creation and coordination of the various child or slave processes when measuring performance under scalable load, as it is somewhat tricky to ensure that *all* processes are running the benchmarked operation during the time that *any* process could be measuring the performance.

This paper describes the design of the timing harness, demonstrates how to build benchmarks using the harness, and finally presents an analysis of *lmbench* results on a few machines to demonstrate the power of the benchmark suite.

Micro-benchmark suite

lmbench has a suite of micro-benchmarks that measure a variety of important aspects of system performance. Primarily the benchmarks measure either bandwidth or latency, but some of the benchmarks report other features or aspects of system performance, such as the clock speed.

A list of the micro-benchmarks included in *lmbench* within each category includes:

bandwidth file re-read using read() and mmap(), IPC communication using TCP, pipe, and unix sockets.

latency memory latency, TCP and unix socket connection, IPC communication using TCP, UDP, RPCs, pipe, and unix sockets, file creation and deletion, process creation using

Copyright © 2004 John Wiley & Sons, Ltd.

fork(), fork()+exec(), and sh(), select() on file descriptors and network sockets, mmap(), page faults, signal installation and handling, system calls, context switching, basic arithmetic operations on various data types, and overall time to complete N jobs which each do μsecs-worth of work.

other CPU clock speed, cache line size, TLB size, basic operation instruction-level parallelism, memory subsystem instruction-level parallelism, and STREAM[2] benchmarks.

Clearly, this covers a broad segment of commonly used and important features and interfaces used for control and data movement.

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*. A good overview of the various benchmarking approaches, current benchmarks, and their various shortcomings can be found in Mogul [3].

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. This approach can also be used to evaluate different implementations of a given algorithm to identify the fastest, and ATLAS [4] uses this approach to automatically develop tuned linear algebra libraries for each processor.

Other benchmarks, such as the SPEC benchmarks, use a variation on this approach by measuring several applications and combining the results to predict overall performance. SPEChpc96 [5] 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.

Kernel-based benchmarking

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 [6] 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 and Druschel [7] developed a benchmark to measure HTTP server performance which can accurately measure server performance under high load. 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 [8].

John McCalpin's STREAM benchmark is another "kernel" benchmark that measures memory bandwidth during four common vector operations [2]. It does not measure memory latency, and strictly speaking it does not measure raw memory bandwith although memory

Copyright © 2004 John Wiley & Sons, Ltd.

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. Please note that the STREAM benchmarks are completely different from the STREAMS programming model.

Micro-benchmarking

Micro-benchmarking extends the "kernel" approach, by measuring the performance of operations or resources in isolation. *lmbench* and many other benchmarks, such as nfsstone [9], 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.

Prestor [10], Saavedra and Smith [11], and Hristea et al [12] have developed benchmarks which analyze memory subsystem performance. Saavedra assumes a single-level cache, and attempts to determine various cache parameters, such as size, line size, load latency, and associativity by analyzing the latencies of various strided access patterns over the stride and footprint parameter space. Prestor extends this work in a variety of ways for non-uniform memory access (NUMA) multi-processor machines, specifically the SGI Origin family. Hristea makes distinctions between different types of memory load events, such as back-to-back loads where the system is saturated with requests and restart loads where a request is submitted while the memory subsystem is otherwise idle, and developed a family of micro-benchmarks to measure the various load latencies.

Saavedra [13] 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 [14] applied this technique using the features measured by *lmbench* as the basis for application prediction.

I/O benchmarking

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 [16], and the Andrew benchmark [17] 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.

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Bonnie [18] measures sequential, streaming I/O bandwidth for a single process, and random I/O latency for multiple processes. It uses some per-character I/O interfaces, and its output amply demonstrates why these interfaces are never used in practice.

Peter Chen developed an adaptive harness for I/O benchmarking [19, 20], 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.

Unfortunately, there are a number of issues left unexplored even by Chen, such as the slow fragmentation of data and free space in file systems over time and its effect on file system performance [21]. Other factors are usually ignored, such as the interplay of multiple I/O streams accessing a single device, which can result in two sequential streams with potentially excellent performance being converted at the device into a single "random" stream with horrendous performance. While *lmbench3* can be used to measure performance with several I/O streams, it does not address other open issues, such as file system fragmentation.

Parallel systems benchmarking

Benchmarking parallel or distributed systems is more complex than single-CPU systems because of the myriad ways in which components and subsystems can interact and communicate. In addition, there are different aspects of system performance that may be important, from overall system throughput and utilization to inter-thread communication and synchronization.

Parkbench [22] 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 [23] and Genesis [24].

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 [25] was published, Brown et al [1] examined the *lmbench* benchmark suite and published a detailed critique of its strengths and weaknesses. Largely in response to these remarks, development of *lmbench2* by Larry McVoy and Carl Staelin 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* [26] 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

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

1% accuracy, and it automatically repeats most experiments eleven times and reports the median result. The median result is reported because the timing results are often highly skewed due to scheduling artifacts and other issues. In the face of such heavy-tailed, non-uniform distributions, the most robust representative feature is typically the median result [27]. Eleven repetitions was chosen because in practice it was the smallest number of repetitions that yielded stable results.

lmbench3 was developed largely by Carl Staelin and focussed on extending *lmbench*'s functionality into measuring multi-processor scalability and basic aspects of processor micro-architecture. This required support for measuring performance with more than one client process active at a time, which was achieved by creating a new timing harness which can measure system performance under parallel, scalable loads. *lmbench3* also includes a number of new benchmarks which measure various aspects of the processor architecture, such as basic operation latency and parallelism. It is this version that is the subject of this paper.

TIMING HARNESS

In order to accurately measure performance, a timing harness is used that manages various aspects of the experimental process to ensure accurate timing results. Since *lmbench* is written in ANSI-C using POSIX interfaces, it does not have access to operating system or hardware-specific high resolution timers that may provide accuracy even down to the CPU clock tick. Rather, it must use the less accurate gettimeofday() interface which can support timers with resolutions as small as 1μsec, but many implementations (silently) provide only 10ms resolution.

In addition, to measure scalable performance, the timing harness must manage the creation of child processes to run the benchmark and coordinate also their activities to ensure that all children are executing the benchmarked activity during all measurement periods.

Finally, the timing harness must collect and collate the timing results so as to report a representative result for the benchmark. The default is that each experiment is repeated eleven times and the median result is reported. However, both the number of repetitions and the reported statistic may be changed as needed.

The timing harness interface is contained in the header file bench.h and contains three elements:

```
typedef void (*benchmp_f)(iter_t iterations, void* cookie);

extern void benchmp(benchmp_f initialize, benchmp_f benchmark,
                    benchmp_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);
```

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

```
extern void mb(uint64 bytes);
```

A brief description of the *benchmp* parameters:

initialize is an optional pointer to a function that may be used to initialize the system. This is called once with *iterations* set to zero at the very beginning, and it is called again before each timing interval with *iterations* set to the number of iterations that the benchmark will run.

benchmark is a pointer to a function that will run the benchmarked activity *iterations* times.

cleanup is an optional pointer to a function that may be used to cleanup after running the benchmark. It is called once just before exit with *iterations* set to zero, and it is called after each timing interval with *iterations* set to the number of iterations that the benchmark ran.

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 strange behavior.

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

warmup benchmarks 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 number of times the experiment should be repeated.

cookie 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 *lmbench* 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 $100 \cdot get_n()$, 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 $get_n() \cdot parallel \cdot ``size''$.

The harness is designed to accomplish a number of goals:

Copyright © 2004 John Wiley & Sons, Ltd.

- during any timing interval of any child it is guaranteed that all other child processes are also running the benchmark
- the timing intervals are long enough to average out most transient OS scheduler effects
- the timing intervals are long enough to ensure that error due to clock resolution is negligible
- timing measurements can be postponed to allow the OS scheduler to settle and adjust to the load
- the reported results should be representative and the data analysis should be robust
- timing intervals should be as short as possible while ensuring accurate results
- the benchmark should have an opportunity to initialize and cleanup any necessary data structures or other state both at the start and completion of a measurement series and before and after each timing interval

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 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.

Clock resolution

lmbench uses the *gettimeofday* clock, whose interface resolves time down to 1 microsecond. However, the resolution of many system clocks is only 10ms, 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 $\pm$10ms 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 such systems should be at least 1 second.

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

Copyright © 2004 John Wiley & Sons, Ltd.

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

lmbench determines the timing interval using an experimental method. It has a list of timing intervals, 5ms, 10ms, 50ms, and 100ms, and it attempts to identify the smallest interval which satisfies the measurement accuracy criteria. These intervals were chosen as a reasonable tradeoff between finding the absolute smallest timing interval and having a reasonably small list of intervals to test.

The system has a sample benchmark which dereferences the same pointer in a single-pointer circular chain a given number of times. This allows very good control over the amount of work done inside a timing interval. The first step is to determine how much work (N iterations in $t_N$ms) must be done by this benchmark to take approximately the desired duration. Then eleven measurements are taken for each $\delta \cdot N$ iterations, $\delta \in \{1.015, 1.02, 1.035\}$. If the median measurement for each δ, t_δ satisfies $\left|\frac{\delta \cdot t_N - t_\delta}{t_N}\right| \leq 0.0025$, then the duration is acceptable. The δ values are chosen so that the differences between successive values are 0.005 or 0.5%, and if the errors are all less than or equal to ± 0.0025 then the timing system accuracy is at least $\pm 0.5\%$.

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 parallel and distributed programming problem, 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.

Table I 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.

The signals are sent via four shared pipes. An equivalent design alternative is to use shared memory and semaphores for the inter-process communication and control.

Accuracy

The timing harness also needs to ensure that the timing intervals are long enough for the results to be representative. The *lmbench2* timing harness assumed that only single process results were important, and it was able to use timing intervals as short as possible while ensuring

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Table I. Timing harness sequencing

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

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 *lmbench3* 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!

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 assumes that timeslices are generally on the order of 10-20ms, so one second timing intervals are sufficient. Some schedulers utilize longer time slices, but this has not (yet) been a problem.

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

Copyright © 2004 John Wiley & Sons, Ltd.

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.

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 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.

The design of the signals is shown in Figure 1.

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 include 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 using the *initialize* routine with the parameter *iterations* set to zero. Optional per-timing interval initialization is also done before each timing interval using the *initialize* routine with

Copyright © 2004 John Wiley & Sons, Ltd.

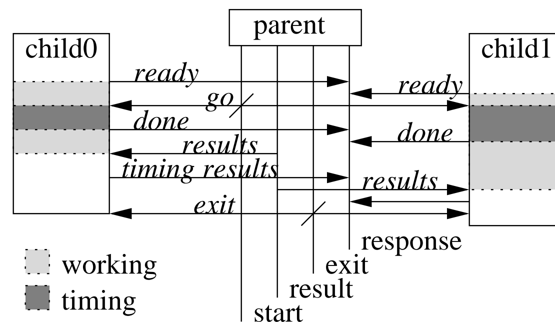

Figure 1. Control signals

iterations set to the number of iterations the benchmark routine is going to do during the next timing interval.

Similarly, each benchmark is allowed to specify a cleanup routine that is run by the child processes after each timing interval and 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.

Calling the initialization and cleanup routines before and after each timing interval is useful for some benchmarks that either consume or destroy system resources because it allows the creation and/or destruction of data structures and artifacts associated with timing interval. For example, *lat_fs* measures file creation time, so after each timing interval it must destroy all the files created during the timing interval. Similarly, when measuring file deletion time, it must create the right number of files before each timing interval. Since the number of iterations can vary, these sorts of initialization and cleanup actions cannot be done during global initialization because the timing harness doesn't necessarily know ahead of time how many files will need to be created or destroyed.

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.

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

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.

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 [17], an average of several results [2], or a trimmed mean [1].

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 [27]. 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 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 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 confidence intervals or the difference between the first and third quartiles. The most common method for computing confidence intervals relies on an assumption that the data samples are normally distributed, which is often emphatically not the case with computer benchmark results. Other methods for computing confidence intervals assume that one knows (or guesses) the underlying distribution. While it may be possible to estimate or approximate the underlying distribution for any given dataset, doing so automatically for all possible datasets is not feasible.

This leaves only the option of reporting confidence intervals or other statistics of variation using methods that are distribution-free, such as the *bias-corrected and accelerated* (BC_a) bootstrap method of Efron and Tibshirani [28]. Alternatively, one may use Gilat and Hill's [29] method for constructing confidence intervals on a quantile, such as the median. This is an enhancement under active consideration.

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

CORE MICRO-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 II 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 [25].

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 benchmark 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.

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.

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

Copyright © 2004 John Wiley & Sons, Ltd.

Table II. Benchmarks

Name	Measures
	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 each do *usecs*-worth of work
lat_proc	procedure call overhead and process creation using *fork*, *fork* and *execve*, and *fork* and *sh*
lat_rand	random number generator
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
	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
	Other
disk	zone bandwidths and seek times
line	cache line size
lmdd	*dd* clone
mhz	CPU clock speed
par_mem	memory subsystem ILP
par_ops	basic operation ILP
stream	STREAM clones [2]
tlb	TLB size

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

data to disk on one system, but only includes the time to flush a portion of data to disk on another system).

As determined 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, and then each process accesses its private file. When measuring accesses to a single file, each child simply uses the designated file directly.

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.

Unscalable benchmarks

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

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 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.

The benchmarks that measure aspects of memory-subsystem micro-architecture, *lat_dram_page*, *line*, *par_mem*, and *tlb*, were not parallelized because the multiple processes'

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

memory access patterns would likely interfere with one another. For example, in *lat_dram_page*, those accesses which were supposed to be to open DRAM pages could well be accessing closed DRAM pages, invalidating the benchmark.

lmdd was not parallelized because it is supposed to be a clone of *dd*, and it wasn't clear what a parallel form of *dd* would look like.

RESULTS

We have included some sample results for an IBM xSeries 370 eight-way 700 MHz Pentium III machine running Linux 2.4.18-3smp. This machine has an interesting architecture, in that the system is essentially split into two four-way machines with a cross-system bus. Each set of four processors has its own memory, giving the system a NUMA architecture, however the Linux kernel used in the experiments was not NUMA-aware.

Because our results were very noisy, we ran each benchmark thirty one (31) times, with each sample being the median of eleven (11) measurements taken during the benchmark.

This noise can be explained as arising from a number of sources: NUMA memory architecture, caching issues, and scheduler placement decisions. Optimal placement would spread all jobs across the processors, memory would be allocated from the "closest" memory bank, and jobs would not migrate once they started. This would maximize processor utilization, cache performance, and minimize memory latency and congestion. However, the 2.4 Linux kernel is not NUMA-aware, so it allocated memory in a fashion that was blind to the non-uniformity of memory access costs and memory subsystem congestion and contention issues. Consequently, on this machine, the scheduler is equally likely to assign the process to a processor in the other half of the system away from the memory being used by that process, which increases the result variability. In addition, as the number of processes increases, and the amount of traffic crossing between halves of the system increases, performance of the bridge linking the two halves together can become a bottleneck, further increasing variability because the cross-talk traffic is a function of the random job placement.

Since the Linux 2.4 schedulers did not include processor affinity, processes could migrate from processor to processor much too often. Each time the process migrated to a new processor, it started over with a cold cache, temporarily and dramatically reducing performance.

All these factors interact and contribute to the high variability in the measurements, particularly for memory-intensive benchmarks.

Figure 2 show the best median results for various process-related latencies, such as system call and process creation overhead. It is clear that many operations scale perfectly with increasing number of processes, up to the number of processors in the test machine, such as *null call, signal handler install, signal handling*, and *TCP select on 100 file descriptors*. Other operations demonstrate contention for resources by exhibiting increased latencies as more processes access the (shared) resources, such as *null I/O, open/close*, and *stat* with a shared file descriptor, or the process creation operations *fork, exec*, and *sh*.

Figure 3 show the best median results for various communication-related operations. From this graph we can see that remote procedure calls is more expensive than a simple packet "ping-pong" over both UDP and TCP transport layers. Additionally, communications over

Copyright © 2004 John Wiley & Sons, Ltd.

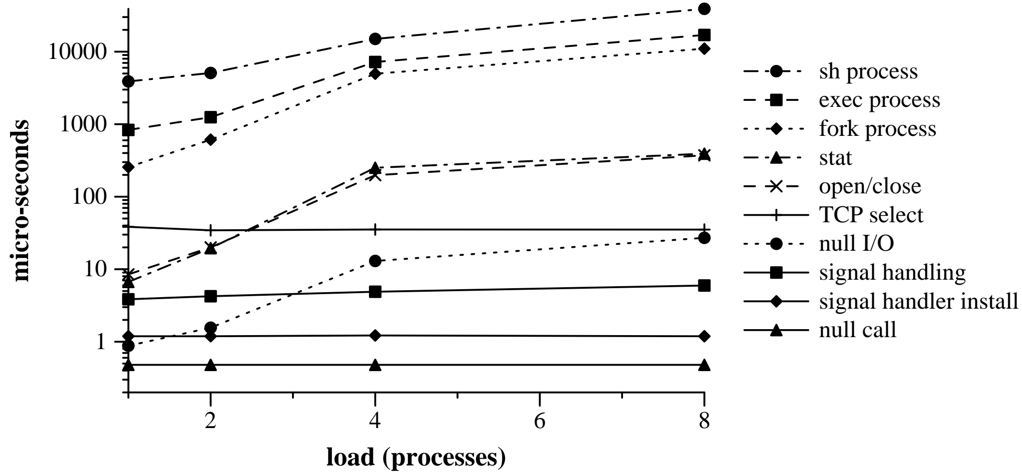

Figure 2. Process latencies

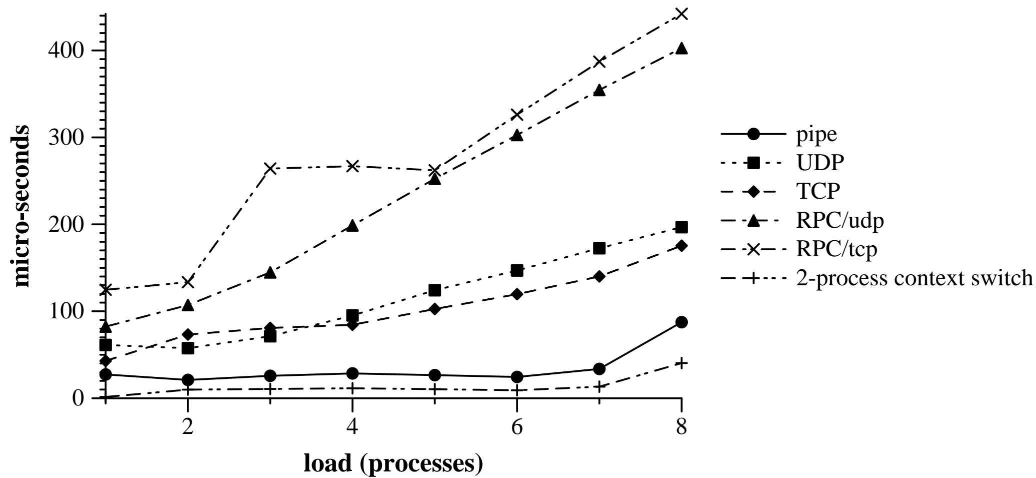

Figure 3. Communication latencies

Copyright © 2004 John Wiley & Sons, Ltd.

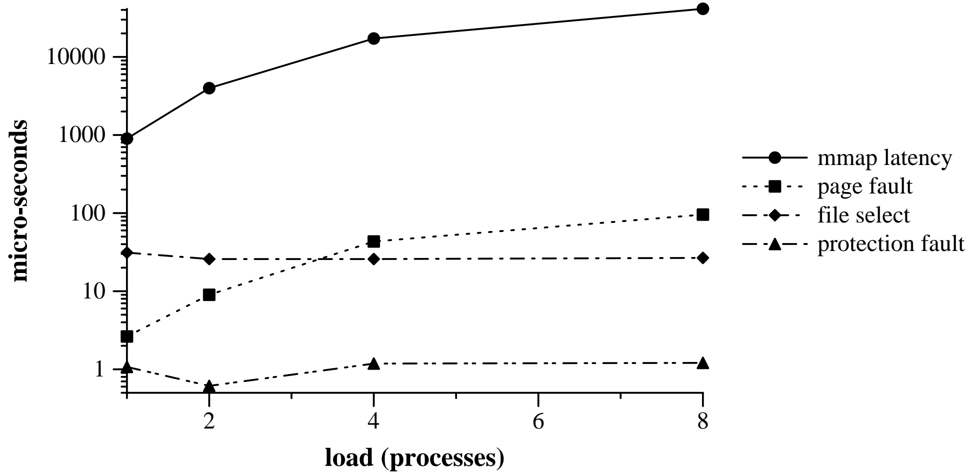

Figure 4. File latencies

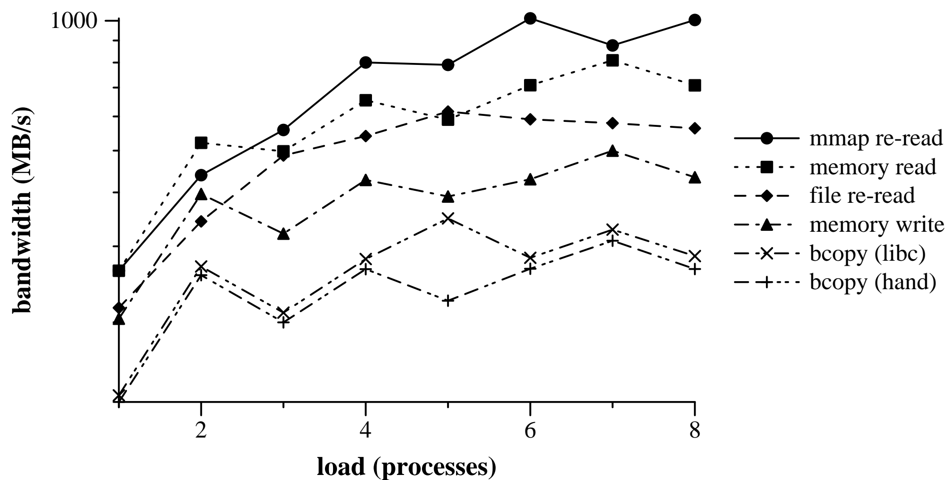

Figure 5. Memory bandwidths

pipes are substantially faster than either TCP or UDP, with costs only incrementally larger than context switching.

Figure 4 demonstrates that the Linux kernel scales poorly for operations that lock the file, such as *mmap* and *page fault*, while operations that do not require locking the file scale very well.

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

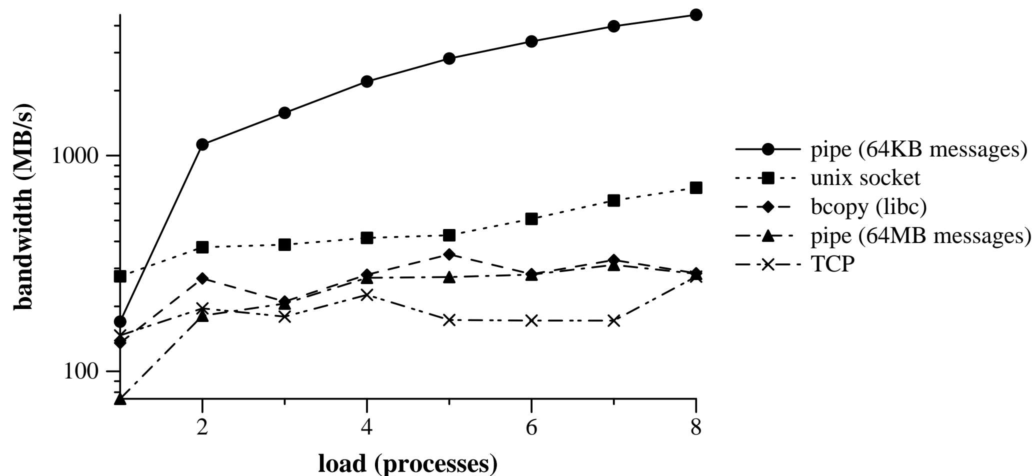

Figure 6. Communication bandwidths

Figure 5 shows the best median bandwidth for various memory operations versus load scaling when manipulating a 64MB memory buffer. Clearly, read operations are faster than write, which are faster than copy operations, and reading data from files via *mmap* is faster than using the *read* interface. Also, *mmap* file re-read appears to be faster than simply accessing memory, which is likely due to cache affects since each of the processes is accessing the same file and is therefore accessing the same pages.

Figure 6 shows the best median bandwidths for some inter-process communication methods when copying 64MB from one process to another. TCP, unix socket, and pipe (64KB messages) all pass data in 64KB chunks from one process to another. Unix socket and pipe (64KB messages) both have performance greater than simple bcopy, which is likely due to cache affects — the sending and receiving process are assigned to the same processor and all or part of the 64KB send buffer and 64KB receive buffer are cache resident. For contrast, pipe (64MB messages) performance is comparable to simple bcopy implying that pipe can (sometimes) be accomplished with as little as a single bcopy and that the improved performance of pipe (64KB messages) is due to cache affects.

CUSTOMIZING *lmbench*

In some ways the suite of micro-benchmarks shipped with *lmbench* can be thought of as an extensive and useful collection of sample benchmarks that use the timing harness. However, the real power of the system lies in the ability to rapidly and confidently create new custom benchmarks that measure specific features of interest to the user.

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

The primary constraint is that the benchmarked operation must be idempotent (i.e., repeated execution of the benchmark has the same effect as a single execution). The timing harness must be able to repeat the operation any number of times to ensure that the aggregate delay is large enough to provide sufficient timing accuracy. This can require careful consideration during benchmark construction. For example, a file deletion benchmark must be able to delete any requested number of files, or a floating point division benchmark should not divide by one (due to software and hardware optimizations) but repeated division ($f = fracfg$) by a number other than one can lead to underflow or overflow conditions yielding potentially misleading results.

One potential problem with micro-benchmarks is that they explicitly ignore potential interactions between operations or components [3]. Using the *lmbench* timing harness, it is easy to encapsulate and measure operations at various levels of complexity and granularity. This makes it much easier to evaluate the performance of the operations both in isolation and as part of a more complex system.

A simple benchmark to measure the latency of the getppid() system call would look like:

```
#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);
}
```

It is fairly easy to benchmark more complex aspects of system performance.

SLEEP benchmark

Someone wanted to use the high resolution sleep interfaces for an application which required a sleep call with micro-second accuracy, so they asked us to measure the accuracy of the various interfaces. The author was aware of at least four standard interfaces that could provide that resolution: usleep(), nanosleep(), select(), and itimer. Within a few minutes the developers had a new benchmark, *lat_usleep*, which measured the actual duration of sleep times for a given requested sleep time.

Figure 7 shows the results of running this benchmark with realtime priority under Linux 2.4.19 on a 1.8GHz Pentium 4 Xeon with various requested sleep durations. The results show how the nanosleep() interface provides reasonably accurate sleep durations for requests less

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

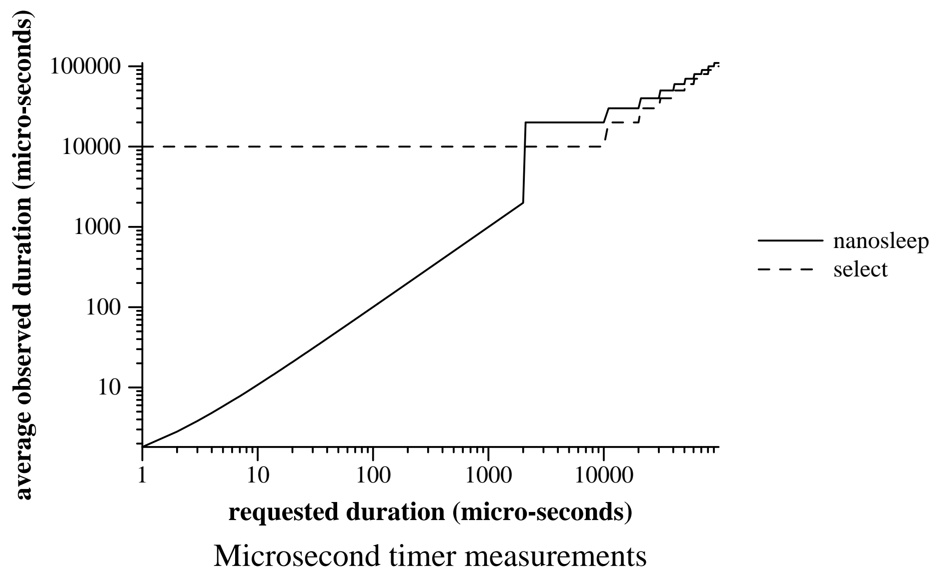

Figure 7. Sleep timer durations

than 2000μsec, but in all other cases the durations have very low resolution (roughly two scheduler clock ticks), with durations between 10,000 and 20,000μsec longer than requested. On the other hand, select() has poor performance across the board, but the durations are consistently between 0 and 10,000μsec too long. The results for usleep() and itimer() are not included on the graphs because they match the results for nanosleep() and select() respectively.

Unfortunately, based on these graphs we were able to determine that none of the interfaces provide a microsecond-level sleep capability sufficient for the purpose.

PMAKE-like benchmark

Another time, someone wanted to figure out how well some clustering software worked in a *pmake*-like situation. In this case the problem can be stated as measuring the time to completion when creating N jobs which each do μsecs-worth of work. Developing this benchmark took less than an hour.

Figure 8 shows the results of running this benchmark on a four machine cluster of dual processor HP x4000 workstations running openMosix Linux 2.4.19. Note that there are a total of eight processors, so with perfect scaling the 8-job latency would be identical to the 1-job latency.

There is little difference between the 1-job and 2-job curves, indicating the Linux load balancing is very effective. The 4-job and 8-job curves show that openMosix does no load

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

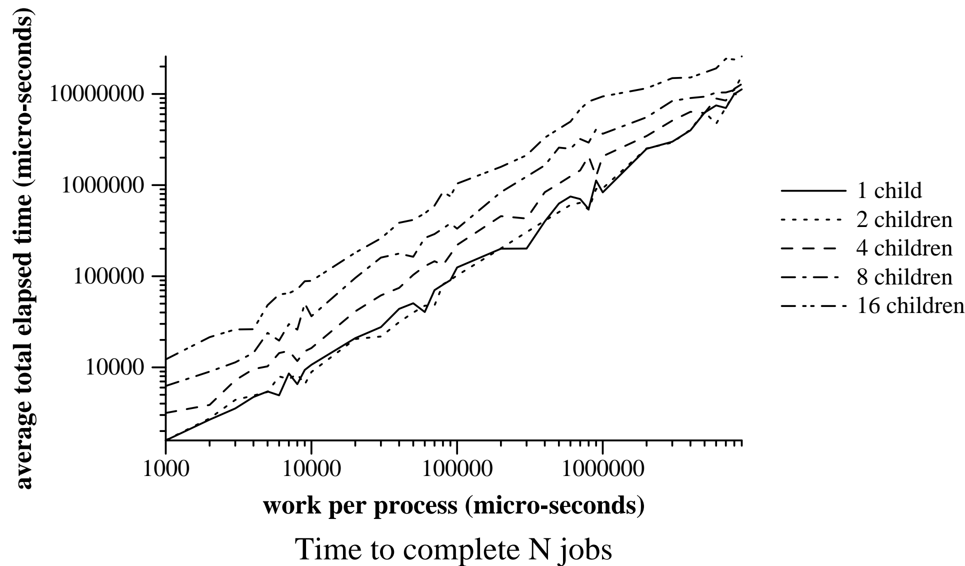

Figure 8. *pmake*-like workload

balancing for jobs that run less than a second, but starting at one second the load balancing beginning to take effect, and the durations drop to be similar to the 1- and 2-job curves. This shows the effectiveness of the openMosix dynamic load balancing system.

More sophisticated variants of this benchmark can be easily created to measure openMosix's ability to deal with mixtures of long- and short-running processes.

STREAM benchmarks

Another example of creating new benchmarks quickly and easily using the *lmbench3* timing harness is the new *stream* micro-benchmark which measures the performance of John McCalpin's STREAM benchmark kernels for both STREAM version 1 [2] and version 2 [30]. 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 III is based on McCalpin's tables [30] and shows the four kernels for version 2 of the *stream* benchmark. While the byte and FLOPs counts are largely architecture independent, since they are a function of the data size (eight byte doubles) and kernel, there are some architectural features which can affect both these counts.

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Table III. STREAM version 2 costs

Kernel	Code	Bytes		FLOPS
		read	write	
FILL	$a[i] = q$	0(+8)	8	0
COPY	$a[i] = b[i]$	8(+8)	8	0
DAXPY	$a[i] = a[i] + q \cdot b[i]$	16	8	1–2
SUM	$sum = sum + a[i]$	8	0	1

Cache lines are almost invariably bigger than a single double, and so when a write miss occurs the cache will typically read the line from memory and then modify the selected bytes. Sometimes vector instructions such as the Intel Streaming SIMD Extensions (SSE) and AMD 3DNow! instructions can avoid this load by writing an entire cache line at once. The parenthesized numbers in the *read* column represent the average number of bytes read into the cache as a result of the write to that variable. 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.

In addition, some architectures such as the PowerPC and Itanium families support fused floating point multiply-add instructions which can do both the multiply and add operations for TRIAD and DAXPY in a single operation [31], so the physical FLOPS count would be 1 for these architectures using these instructions, and 2 otherwise.

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, *add*, *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.

One difference between *lmbench*'s and McCalpin's STREAM implementations is the fact that the *lmbench* implementation reports the median result while the original implementation reports the best result. In *lmbench* we want to report numbers that are representative rather than the best possible results (which may be rarely seen).

Figure 9 shows the STREAM bandwidth figures for the version 2 kernels accessing 32MB of RAM on an IBM xSeries 370 eight-way 700 MHz Pentium III machine running Linux 2.6.8.1. At each point the benchmark was run one thousand and one (1,001) times, and the median value was reported, as the results contained a fair amount of variance. Clearly the memory subystem scales poorly, with results similar to those reported in Figure 5.

Figure 10 shows the best STREAM bandwidth obtained by a single process while the system is under the specified load for the STREAM version 2 *daxpy* kernel. This curve helps to shed

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

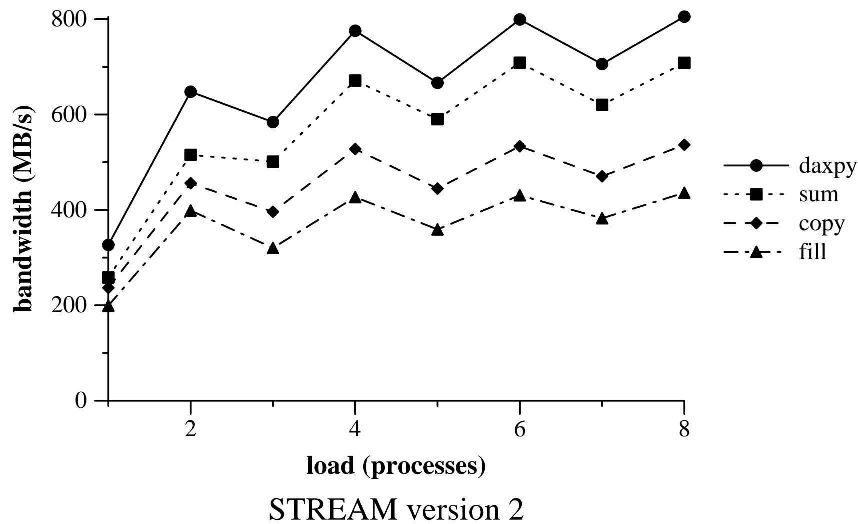

Figure 9. STREAM system bandwidths

some light on the situation, in that we can see that it is possible for performance of the fastest process to remain relatively flat until there are six processes running, at which point the best possible performance drops precipitously.

Figure 11 shows the histograms of the median performance. Interestingly, there is essentially no variation in the performance under loads one (1) and eight (8), and most of the other distributions are multi-modal with very narrow peaks.

Further investigation revealed that this system uses the Intel Profusion chipset [32, 33], which has a split-bus architecture. The chipset has two processor busses, each with four processors. The processor busses are connected via an L3 cache to a common processor bus which provides access to the various I/O devices and the system memory. The memory controller has five ports, so it can only service five memory requests at a time.

Given this information, we can look at Figures 10 and 11 for new insight. Figure 10 shows that so long as there are enough memory ports, it is possible for a single process to proceed at full speed. However, once processes start contending for memory port resources, performance starts degrading. Figure 11 also supports the theory that the two processor-bus architecture

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

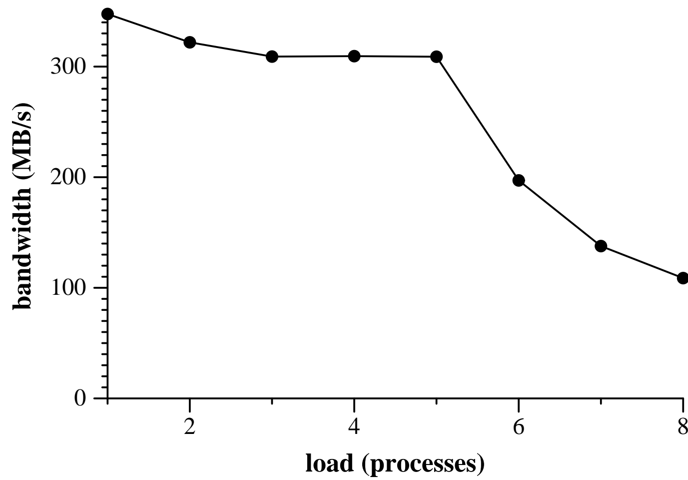

Figure 10. Maximal per-process STREAM version 2 *daxpy* bandwidths

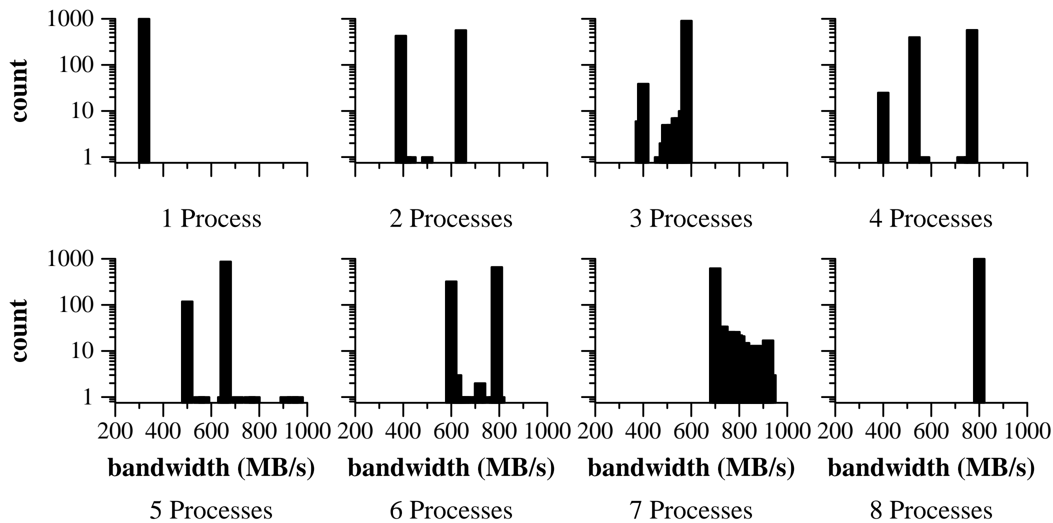

Figure 11. STREAM version 2 *daxpy* system bandwidth histograms

Copyright © 2004 John Wiley & Sons, Ltd.
Prepared using speauth.cls

impacts performance in interesting ways that should be addressed by the processor scheduling algorithm.

For example, performance of two (2) processes can vary by nearly 50% depending on how the processes are allocated to processors. If the processes are assigned to processors on difference processor busses, then processor bus and L3 cache contention is minimized and each process runs at full speed or roughly 650MB/s in aggregate. If the processes are assigned to processors on the same processor bus, then processor bus and L3 cache contention becomes an issue and system performance suffers, with roughly 400MB/s in aggregate. Note that the height of the peaks corresponds fairly closely with the probabilistic calculation given the assumption of uniform random assignment of jobs to processors: roughly $\frac{4}{7}$ for processes going to processors on separate busses, and roughly $\frac{3}{7}$ for processes going to processors on the same bus.

Similarly, the two peaks in performance of three (3) processes can be explained by the two cases: three processes on processors on a single bus, and a two-one split of processes to processors on separate busses. In the first case of three processes on a single bus, performance is no better than two processes on a single bus because the processor bus and L3 cache are already saturated. In the second case of two processes on one bus and one process on a second bus, we again see no overall system improvement over the two processes on separate busses case above. Again, the relative likelihood of the two cases can be explained assuming uniform random assignment of processes to processors, with the added assumption that the instances between the two peaks are explained by scheduler instability and re-assignment of processes during the benchmarking, so for part of the time all three processes were on a single bus, with roughly $\frac{3}{7} \cdot \frac{2}{7} = 0.12$ of the runs matching the first case, and the remainder matching the second case.

Using the observation that a single processor bus is saturated at about 400MB/s, one may predict that the whole system will be saturated at about 800MB/s. Looking at the eight processor results, one may see that this result is confirmed.

The remaining mystery are the results for the seven process case where some of the results are greater than the apparent system limit of 800MB/s. These could occur because *lmbench* uses the median measurement from all timing measurements on all child processes. If the scheduler left three processes on a single bus, and from the remaining four processes it kept revolving one of the processes to the bus with the three “static” processes, then the median result would come from one of the four processes which would have had some time in the less congested bus, resulting in an apparent overall improvement in system throughput.

The wide variation in system throughput as an apparent result of scheduler decisions indicate that this system could benefit from scheduler optimizations that took into account the physical architecture and tried to balance the load between processor busses.

CONCLUSIONS

In conclusion, lmbench is a powerful suite of micro-benchmarks built on top of a flexible timing harness. The set of benchmarks contained in the distribution provide a good summary of important aspects of system performance, and the timing harness makes it easy to rapidly construct benchmarks tailored for the customer’s problem.

Copyright © 2004 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Future extensions and enhancements to *lmbench* might include the addition of benchmarks to measure various thread-level operations and overheads, such as thread creation or synchronization. Similar benchmarks might be created to evaluate the various overheads and latencies associated with MPI and PVM distributed computing systems. We also hope to extend the suite of benchmarks that analyze various aspects of the micro-architecture, especially the memory hierarchy.

lmbench is available for anonymous ftp from *ftp://ftp.bitmover.com/lmbench/*

ACKNOWLEDGEMENTS

lmbench was originally developed by Larry McVoy, initially at Sun Microsystems and later at Silicon Graphics. Many people have provided assistance, guidance, feedback, and support during the development of this system. We would particularly like to thank Eric Anderson (HP), Bruce Chapman (SUN), Mani Fischer (HP), Larry McVoy (BitMover), Christine Moore (OSDL), David Mosberger (HP), Wayne Scott (BitMover), John Wilkes (HP), and Mitch Wright (HP) for their assistance during the long development process for *lmbench* version 3. Last, but certainly not least, we should like to thank the Open Source Development Lab for providing access to the eight-way Pentium III server used for our some of our performance analysis.

REFERENCES

1. 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*, Seattle, WA, June 1997, pp. 214–224, http://www.eecs.harvard.edu/~vino/perf/hbench/sigmetrics/hbench.html.
2. John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture newsletter*, Dec. 1995.
3. Jeffrey C. Mogul, "Brittle metrics in operating systems research," in *Proceedings 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, Mar. 1999, pp. 90–95.
4. R. Clint Whaley and Jack Dongarra, "Automatically tuned linear algebra software," UT-CS-97-366, Department of Computer Science, University of Tennessee, Knoxville, TN, 1997, http://math-atlas.sourceforge.net/.
5. Standard Performance Evaluation Corporation, "SPEC hpc96 benchmark," 1996, http://www.specbench.org/hpg/hpc96/.
6. R.P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.
7. Guarav Banga and Peter Druschel, "Measuring the capacity of a web server," in *Proceedings USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997, pp. 61–71.
8. 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, pp. 69–83.
9. Barry Shein, Mike Callahan, and Paul Woodbury, "NFSSTONE: A network file server performance benchmark," in *Proceedings USENIX Summer Conference*, Baltimore, MD, June 1989, pp. 269–275.
10. Uros Prestor, "Evaluating the memory performance of a ccnuma system," M.S. thesis, Department of Computer Science, University of Utah, May 2001.
11. R.H. Saavedra and A.J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, vol. 44, no. 10, pp. 1223–1235, Oct. 1995.
12. Cristina Hristea, Danial Lenoski, and John Keen, "Measuring memory hierarchy performance of cache-coherent multiprocessors using microbenchmarks," in *Proceedings of Supercomputing '97*, San Jose, CA, Nov. 1997, http://www.supercomp.org/sc97/proceedings/TECH/HRISTEA/.

Copyright © 2004 John Wiley & Sons, Ltd.

13. Rafael H. Saavedra-Barrera, *CPU Performance evaluation and execution time prediction using narrow spectrum benchmarking*, Ph.D. thesis, Department of Computer Science, University of California at Berkeley, 1992.
14. 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, pp. 102–107.
15. Arvin Park and J. C. Becker, "Iostone: a synthetic file system benchmark," *Computer Architecture News*, vol. 18, no. 2, pp. 45–52, June 1990.
16. Barry L. Wolman and Thomas M. Olson, "IOBENCH: a system independent IO benchmark," *Computer Architecture News*, vol. 17, no. 5, pp. 55–70, Sept. 1989.
17. 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*, vol. 6, no. 1, pp. 51–81, Feb. 1988.
18. Tim Bray, "Bonnie benchmark," 1990, http://www.textuality.com/bonnie/.
19. Peter M. Chen and David Patterson, "Storage performance — metrics and benchmarks," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1151–1165, Aug. 1993.
20. 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*, vol. 12, no. 4, pp. 308–339, Nov. 1994.
21. Keith A. Smith and Margo L. Seltzer, "File system aging — increasing the relevance of file system benchmarks," in *Proceedings of the 1997 SIGMETRICS Conference*, Seattle, WA, June 1997, pp. 203–213.
22. PARallel Kernels and BENCHmarks committee, "PARKBENCH," 2002, http://www.netlib.org/-parkbench/.
23. NASA Advanced Supercomputing Division, NASA Ames Research Center, "NAS parallel benchmarks," http://www.nas.nasa.gov/NAS/NPB.
24. Ian Glendinning, "GENESIS distributed memory benchmark suite," 1994, http://wotug.ukc.ac.uk/-parallel/performance/benchmarks/genesis.
25. Larry McVoy and Carl Staelin, "*lmbench*: portable tools for performance analysis," in *Proceedings USENIX Winter Conference*, San Diego, CA, Jan. 1996, pp. 279–284.
26. Carl Staelin and Larry McVoy, "*mhz*: anatomy of a microbenchmark," in *Proceedings USENIX Annual Technical Conference*, New Orleans, LA, June 1998, pp. 155–166.
27. Raj Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley-Interscience, New York, NY, Apr. 1991.
28. Bradley Efron and Robert J. Tibshirani, *An Introduction to the Bootstrap*, vol. 57 of *Monographs on Statistics and Applied Probability*, Chapman and Hall, New York, NY, 1993.
29. David Gilat and T. P. Hill, "Strongly-consistent, distribution-free confidence intervals for quartiles," *Statistics and Probability Letters*, vol. 29, pp. 45–53, 1996.
30. John D. McCalpin, "The stream2 home page," 2002, http://www.cs.virginia.edu/stream/stream2/.
31. "Fused multiply add," 2004.
32. Intel, "Profusion — an 8-way symmetric multiprocessing chipset," July 1999, http://netserver.hp.com/-docs/download.asp?file=tp_profusion(r).pdf.
33. Tao Zhou, "Profusion architecture," November 1999, http://www.minnetmag.com/Windows/Article/-ArticleID/7273/7273.html.

Copyright © 2004 John Wiley & Sons, Ltd.