mhz: *Anatomy of a micro-benchmark*

Carl Staelin
Hewlett-Packard Laboratories

Larry McVoy
BitMover, Inc.

Abstract

Mhz *is a portable ANSI/C program that determines the processor clock speed in a platform independent way. It measures the execution time of several different C expressions and finds the* ***greatest common divisor*** *to determine the duration of a single clock tick.*

Mhz *can be used by anyone who wants or needs to know the processor clock speed. In large installations it is often easier to experimentally determine the clock speed of a given machine than to keep track of each computer. For example, a platform-independent database system optimizer may use the clock speed while calculating the performance tradeoffs of various optimization techniques.*

To run the benchmark long enough for timing to be accurate, mhz *executes each expression in a loop. To minimize the loop overhead the expression is repeated a hundred times. Unfortunately, repetition enables many hardware and compiler optimizations that can have surprising effects on the experimental results. While writing* mhz, *much of the intellectual effort went into the design of expressions that minimize the opportunities for compiler and hardware optimization.*

Mhz *utilizes* lmbench 2.0*'s new timing harness, which manages the benchmarking process. The harness automatically adjusts the benchmark to minimize run time while preserving accuracy, determines the necessary timing duration to get accurate results from the system clock, and measures and accounts for both loop overhead and measurement overhead. It is used throughout* lmbench 2.0 *and can be used to measure the performance of other applications.*

1 Introduction

Mhz is a portable ANSI/C program that determines the processor clock speed in a platform independent fashion, which does not depend on any specific compiler, operating system, or processor. *Mhz* is part of the *lmbench* [1] suite of micro-benchmarks and was used to develop the new timing methodologies for *lmbench 2.0*. *Lmbench*'s guiding philosophy can be described as "accuracy, speed, portability, and simplicity." Each of these tenets impacted the design of *mhz*.

At first glance, determining the processor clock speed seems simple; time the execution of a short number of instructions and divide by the number of instructions. There are several problems with this simple approach, such as the lack of standard clocks with enough resolution to measure the duration of a few instructions accurately. In addition, *mhz* is written in portable ANSI/C that can be compiled into an unknown sequence of instructions of unknown length.

There are a variety of problems that need to be addressed in order to accurately measure time intervals on various processors under various operating systems. On processors with cycle times of 5 nano-seconds, some operating systems have low-resolution clocks, as poor as 10,000,000 nano-seconds, while others have 1,000 nano-second resolution clocks.

Lmbench 2.0 incorporates an entirely new timing harness which automatically controls the experimental system to provide accurate results on all platforms. For example, it determines how long experiments have to run in order for timing results to be accurate within 1% and then controls the experiments so they run just that long. It also automatically corrects for various overheads, such as loop overhead and timing measurement overhead. Considerable effort went into preserving accuracy while minimizing run time, which has paid off in shorter run times (with the same or better accuracy as *lmbench 1.0*) on systems with relatively fine grained clocks.

Determining the clock speed in a platform independent manner is surprisingly difficult because there is no way to measure one clock tick. The inspiration for the solution was based on hazy memories from high school chemistry and physics, of techniques used by nineteenth century chemists and physicists to determine the atomic weight of elements and the charge of an electron [2,3,4].

Section 2 describes *lmbench* 1.0's solution and its limitations, while Section 3 provides some background on computer architecture and how it affects *mhz*. Sections 4 and 5 introduce and describe the newer solution. The experimental methodology used by *lmbench* in general and *mhz* in particular are described in Sections 6 and 7, and the results are presented in Section 8. The Appendix contains a description of the *lmbench 2.0* benchmarking API and a brief tutorial on writing benchmarks using the *lmbench* API.

2 *lmbench 1.0*'s solution

Lmbench 1.0 includes a version of *mhz* that was accurate for a wide range of processors, but contained processor-specific code. It has a single loop, which runs for about a second. The clock speed is the (estimated) number of clock ticks divided by the elapsed time. The number of clock ticks is approximated by multiplying the number of loop iterations (say 10,000) by the number of expressions per loop (1000) and the number of clock ticks per expression. The number of clock ticks per expression is known for some processors and assumed for others.

```
main(int ac, char *av[])
{
    register int a = 1, N = 10000;
    double usecs, mhz;

    start();
    for (i = 0; i < N; ++i) {
        a>>=ac; // expression 1
        a>>=ac; // expression 2
        …
        a>>=ac; // expression 1000
    }
    usecs = stop();
    mhz = N * 1000 / (double)usecs;
}
```

Figure 1

Figure 1 contains pseudo-code for a simple *mhz* that works for many existing processors. It assumes that each shift operation takes a single clock tick. To determine the processor clock speed, just run the benchmark long enough (say 10,000 iterations), and then divide the number of clock ticks (10000 * 1000) by the duration. *Lmbench 1.0*'s *mhz* had processor-specific expressions selected at compile time based on the operating system.

Although the original approach worked on about 90% of the platforms tested, it has several limitations:

- The expressions are processor-dependent.
- The number of clock ticks per expression is not always known a-priori.
- The loop size is fixed and provides no guarantee that the timing interval is significant relative to the system clock resolution.
- The timing loop is only run once, so it is susceptible to errors caused by other independent activity on the processor.

An approach that would be accurate on all modern and anticipated architectures was needed.

2.1 Other approaches

The approach described above requires *mhz* to know the number of clock ticks per expression. This is infeasible since *mhz* is written in ANSI/C and intended to run on a wide variety of processors. We could not find expressions that require a fixed number of clock ticks on all processors. Clearly a method for determining the clock speed that doesn't require such information is needed.

Several techniques were investigated, such as measuring the execution time of two expressions, subtracting the two times, and hopefully getting the duration of a single clock tick. Other techniques include: creating loops with different ratios of two expressions (e.g., a++;a>>=1; and a++;a++;a>>=1 which are 1:1 and 2:1 respectively), and varying the number of times an expression is repeated within the loop. Some of the techniques, such as measuring the difference between two expressions, suffered from the same weakness as the solution in *lmbench 1.0*. Unfortunately, none of these approaches works. At best, most approaches could give the time to execute a single expression, which can already be measured.

3 Computer architecture

Modern computer architectures are complicated and highly optimized. Many of these optimizations are useful for general purpose programs, but can wreak havoc on our micro-benchmark. They make it nearly impossible to predict exactly what happens during execution.

3.1 Superscalar

Super-scalar processors have multiple computational units and can execute multiple operations in a single cycle. Super-scalar processors can also overlap the execution of adjacent instructions, which means the average number of clock ticks per instruction is non-integral [5].

For example, the expression `a+=b+a+a` might be compiled into:

```
ADD   r1,r1,r3  ; r3=a+a
ADD   r2,r1,r4  ; r4=a+b
ADD   r3,r4,r1  ; r1=(a+b) + (a+a)
```

A superscalar processor with two arithmetic units could execute three instructions in two clock cycles by executing the first two instructions in parallel. This would make the average number of cycles per instruction 0.66.

3.2 Instruction reorder buffer

Instruction reorder buffers provide limited workflow-like architecture capabilities to otherwise traditional processors [6,7,8]. The processor keeps track of inter-instruction dependencies and executes an instruction as soon as its data is available (data may be unavailable because it has not arrived from memory yet or because it is the result of an instruction that hasn't completed yet). Unlike dataflow processors, instruction reorder buffers have a bounded (and limited) size, so there is a sliding window of workflow-like capabilities.

Suppose there is a processor with two arithmetic units and one barrel-shifter and the following assembly code:

```
ADD   r1,r2,r3  ; r3=r1+r2
SHR   r3,1,r4   ; r4=r3>>1
ADD   r1,r5,r6  ; r6=r1+r5
```

During execution the CPU will execute the two ADD instructions in parallel because all the arguments are available, and then it will execute the SHR instruction as soon as the first ADD completes.

Instruction reorder buffers combined with super-scalar processors provide the system with a great deal of flexibility and many opportunities for overlapping computations. Unfortunately, that flexibility makes it difficult to craft C expressions that preclude parallel execution.

3.3 VLIW

At least one next generation processor will use a very-long-instruction-word (VLIW) architecture. Each VLIW instruction includes several independent sub-instructions that may execute in parallel [9]. The compiler optimizer technology for VLIW is complex because of this new parallelism.

The next section explains why we see no reason for *mhz* to work incorrectly on VLIW processors.

4 *mhz* solution

Mhz's computes the clock speed using the *greatest common divisor* (GCD) of the execution time for nine expressions, assuming that the execution time for each is an integral multiple of the time taken by a single clock tick. This technique makes no assumptions about the number of clock ticks for any single instruction or the number of instructions used to implement a given expression, except that it executes in an integral number of clock ticks.

To ensure that each expression executes in an integral number of clock ticks (on average), *mhz* uses tightly interlocked operations so processors cannot overlap the execution of the expressions.

Mhz can compute the CPU cycle time if the compiler generates at least two instruction sequences with relatively prime cycle counts. *Mhz* uses several different sequences to increase the chance that two sequences will have relatively prime cycle counts on any given architecture.

The *relatively prime* condition is necessary for the greatest common divisor method work. If all the cycle counts have a common factor (e.g. 2), then the apparent CPU speed will be reduced by that common factor. Also, if there is so much variability in the data that there is no apparent GCD, then *mhz* will return a result that is too large. The instruction sequences are chosen so that there are almost always two sequences with relatively prime lengths.

The processor's clock speed is the GCD of the execution times of the various instruction sequences. For example, suppose *mhz* is trying to compute the clock speed for a 120MHz processor, and there are two instruction sequences:

1.	SHR	(2 cycles)
2.	SHR;ADD	(3 cycles)

If the execution times are:

1.	SHR	11.1ns (2 cycles)
2.	SHR;ADD	16.6ns (3 cycles)

The GCD is 5.55ns and the calculated clock speed is indeed 120MHz. Aside from problems caused by experimental noise, this method should always work with instruction sequences that have relatively prime cycle counts.

Suppose the two instruction sequences have cycle counts that are not relatively prime:

1.	SHR	11.1ns (2 cycles)
2.	SHR;ADD;SUB	22.2ns (4 cycles)

```
double
gcd(double e[], int esize)
{
/* assumption: shortest expression has
 * no more than MAX_COUNT instructions */
#define MAX_COUNT      6
  int   i, j, size;
  double min_e, min_chi2, result, a, b, chi2;
  double *y, *x = (double *)
       malloc(esize*esize*sizeof(double));

  /* find the smallest value */
  min_e = double_min(e, vsize);

  /* {e[j]:j},{|e[j]-e[i]|:i,j,i!=j},{0,0} */
  construct_dataset(e, esize, &y, &size);

  for (i = 1; i < MAX_COUNT; ++i) {
    b = min_e / i; /* clock tick guess */
    for (j = 0; j < size; ++j)
      x[j] = floor(y[j] / b + 0.5);

    /* regression of the samples */
    regression(x, y, size, &a, &b, &chi2);

    if (i == 1 || i*i*chi2 < min_chi2) {
      result = b;
      min_chi2 = chi2;
    }
  }
  free(x);
  free(y);
  return result;
}
```

Figure 2

The GCD will be 11.1ns, and the clock speed will appear to be 60MHz, which is the true speed, 120MHz, divided by the common factor, 2.

Mhz uses nine expressions, which have been carefully designed to minimize this problem. Finding expressions that execute in an integral number of clock ticks on all processors is non-trivial and is addressed below.

4.1 Greatest common divisor

Finding the GCD of the expression execution times can be non-trivial. Since integer arithmetic does not apply to an array of real-valued observations, *mhz* can not do integer arithmetic to find the GCD. In addition, the observations contain noise, which can obscure the true GCD. *Mhz* can, however, compute the GCD by assuming that each C expression executes in an integral number of clock ticks.

Assuming a single clock tick is b nano-seconds, each experimental observation, e_j, can be converted into an integer number of clock ticks, c_j, where c_j = `floor`(e_j / b + 0.5). The set of points $\{c_j,e_j\}$ should be nearly linear, and the linear regression should have y-intercept 0 and slope b.

Mhz cannot directly calculate b, but it can make a series of educated guesses and choose the best guess. The guesses, b_i, are based on the fact that each experimental time is an integral multiple of b, and are created so $b_i \equiv \min(e_j) / i$. The least-mean-squares linear regression of $\{c_j,e_j\}$ gives a better estimate of b than the initial guess b_i because it is based on all the experimental observations.

The best b_i can be chosen using the chi-squared error of the least-mean-squares linear regression. When $b_i > b$, the chi-squared error will be large because some observations will have poor fits. When $b_i \approx b$ (within the usual experimental error), the chi-squared error will represent the experimental error, and will be far smaller than errors for $b_i > b$. When $b_i < b$ and b is a multiple of b_i, the chi-squared error will be equal to or smaller than the error of b_i because noisy observations may have a slightly improved fit.

Since multiples of the first best fit will have an equal or smaller chi-squared error measure, *mhz* chooses the first fit that significantly reduces the chi-squared error. Comparing the (current) minimum chi-squared error with an i^2 weighted chi-squared error favors previous minimum chi-squared errors and prevents *mhz* from choosing multiples of the correct result.

Figure 2 contains the routine `gcd()`, which computes the GCD. It finds the minimum execution time, `min_e`. `construct_dataset()` creates the dataset y_j, which includes all the experimental measurements e_j, and adds data points with the difference between each pair of observations. To ensure that the regression runs through the origin, it also adds the point (0,0). For each integral number of clock ticks, i, $i \in \{1, 2, \ldots, 6\}$, it computes b_i, the points $\{c_j,y_j\}$, and the least-mean-squares linear regression [10]. The linear regression gives the chi-squared error and $\boldsymbol{a}$ and $\boldsymbol{b}$ such that: $y = \boldsymbol{a} + \boldsymbol{b}x$. If the weighted chi-squared error is less than the minimum chi-squared error, `gcd()` discards the previous result and saves the current minimum.

5 Atomic expressions

Mhz needs simple C expressions that can be strung together without being optimized out of a loop by a smart compiler. The key is to prevent the processor from computing expressions in parallel or overlapping execution of adjacent expressions. Thus each C expression and sub-expression must depend on the result of the previous expression and it must have no sub-expressions that can begin execution before the completion of the previous expression. Otherwise the processor may utilize the inherent parallelism in the expression and overlap the execution of adjacent instantiations of an expression.

This dependency is critical to the design of the C expressions. For example, the expression `a+=a` satisfies the dependency criteria because the next instantiation of the expression cannot be evaluated until the current expression has completed. The expression `a+=b+c` is not completely dependent because the `b+c` sub-expression may be calculated in parallel with the previous instantiation's `a+(b+c)` sub-expression.

5.1 Compiler interactions

Designing the expressions that execute in an integral number of clock ticks (on average) with enough variety to ensure that there are two expressions with relatively prime cycle counts was difficult. The problems were increased by the compiler and processor optimizations and by compiler bugs and limitations.

We experimented with instruction sequences that use pointer accesses to cached memory locations and multi-variable integer arithmetic. Nearly all such expressions are optimized by modern processors that utilize super-scalar processing and instruction rescheduling to overlap execution of adjacent instances of the same expression.

Optimizing compilers gave us a number of headaches because they are able to optimize away many candidate expressions, if they are in simple loops. For example, the expression `a++;` was easily optimized. So we needed to find mathematical expressions that compiler writers either could not or have not bothered to optimize out of a loop.

Sometimes, the optimizer simply discarded the entire loop because the result was not used anywhere. Consequently, *lmbench* is sprinkled with calls to `use_result()`, a dummy procedure whose sole purpose is to fool compilers into thinking its argument is used somewhere else in the program.

Nearly all expressions using several integer variables were useless because they did not interlock correctly, i.e., advanced processors could overlap sub-expressions of the same expression or sub-expressions of adjacent expressions, and consequently, the average number of instructions per expression was non-integral.

There were a few arithmetic expressions that gave one or more compilers trouble (e.g. core dump, infinite loop, or erroneous output):

- a>>=a;
- a+=b+a;
- a+=b;b+=a;

One or more compilers optimized away the following C expressions:

- a+=a; // ADD optimized to a=0
- a&=a; // AND optimized away completely
- a^=a; // XOR optimized to a=0
- a+=b; // ADD optimized to a+=b+b+b+…
- a+=a;a-=a; // ADD;SUB optimized to a=0

The expression `a+=a` can be optimized to `a=0` because our loops contained one hundred copies of `a+=a` in a single iteration of the loop. Each instance of `a+=a` is equivalent to `a<<=1` for unsigned integers. Since C integers have 32 bits (or at most 64 bits), and since one hundred instances of `a+=a` is equivalent to `a<<=100`, the whole loop can be optimized to the single expression `a=0`.

The loop containing the expression `a&=a` can be optimized away because `a&=a` doesn't change the value of `a`. On the other hand, `a^=a` is equivalent to `a=0`, so the loop containing that expression can simply be replaced by the single expression `a=0`. Similarly the sequence `a+=a;a-=a;` is equivalent to `a=0` since `a-a=0`.

The expression `a+=b` provides a wide variety of possible optimizations when put in a loop with a hundred repetitions. One simple optimization is to set `a+=b+b+b+b+`… This allows superscalar hardware to execute multiple sub-expressions in parallel, which means that the number of clock cycles needed to compute the sum is not necessarily a multiple of 100. In addition, compilers may optimize the inner loop to `a+=100*b`.

5.2 mhz expressions

To maximize the possibility that cycle counts will be relatively prime, nine expressions were selected. For example, the expressions `a>>=b` and `a>>=a+a` differ by a single ADD operation, so on most machines their execution will differ by a single clock tick. There are similar small differences between many of the expressions.

The expressions are:

```
1. p=*p;
2. a^=a+a;
3. a^=a+a+a;
4. a>>=b;
5. a>>=a+a;
6. a^=a<<b;
7. a^=a+b;
8. a+=(a+b)&07;
9. a++;a^=1;a<<=1;
```

```
#define MHZ(M, expression)                      \
void                                            \
_mhz_##M (register long n,                      \
          register TYPE **p,                    \
          register TYPE a,                      \
          register TYPE b)                      \
{                                               \
    for (; n > 0; --n) {                        \
       HUNDRED(expression)                      \
    }                                           \
    use_pointer(p + a + b);                     \
}                                               \
                                                \
void                                            \
mhz_##M(int enough)                             \
{                                               \
    TYPE       i = 1;                           \
    long       n = 1;                           \
    TYPE      *x=(TYPE *)&x,                    \
    TYPE     **p=(TYPE **)x;                    \
    _mhz_##M(1, p, 1, 1);                       \
    BENCH1(_mhz_##M(n,p,i,i);n=1;, enough)      \
    save_n(100 * get_n());                      \
}
```

Figure 3

Figure 3 shows how the expressions are embedded in the timing harness. Each `MHZ()` macro creates both the function used to measure the execution time of a given expression and the corresponding simple function. Additional pieces of the harness, such as the experimental timing subsystem, are explained below.

Each expression is repeated 100 times in a loop embedded in a simple function (e.g., `_mhz_1()`). Another function (e.g., `mhz_1()`) uses the standard *lmbench* timing macro, `BENCH1()`, to measure the duration of each iteration of the loop in the corresponding simple function. The loop is embedded in a separate subroutine to increase the likelihood that the compilers would utilize register variables as intended.

Different processors can execute the expressions using different instructions and in varying number of clock ticks, but in general there are at least two expressions taking relatively prime number of clock ticks. Also, in each case, the various pieces of each expression are completely dependent and there are no two sub-expressions that can be executed in parallel. Adjacent instantiations of expressions are completely dependent so a processor cannot overlap execution.

6 *lmbench 2.0* timing harness

The single most important element of a good benchmark suite is the quality and reliability of its measurement system. *Lmbench 2.0* includes a timing harness that manages the experimental timing process to produce accurate results in the least possible time. *Lmbench 2.0* gets more accurate results in less time than *lmbench 1.0* by considering clock resolution, auto-sizing the duration of each benchmark, and conducting multiple experiments. Methods for measuring and eliminating several factors that influence the accuracy of timing measurements, such as the system clock resolution, are described below.

The timing harness includes two macros, `BENCH()` and `BENCH1()`, which provide a uniform method for conducting experiments. `BENCH1()` does one experiment and saves the result, while `BENCH()` does eleven experiments using `BENCH1()` and saves the median result. Benchmarked operations must be idempotent so they can be repeated indefinitely.

```
#include "bench.h"
int
main(int argc, char *argv[])
{
        BENCH(lrand48(), 0);
        micro("lrand48()", get_n());
        exit(0);
}
```

Figure 4

Figure 4 shows a complete example of a benchmark that measures the performance of `lrand48()` and reports its performance in micro-seconds. Please see the Appendix for a description of the *lmbench 2.0* benchmarking API and a brief tutorial on writing benchmarks using the API.

6.1 Clock resolution

Lmbench uses `gettimeofday()` to measure the time and compute the time intervals. Unfortunately, `gettimeofday()` has varying resolutions across different flavors of UNIX, and there is no standard method for querying the operating system to find the resolution of the system clock.

Lmbench includes a module, `compute_enough()`, that automatically computes the time interval required to reduce the timing error (due to clock resolution) to less than 1%. The module increases the timing interval until small variations in the measured work produce correspondingly small variations in the measured time. If a 100 milli-second interval is insufficient, the system uses 1second timing interval.

To verify that a timing interval is accurate to within 1%, it determines how many loop iterations consume the desired time, and then jiggles the number of iterations by 0.5% to time the duration of 100.0%, 100.5%, 101.0%, and 101.5% iterations. If the times are 100.0±0.1%, 100.5±0.1%, 101.0±0.1%, and 101.5±0.1%, the timing interval is presumed to be accurate to within 1%.

6.2 Timing overhead

Once the timing interval "enough" has been computed, the overhead of the timing measurements must be measured. The overhead is significant only on systems where the timing interval is relatively short.

The timing overhead is measured by benchmarking `gettimeofday()`. In *lmbench* the timing overhead is the time to exit `gettimeofday()` at the start of the timing interval plus the time to enter `gettimeofday()` at the end of the timing interval, so the time to call `gettimeofday()` represents the timing overhead.

6.3 Loop overhead

Sometimes, the overhead associated with the `for()` loop can be significant compared to the duration of the benchmarked feature, so the loop overhead needs to be measured and subtracted from the execution time. As far as possible, all micro-benchmarks in *lmbench 2.0* have been designed to minimize the impact of loop overhead on experimental results. Micro-benchmarks measuring fast operations have multiple instances of the operation in the loop to reduce the relative magnitude of the loop overhead.

To compute the loop overhead, *lmbench* uses two loops, the first with one instance of an expression and the second with two instances of the expression, giving two equations:

$$T_1 = N_1(loop\_overhead + work)$$
$$T_2 = N_2(loop\_overhead + 2\,work)$$

Where T_1, T_2 are the measured execution times and N_1, N_2 are the loop iteration counts. These equations can be solved for the loop overhead:

$$loop\_overhead = \frac{2T_1}{N_1} - \frac{T_2}{N_2}$$

6.4 Loop auto-sizing

Lmbench 1.0 uses fixed-size loops for many of the benchmarks. The loop sizes were hand-selected to run for about a second on contemporary processors. With processor speeds doubling every eighteen months, *lmbench* needs loops that can automatically scale themselves so the benchmark's accuracy is not compromised by faster processors.

All timing intervals must have the necessary accuracy, but the system does not know a-priori how many iterations are needed to run for the desired time. The experiments are repeated until the experiment runs for at least 95% of the desired time interval. `BENCH1()` adjusts the iteration count after each timing interval. If the measured time is less than 150 microseconds, then the iteration count is multiplied by 10, otherwise the iteration count is scaled by 1.1 times the ratio of the desired time to the measured time.

Some systems with low-resolution clocks return small integral values for intervals smaller than the clock resolution. *Lmbench* assumes that all timing results smaller than 150 microseconds are meaningless and multiplies the iteration count by 10. Otherwise *lmbench* can use the timing information to compute the iteration count needed for the timing interval to be long enough. Since the timing information has experimental noise, *lmbench* sets the iteration count a little larger than necessary.

6.5 Multiple experiments

Lmbench 1.0 reports the results for only one timing interval. As a result, *lmbench 1.0* is vulnerable to independent activity that steals processor time from the benchmark. In practice, the timing intervals are so big that the impact on the results was minimal, unless there is substantial activity. However, *lmbench 2.0*'s shorter timing intervals enabled by the loop auto-sizing and clock resolution detection mean that relatively little independent activity could have a significant impact on a single experiment.

Lmbench 2.0 performs multiple experiments and reports the median result. In general, the median is more robust and stable in the face of noise than the average result [10,11,12].

7 Making *mhz* really work

Mhz has different requirements and sensitivities than the rest of *lmbench*. *Mhz* is more sensitive to small errors in any given experiment than any of the other benchmarks in *lmbench*. As a result, *mhz* includes a variety of techniques to detect or minimize the impact of noisy data on its accuracy. *Mhz* needs the ability to detect when the data is too noisy to generate an accurate result and to detect obviously erroneous data. *Mhz* also needs to be insensitive to single experimental results that are inaccurate.

Since *mhz* measures the clock speed, and since most experimental errors increase the measured time, *mhz* uses the minimum experimental result for each expression, rather than the more standard median.

Mhz determines the experimental results are too noisy to provide a reliable answer by calculating the MHz twice, once using the minimum values for each expression, and once using the next larger values. If the difference between the two results is less than 1% or 1MHz, then the data is accepted. Otherwise, *mhz* assumes the results are invalid and retries the experiments, or on the third failure, it tells the user the system is too busy.

To reduce the impact of bursts of independent activity on the experimental results, *mhz* does not use the standard `BENCH()` macro. `BENCH()` takes all the measurements for a single expression, so a burst of activity might affect all the timing intervals for a single expression. To spread the experimental error over the all the expressions and maximize the chance of getting some valid results for each expression, the data collection is done in the `main()` procedure in a pair of nested loops. The inner loop iterates over the expressions and the outer loop iterates over the measurements.

`filter_data()` discards results that are obviously outliers. These are usually caused by optimizations that allow the system to optimize a long loop into a few instructions, which makes the number of clock ticks per expression approach zero. Since the C expressions used by *mhz* require a few instructions each, all the experimental results should be within a few multiples of each other. Results further from the median result can therefore be ignored.

The GCD is sensitive to even one noisy value. In order to reduce the impact of any single value, *mhz* computes the GCD for all valid subsets of the data points and chooses the mode (most common value) of the GCDs. Valid subsets have at least two *independent* data points. Data points are *independent* if the execution differs by one or more clock ticks.

Unfortunately, not all data points are *independent*. Some basic C expressions take the same number of clock ticks, but have slightly different experimental times due to noise. The GCD for a set of non-*independent* points will not be a single clock tick. `classes()` ensures that at least two data points appear to have different numbers of clock ticks. Heuristically, they are considered different if the values differ by more than 5%. The subset is ignored if no two points in a subset differ by more than 5%.

8 Results

Mhz has been tested on a wide range of processors, including: PA-RISC (PA-7000, PA-7200, PA-8000), Intel (486, Pentium, PentiumPro, Pentium II), DEC Alpha, PowerPC (PPC-603, PPC-604, PPC-604e), AMD (K5, K6), Sun (MicroSPARC, SuperSPARC, Ultra-I, Ultra-II), MIPS (R4000, R5000, R10000), Cyrix, Cray T3E, and Motorola 68020. It has also been tested on a wide variety of operating systems, including: HP-UX, IRIX, Linux, SunOS, AIX, BeOS, MkLinux, MachTen, OSF1, Unicos/mk, FreeBSD, and Plan9.

We released an alpha version of *mhz* to comp.arch and comp.benchmarks and a cast of volunteers, and received the results for 643 runs of *mhz*. The output of this alpha version of *mhz* includes all the data gathered by *mhz*. Out of 643 runs, 624 runs contained data that would have been accepted by *mhz* as valid. *Mhz* calculated the processor speed within 5% in 611 of 624 runs. *Mhz* had an error greater than 5% in 13 runs. Of those 13 runs, 10 were from one machine that had another CPU-bound process consuming 50% of the processor time, and *mhz*'s result was 50% of the clock speed.

Of the remaining 3 experiments, one was on a Cray T3E running Unicos/mk, and two were on a Sun UltraSPARC II running SunOS 5.5.1. On the Cray, *mhz* reported a clock speed of 633MHz instead of 600MHz. The version of *mhz* used in the experiments included the loop overhead, and the loop overhead measured in this experiment was far too large, artificially depressing the observed times, and inflating the apparent clock speed. We fixed the bug in the loop overhead calculation that caused the problem. On the Sun, the measured times are longer than expected, and the calculated processor speed is lower than expected. We suspect that there were other processes running on the system.

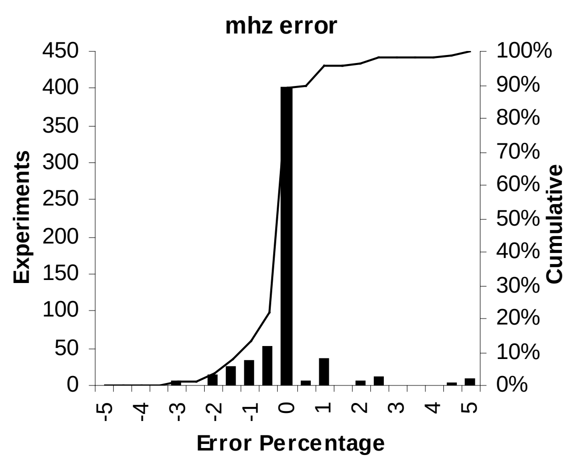

Figure 5

A histogram of *mhz*'s error distribution for the 611 runs is shown in Figure 5. Each bucket represents

0.5% error. *Mhz* is accurate, getting results ±1% 82% of the time, and results ±2% 93% of the time.

Figure 6 includes a selection of results for various processors and operating systems. Please note that the descriptive information is based on information provided by volunteers and may not always be complete.

Machine	Operating System	MHz
Motorola 68040	4.4BSD-Lite	25
Intel i386	Linux 2.0.33	33
Intel i486	FreeBSD 2.2.2	33
Sun 4m	SunOS 4.1.4	36
DEC R3000	ULTRIX 4.3	40
Sun SPARCstation-10	SunOS 5.5.1	40
Sun SPARCstation-20	SunOS 5.5	50
IBM POWER2	AIX	59
Sun Superserver-6400	SunOS	60
HP 730	HP-UX 10.20	66
Sun SPARCstation-4	SunOS	70
HP 715	HP-UX 9.05	80
Sun SPARCstation-5	SunOS	85
Intel Pentium	Linux 1.2.13	90
Sun SPARCstation-20	SunOS 5.5.1	90
Apple PowerMac 603e	Machten	100
HP 715/100	HP-UX 9.07	100
HP 725/100	HP-UX 9.07	100
Intel Pentium	Plan9	100
Intel Pentium	Linux 2.0.0	100
SGI R4000 IP17	IRIX 5.3	100
Sun SPARCstation-5	SunOS 5.5	110
Cyrix 6x86-P150+	Linux 2.0.33	120
DEC R4400	ULTRIX 4.4	120
HP 770	HP-UX 10.20	120
Sun SPARCstation-20	SunOS 5.5.1	125
Cyrix	Linux 2.1.60	133
DEC AlphaAXP	OSF1 3.2	133
Sun Ultra-1	SunOS 5.6	143
DEC Alpha 347	OSF1 3.0	144
DEC AlphaAXP	OSF1 3.2	150
Intel Pentium	Linux 2.0.0	166
Sun SPARCstation-20	SunOS 5.5.1	166
Sun Ultra-1	SunOS 5.5.1	167
Sun Ultra-2	SunOS 5.6	167
DEC AlphaAXP	OSF1 3.2	175
BeMac PowerPC 604e	BeOS	180
HP 780	HP-UX 10.20	180
SGI R5000 IP32	IRIX 6.3	180
DEC AlphaAXP	OSF1 3.2	190
DEC AlphaAXP	OSF1 3.2	200
HP 899	HP-UX 10.20	200
Intel PentiumPro	SunOS 5.5.1	200
Intel PentiumPro	Linux 2.0.32	200
MIPS 10000	ReliantUNIX-Y	200
SGI R4000 IP22	IRIX 6.2	200
Sun Ultra-1	SunOS 5.6	200
Sun Ultra-2	SunOS 5.5.1	200
IBM PowerPC604e	AIX	233
Sun Ultra-Enterprise	SunOS 5.5.1	248
Sun Ultra-2	SunOS 5.6	296
Cray T3E	Unicos/mk 2.0.2.12	300
IBM PowerPC 604e	AIX	332
Cray T3E-1200	Unicos/mk 2.0.2.12	600
Dec Alpha	Linux 2.0.30	600

Figure 6

9 Conclusions

Mhz is a portable C program that can quickly and accurately determine the clock speed of the host processor. *Mhz* demonstrates the utility of a simple mathematical principle: relative primality. *Mhz* also demonstrates many of the experimental and timing features found in *lmbench 2.0*.

Lmbench, including *mhz*, can be downloaded from:

http://www.kernel.org/pub/software/benchmark/lmbench

Lmbench is intended as a toolkit for application and systems programmers to analyze application and system behavior.

10 Acknowledgements

This work would not have been possible without the support, encouragement, and contributions of the many users of *lmbench*. We thank the many people who ran alpha versions of *mhz* and gave us invaluable feedback on its performance and accuracy on a wide range of processors and operating systems. The current program only works because of their efforts.

We would also like to thank Mary Solomon for her research into the experimental and analytic methods of the nineteenth century chemists and physicists, and Prof. Phyllis Brauner and Prof. Len Soltzberg who cheerfully shared their knowledge of nineteenth century chemistry.

Finally, we would like to thank the anonymous referees, Sigal Ar, Fred Douglis, Darryl Grieg, William Long, and Udi Manber for reviewing drafts of the paper, and Patricia Markee for her extensive editorial assistance.

11 Bibliography

[1] Larry McVoy and Carl Staelin, *lmbench: Portable tools for performance analysis*. USENIX technical conference. San Diego, CA. January 1996. pp. 279-284.

[2] R. A. Millikan, *The Isolation of an Ion, a Precision Measurement of its Charge, and the Correction of Stokes's Law*. The Physical Review XXXII(4). April 1911.

[3] Frederick Soddy, *The Interpretation of the Atom*. G. P. Putnam's Sons, New York, New York. 1932.

[4] Arthur Schuster, *The Progress of Physics during 33 years (1875-1908)*. Cambridge University Press, Cambridge, United Kingdom. 1911.

[5] David A. Patterson, John L. Hennesy and David Goldberg, *Computer Architecture: A Quantitative Approach*. Second Edition. Morgan Kaufman. 1996. pp. 221-354.

[6] Anne P. Scott, Kevin P. Burkhart, Ashok Kumar, Richard M. Blumberg, and Gregory L. Ranson, *Four-Way Superscalar PA-RISC Processors*. Hewlett-Packard Journal 48(4). August 1997.

[7] PowerPC 604 Risc Microprocessor, *http://www.chips.ibm.com/products/ppc/DataSheets/604/604-180.html*. June 1997.

[8] PentiumPro processor dynamic execution, *http://pentium.intel.com/procs/ppro/info/dynexec.htm*. April 1997.

[9] B. Ramakrishna Rau and Joseph H. A. Fisher, *Instruction-Level Parallel Processing: History, Overview, and Perspective*, Journal of Supercomputing 7(1/2). 1993.

[10] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Second Edition. Cambridge University Press, Cambridge, United Kingdom. 1992. pp. 656-706.

[11] Raj Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, New York. 1991. pp. 183-186.

[12] Russell Langley, Practical Statistics Simply Explained. Dover, New York, New York. 1970. pp. 51-88.

Appendix

The *lmbench 2.0* benchmark API is specified, with descriptions of the timing and reporting functions. In addition, a brief tutorial on constructing benchmarks using *lmbench 2.0* is included. Please see the *lmbench 2.0* release for the complete sources and documentation.

lmbench API

`BENCH(loop_body, enough);`

This macro is the standard interface to *lmbench 2.0*'s timing subsystem. It repeats the experiment `TRIES` times and reports the median value, unless enough is larger than 100milli-seconds. It uses the macro `BENCH1()` to run each experiment. If `enough` is non-zero, the experiment must run for at least `enough` micro-seconds.

`BENCH1(loop_body, enough);`

`BENCH1()` is the heart of the benchmarking system. It automatically calculates `enough` and actually benchmarks `loop_body`. It ensures that each experiment is run long enough that the timing errors are minimized. Environment variables `ENOUGH`, `TIMING_O`, and `LOOP_O` affect the initialization of the timing parameters.

`uint64 get_n();`

Returns the number of times `loop_body` was executed during the timing interval.

`void milli(char *s, uint64 n);`

Print out the time per operation in milliseconds. `n` is passed as a parameter because each `loop_body` could contain several instantiations of the operation, and there has to be a way to adjust the parameter.

`void micro(char *s, uint64 n);`

Print out the time per operation in microseconds.

`void nano(char *s, uint64 n);`

Print out the time per operation in nanoseconds.

`void mb(uint64 bytes);`

Print out the bandwidth in megabytes per second.

`void kb(uint64 bytes);`

Print out the bandwidth in kilobytes per second.

Tutorial

Creating benchmarks using the *lmbench 2.0* timing harness is easy. There are two attributes that are most critical for performance, latency and bandwidth, and *lmbench 2.0*'s timing harness makes it easy to measure and report results for both.

The timing harness can be used to quickly create accurate performance measurements for a wide range of features and applications. For example, if a signal processing application needs a fast FFT routine, the programmer could take several implementations, quickly benchmark them, and choose the fastest. Alternatively, the program could include a library of FFT routines and automatically choose the fastest based on the routine's performance on the particular hardware.

There are a number of factors to consider when building benchmarks. The most important thing to understand is what, exactly, you are trying to measure. If you are trying to find out how long it takes to generate a pseudo-random number, multiply two 500x500 matrices, or copy 1MByte, then *lmbench* can help you accurately measure and report that information quickly. You should also understand the conditions under which you would like to measure the performance. For example, if you want to know how long it takes to copy 4KBytes, then you should understand whether you want to find out how long it takes to copy 4KBytes from and to the cache, or from memory to the cache.

It is useful to form a hypothesis about the feature being measured. Using previously gathered information, it may be possible to accurately predict the performance. Then, build the benchmark, measure the performance, and test the hypothesis. If the hypothesis is wrong, you will have learned something new.

Measuring latency

Latency is usually important for frequently executed short operations, such as memory accesses. Since it is so easy to measure latency using *lmbench 2.0*, it becomes possible to quickly answer questions that arise during system design. For example, simulators may use random numbers frequently, so random number generator performance may be important to overall simulator performance. It takes a few minutes and a few lines of code to measure the performance of a random number generator:

```
1       #include “bench.h”
2       int
3       main(int argc, char *argv[])
4       {
5               putenv(“LOOP_O=0.0”);
6               BENCH(lrand48(), 0);
7               micro(“lrand48()”, get_n());
8               exit(0);
9       }
```

Line 1 includes the *lmbench* header, which contains the macros, type definitions, and function declarations for *lmbench*. Line 5 sets the environment variable LOOP_O to 0.0 so *lmbench* won't waste time calculating the negligible loop overhead. Line 6 uses the `BENCH()` macro to benchmark the `lrand48()` function. Since the `BENCH()` parameter `enough` is 0, *lmbench* will automatically calculate the necessary timing duration. Line 7 uses `micro()` to report `lrand48()`'s performance in micro-seconds.

Measuring bandwidth

The other major component of system performance is bandwidth, which is of primary importance while moving large chunks of data. The mechanics of measuring bandwidth are very similar to those for measuring latency. In many cases the only difference is the function used to report the results.

For example, `bcopy()` is the traditional C-library routine for copying data. It is often heavily optimized because it can measurably affect overall system performance in some commercial benchmarks. A simple benchmark to measure `bcopy()` performance might look like:

```
1       #include “bench.h”
2       #define M       (1024*1024)
3       int
4       main(int argc, char *argv[])
5       {
6               char    *a = malloc(M);
7               char    *b = malloc(M);
8               putenv(“LOOP_O=0.0”);
9               BENCH(bcopy(a,b,M), 0);
10              mb(M * get_n());
11              exit(0);
12      }
```

Lines 6 and 7 allocate two 1MByte chunks of memory. Line 9 benchmarks `bcopy()`'s performance while copying 1MByte of data from chunk `a` to chunk `b`. Line 10 reports the bandwidth in megabytes per second; during the benchmark timing interval `bcopy()` copied `M*get_n()` bytes.

There are some problems with this particular benchmark because of caching effects when the memory cache is large. Since the `bcopy()` will be repeated several times during benchmarking, the data is more likely to be in the cache, so the benchmark will measure `bcopy()` performance for cached data. If one is trying to measure `bcopy()` performance for non-cached data, this benchmark would need to be modified. For example, allocating larger segments of memory (e.g. 16MBytes) and only copying 1MByte at a time would increase the likelihood of measuring cold-cache results. The modified benchmark would look like:

```
1       #include “bench.h”
2       #define N       16
3       #define M       (1024*1024)
4       int
5       main(int argc, char *argv[])
6       {
7               int     o = 0;
8               char    *a = malloc(M * N);
9               char    *b = malloc(M * N);
10              putenv(“LOOP_O=0.0”);
11              BENCH(bcopy(a+o,b+o);
12                      o=(o+M)%(N*M);,0);
13              mb(M * get_n());
14              exit(0);
15      }
```

Line 2 defines a new constant, N, which is the number of unique segments the benchmark will use. Line 7 defines the offset variable, o, which is used to select the appropriate segment. Lines 8 and 9 allocate the memory for the segment. The `bcopy()` in line 11 copies data from one segment of `a` to a segment of `b`. The offset is updated in line 12 to point to the next segment, modulo the number of segments.

Statistics

It is important to understand the phenomena being measured. Some features, such as clock speed, are constant and variance in the results is caused by experimental noise. Other features, such as context switch times or disk I/O, have intrinsic variance.

The minimum result may be used in place of the median result in some circumstances. For example, when measuring memory latency as a function of actively used memory size, the median result would usually be used because of the variance added by caching effects. However, when determining the cache size, the minimum result might be used.

Special care must be taken when subtracting two measurements, which is usually done when subtracting overhead from an operation. The error in the subtracted result will be larger than the error in the joint measurement since the error in the overhead measurement must be added to the joint error. Also, the subtracted result is smaller so the percentage error has increased. If the overhead is a reasonable fraction of the total measurement, then the error in the result can be significant.

Common pitfalls

Sometimes benchmarks measure effects other than the intended result. There are many ways to make subtle, or even egregious, mistakes in benchmark design. Two common mistakes are measuring partial operations or measuring the wrong operation. Also, think about other factors that can affect the results, such as: caching effects, physical page placement and its effect on direct-mapped caches, process scheduling and cache-stealing in multiprocessors.

Measuring partial operations

Measure the whole operation, not just part of it. If you don't measure the whole operation you can sometimes miss significant features. For example, it is generally accepted that `mmap` is faster than `read` to read a file. If you measure the time to read a file without including the `open`, `close`, and `mmap` overhead, then `mmap` will usually appear to be faster than `read`. However, setting up the mapping is not free and some operating systems, such as Sun's (the origin of `mmap`), have optimized `read` so it is sometimes faster than `mmap`. Unless `mmap` is included in the benchmark, its overhead will not be included in the total cost. When comparing the sequence `open/mmap/bcopy/close` with `open/read/close`, `read` is faster than `mmap` for small files, typically under 32K. In addition, not all operating systems provide read-ahead for `mmap`'ed files, which can result in a 2-3x performance penalty for `mmap` during large sequential read accesses.

Measuring the wrong operation

Measuring the operation you intend to measure. Caches and caching effects are the usual source of problems. It is very important to decide whether you want to measure warm-cache or cold-cache performance. In general it is appropriate and easier to measure warm cache performance. There are other ways to measure the wrong operation, such as allowing overhead to dominate the measurement. For example, *mhz* demonstrates *lmbench*'s ability to measure very short operations, but we limited the loop overhead to a few percent by repeating each operation many times within the measurement loop. Otherwise the loop overhead could have been the dominant feature in the measurement loop.

Multi-process and networking benchmarks are especially prone to errors. Obscure aspects of system design can profoundly impact the benchmark. For example, the behavior of common TCP implementations during connection establishment limited the rate clients generated requests and completely invalidated some common HTTP benchmarks because the HTTP clients only generated requests as fast as the server could service them. The benchmarks never measured server performance during server overload, when the server spends all of its time acknowledging TCP connections.

For more information

There are a variety of sources of information on statistics, benchmarking, and system behavior. [5,10,11,12] are a good starting point for information. The *lmbench* documentation and man pages can help get you started. Also, the *lmbench* micro-benchmarks are a rich set of examples to use when writing your own benchmarks. We hope that *lmbench* will become a standard element of programmers' toolboxes.