Reliable Benchmarking with Limbo on Inferno

John Bates
Vita Nuova Limited
The Innovation Centre
University Road
York
England YO10 5DG

3rd March 1999
Revised 20th February 2000

ABSTRACT

Our goal was to identify a mechanism for performing accurate, reliable and realistic benchmarking measurements with Limbo on Inferno. The measurements would need to be *accurate* in the sense that we could be confident that they were sufficiently close to the real time taken to perform the measured operation. We would consider a measurement to be *reliable* if it could be obtained repeatedly when carried out under the same conditions at a later date. Additionally, a measurement is only useful if it can be claimed to be related to the expected performance of a real applicable operation; we would call such a measurement *realistic*. The emphasis in this document is on achieving a reliable measurement.

Expectation

We would like to be able to make measurements with code like:

```
s := Sample.new(NREP);
for(i:=0; i<NREP; i++) {
        t = tstamp();
        sys->sleep(0);
        t = tstamp() - t;
        s.add(t-base);
}
```

Where s is an object that can be used to record up to NREP measurements and base is our predetermined estimate of the cost of actually making the measurement. The Limbo function tstamp() uses *devbench* to obtain a microsecond timestamp. Such a measurement would give us an idea of the cost of making a sys->sleep(0) call and by examining a sample of such measurements we could reason about the true cost of the call. In particular, we would get a feel for just how reliable our estimate of the true cost was.

The first task was to determine a suitable value to use for base by measuring how long it took to do *nothing*.

This benchmark is called BASE in the *devbench* benchmark suite.

```
s := Sample.new(NREP);
for(i:=0; i<NREP; i++) {
        t = tstamp();
        t = tstamp() - t;
        s.add(t-base);
}
(n, mean, min, max, std) := s.stat();
base = min;
```

By repeating the measurement a large number of times and then by choosing the minimum measurement we could hope to choose a sensible value for base. Our confidence in the chosen value and in the results of subsequent benchmarking would be greatest if the distribution of measurements was *narrow* with most of the values close to the minimum.

Initial Results

We ran the initial tests on a 150MHz Pentium PC with a standard Inferno 2.0 native kernel and later with an Inferno 2.3 native kernel. The commands

```
bench -o >x
bgrep BASE x | plot
```

run the benchmark suite leaving the observations in a file called x and then extract the line containing the results for the test called BASE and plot them to a Wm window under Inferno. The command bgrep is used in place of grep because the Inferno grep uses sys->print to output lines and so inherits a very small upper bound on line length. The -o option to bench delivers the individual observations rather than summary statistics. The BASE test repeats the above empty loop 1000 times. Plotting the observations in the order in which they were observed gives the surprising result shown in Figure 1. The majority of values appear to be small, but the graph is dominated by the relatively

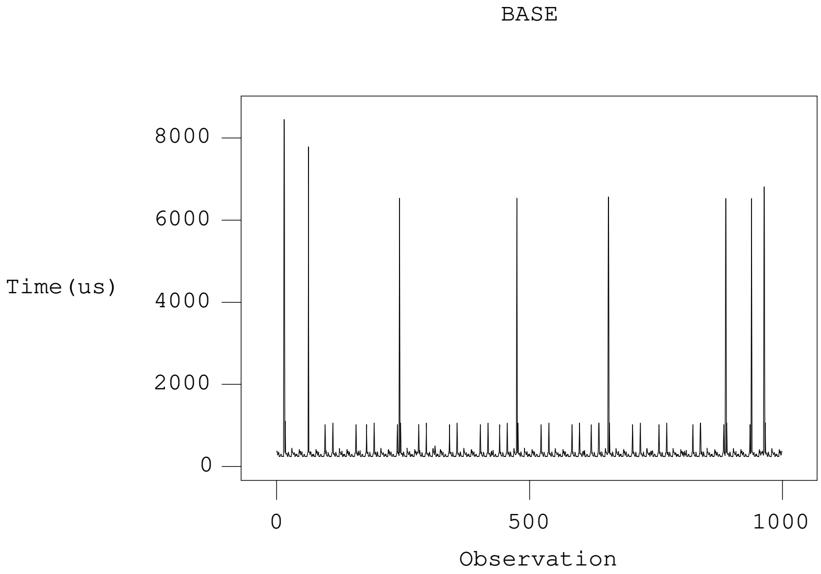

Figure 1. Showing the unexpected variation in the BASE results.

Actually, to generate the graphs in this document I used plot -g which produces grap(1) output.

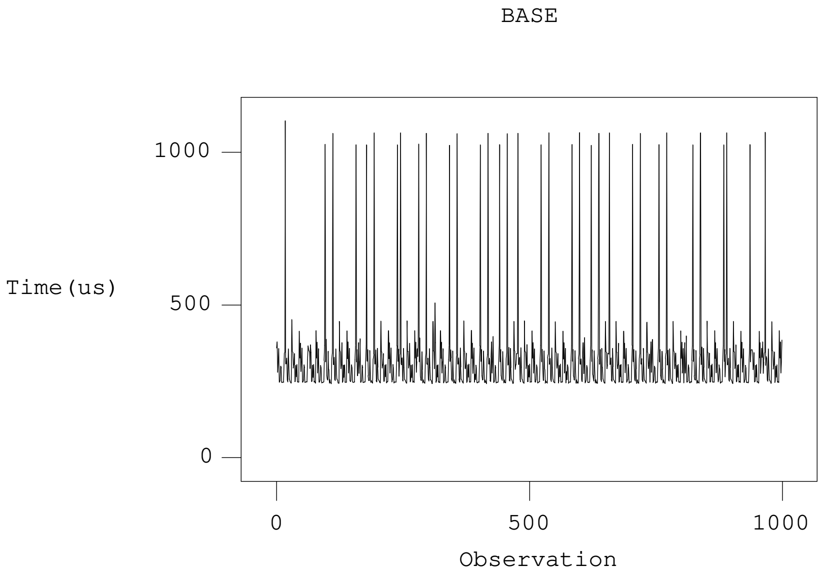

Figure 2. Great variability in the BASE results, even with the large outliers removed.

small number of very large irregular observations which lie at around 6 milliseconds to 8 milliseconds and a clearly discernible second set of values which look to form a more regular pattern at about the 1 millisecond level. The smaller set of values can be seen more clearly by removing the large values and rescaling the graph. The -o option to Plot invokes a crude scheme to remove a layer of outliers. It does this by calculating the sample mean and standard deviation and removing values which lie more than two standard deviations away from the mean. The process is repeated for each occurrence of the -o option to plot. Thus

```
plot -ooo
```

will remove three layers of outliers and then plot the remaining values.

Running

```
bgrep BASE x | plot -o
```

gives the output shown in figure 2.

Again, a large number of small values (less than 500 microseconds) and a smaller number of larger values lying at around about the 1 millisecond level. Running the commands

```
bgrep BASE x | plot -p 1h
bgrep BASE x | plot -p 1h -o -z
```

gives a histogram showing the distribution of the results. Firstly, with all the values and then with one level of outliers removed . In Figure 3 we can see a small set of values up above the 7 millisecond level and then a slightly more focused set at around about 1 millisecond but together these groups make up much less than 10% of the total values. By removing a level of outliers we can see the distribution of the lower values more clearly, as shown in Figure 4. Even this lower cluster ranges from about 220 microseconds up to about 500 microseconds.

The -z option to *plot* forces a zero origin to be part of the plot.

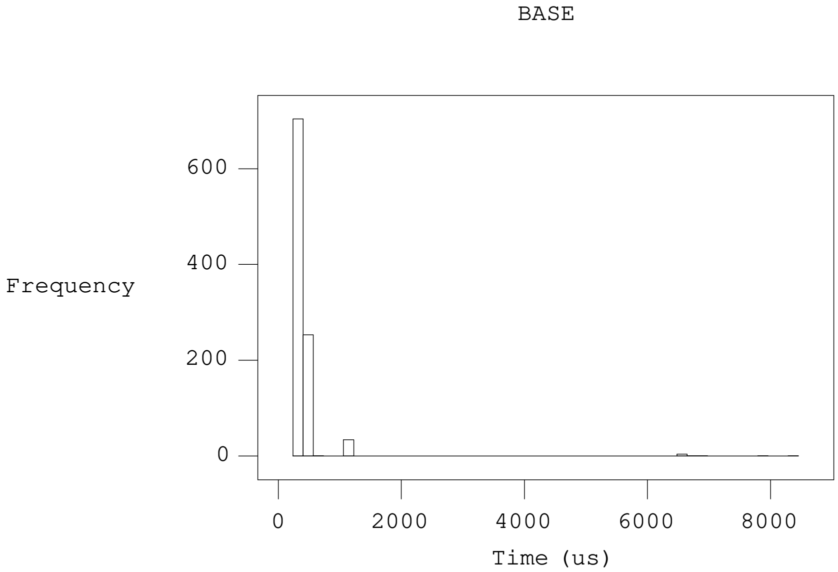

Figure 3. Histogram showing the distribution of results from the BASE benchmark.

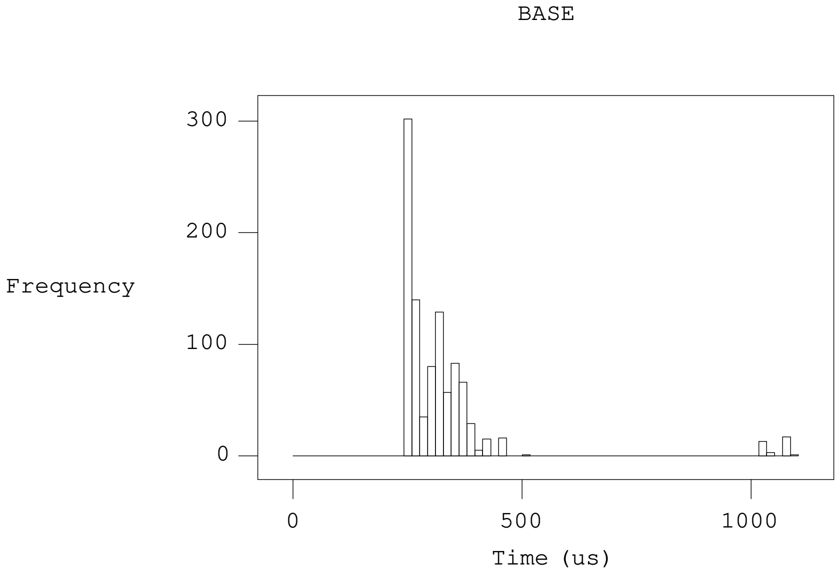

Figure 4. Histogram showing the distribution of results from the BASE benchmark with the large outliers removed.

We can conclude from these initial results that it is not going to be possible to use this mechanism to perform accurate or reliable benchmarking unless its performance can be improved.

The Measurement Process

At this stage we re not actually measuring anything real, but looking to obtain some confidence in our measurements. We have started by observing times to complete the measurement of *nothing*. We do this by writing a Limbo program (*bench.b*) which contains the following code:

```
t0, t1, base: big;
NREP: con 1000;

s := Sample.new(NREP);
treset();
for(i:=0; i<NREP; i++) {
        t0 = tstamp();
        t1 = tstamp();
        s.add(t1-t0);
}
(nil, nil, base, nil, nil) = s.stat();

iob.puts("BASE:1l:Observation::Time:us        ");
s.obs(iob);
iob.puts("\n");
iob.flush();
```

The middle section of code performs the benchmark by taking two consecutive timestamps and noting the difference between them. This is repeated 1000 times and the observations accumulated in the sample variable s. The function s.stat() returns a tuple containing count, mean, minimum, maximum and standard deviation for the accumulated values. We assign the minimum value to our variable base. The function treset() arranges for the timestamp code to start returning small values and so reduces the likelihood of overflow or signing related issues.

The last section of code outputs one line containing the individual observations separated by spaces and with a suitably formatted initial *word* which labels the line so that plot can make a useful interpretation of the results.

The functions treset() and tstamp() make use of files served by *devbench* and look like this:

```
tsfd: ref Sys->FD;
treset()
{
        tsfd = sys->open("/dev/busec",     sys->OWRITE);
        if(tsfd == nil)
                return;
        buf := array[1] of byte;
        n := sys->write(tsfd,   buf, len buf);
        tsfd = sys->open("/dev/busec",     sys->OREAD);
}
tstamp(): big
{
        buf := array[128] of byte;
        n := sys->read(tsfd,   buf, len buf);
        if(n < 0)
                return big 0;

        return big string buf[0:n];
}
```

Devbench must have been previously bound into the namespace

```
bind -b '#x' /dev
```

so that the files

```
/dev/bctl
/dev/bdata
/dev/busec
```

are available. A microsecond timestamp is provided by reading from /dev/busec and is reset to zero each time anthing is written to it. The function reset() resets the timestamp by writing to the file and then opens it for reading. The function tstamp() reads the timestamp from the file and converts it into a big value for use in the program. The device driver ignores the offset when reading from the file so that it is not necessary to seek to the beginning of the file before each read. The important part of the benchmark consists of two consecutive calls to tstamp(). We need to identify where time is being spent; either in the calls to tstamp() or else inside the function itself.

Costing

The *devbench* suite of programs includes one for timing individual lines of Limbo code and breaking down the time spent on each line into times for each of the Dis operations invoked by that line; the program is called *cost*. To find out how long each part of the benchmark timing loop takes we can invoke

```
cost -f bench.b -s bench
```

Which results in the following output - edited to just show the timing loop and the function tstamp(). The first line is the output from *bench* giving the mean, minimum, maximum and standard deviation of microsecond times for 1000 repetitions of the loop. We see a minimum of 402 microseconds and a maximum of just over 9 milliseconds. The remaining lines are the output of *cost*; those beginning with a dash are Dis operation times for the preceding Limbo source code line. Dis operation lines contain an operation name, an execution count and the mean, minimum, maximum and standard deviation of execution times.

```
BASE:summary:Sample:n:Time:us       1000  486  402  9053  499

--  bench.b
...
         treset();
-  frame       1    2.13  us    2.13  us    2.13  us       0  ns
-  call        1     807  ns     807  ns     807  ns       0  ns
         s  :=  Sample.new(NREP*10);
-  mframe      1    4.85  us    4.85  us    4.85  us       0  ns
-  movw        1    2.27  us    2.27  us    2.27  us       0  ns
-  lea         1    1.57  us    1.57  us    1.57  us       0  ns
-  mcall       1    2.71  us    2.71  us    2.71  us       0  ns
         for(i:=0;   i<NREP*10;   i++)  {
-  movw        1    1.99  us    1.99  us    1.99  us       0  ns
-  blew     1001    1.60  us    1.51  us   34.04  us    1.03  us
                  t0  =  tstamp();
-  frame    1000    2.43  us    2.29  us    3.40  us     133  ns
-  lea      1000    1.00  us     953  ns   27.74  us     847  ns
-  call     1000    1.45  us    1.29  us    2.24  us     153  ns
                  t1  =  tstamp();
-  frame    1000    3.45  us    3.01  us    5.57  us     293  ns
-  lea      1000    1.22  us    1.02  us    1.61  us     173  ns
-  call     1000    1.44  us    1.29  us    1.75  us     140  ns
                  s.add(t1-t0);
-  mframe   1000    3.65  us    3.37  us   24.73  us     700  ns
-  movp     1000    1.99  us    1.48  us    2.96  us     267  ns
-  subl     1000    2.43  us    2.23  us    4.80  us     173  ns
-  mcall    1000    2.11  us    1.95  us    2.51  us     120  ns
-  addw     1000    1.62  us    1.48  us    2.50  us     127  ns
-  jmp      1000    1.50  us    1.37  us    1.81  us     120  ns
         }
         (nil,  nil,  base,  nil,  nil)  =  s.stat();
-  mframe      1    2.66  us    2.66  us    2.66  us       0  ns
-  movp        1    1.39  us    1.39  us    1.39  us       0  ns
-  lea         1    1.41  us    1.41  us    1.41  us       0  ns
-  mcall       1    1.37  us    1.37  us    1.37  us       0  ns
-  movl        1    1.56  us    1.56  us    1.56  us       0  ns
         if(sflag)
-  beqw        1    2.14  us    2.14  us    2.14  us       0  ns
                  iob.puts("BASE:summary:Sample:n:Time:us          "
                                                +  s.str()   +  "\n");
-  mframe      1    4.01  us    4.01  us    4.01  us       0  ns
-  movp        1    2.87  us    2.87  us    2.87  us       0  ns
-  mframe      1    1.75  us    1.75  us    1.75  us       0  ns
-  movp        1    1.69  us    1.69  us    1.69  us       0  ns
-  lea         1    1.15  us    1.15  us    1.15  us       0  ns
-  mcall       1    2.35  us    2.35  us    2.35  us       0  ns
-  addc        1   19.57  us   19.57  us   19.57  us       0  ns
-  addc        1    8.55  us    8.55  us    8.55  us       0  ns
-  lea         1    1.49  us    1.49  us    1.49  us       0  ns
-  mcall       1    3.65  us    3.65  us    3.65  us       0  ns
         iob.flush();
-  mframe      1    4.50  us    4.50  us    4.50  us       0  ns
-  movp        1    2.13  us    2.13  us    2.13  us       0  ns
-  lea         1     967  ns     967  ns     967  ns       0  ns
-  mcall       1    1.89  us    1.89  us    1.89  us       0  ns
         }
...
```

```
# return timestamp in microseconds
tstamp(): big
{
        buf := array[128] of byte;
- newa     2000    6.98  us    5.37  us   80.91  us    2.43  us
        n := sys->read(tsfd, buf, len buf);
- mframe   2000    3.05  us    2.59  us    4.05  us     227  ns
- movp     2000    2.33  us    2.14  us    3.21  us     120  ns
- movp     2000    1.19  us    1.14  us    2.93  us      73  ns
- lena     2000    1.07  us     907  ns   34.48  us     760  ns
- lea      2000     893  ns     860  ns   33.00  us     907  ns
- mcall    2000  133.81  us   89.88  us    8.61  ms  514.44  us
        if(n < 0)
- blew     2000    2.27  us    1.76  us    3.43  us     313  ns
                return big 0;

        return big string buf[0:n];
- movp     2000    2.21  us    1.67  us    3.38  us     253  ns
- slicea   2000   10.06  us    8.35  us   39.31  us    1.40  us
- cvtac    2000   12.02  us   11.13  us   16.63  us     553  ns
- cvtcl    2000   16.39  us   15.78  us   45.16  us     787  ns
- ret      2000   10.52  us    9.79  us   13.97  us     333  ns
}
```

We can see from these figures that times for both calls to tstamp() are reasonably low. But looking at the call to sys->read() in tstamp() we can see a maximum of just over 8½ milliseconds for the *mcall* operation. We see too that this operation also has a high standard deviation of 514 microseconds across 2000 calls and so it looks like it may be the cause of the large observed delays.

In order to get a second view of what was happening we surrounded the timing code with calls to enable kernel profiling.

```
profon();
for(i:=0; i<NREP*10; i++) {
        t0 = tstamp();
        t1 = tstamp();
        s.add(t1-t0);
}
profoff();
...
profon()
{
        if(sys->write(kpctl, array of byte "startclr", 8) < 8)
                error("kprof start");
}

profoff()
{
        if(sys->write(kpctl, array of byte "stop", 4) < 4)
                error("kprof stop");
}
```

Having previously bound in *devkprof*

```
bind -b '#T' /dev
```

and opened */dev/kpctl* for writing. By examining the contents of */dev/kpdata* with *kprof* we see that the garbage collection functions *markheap* and *rungc* are taking up a surprisingly large amount of kernel time for this operation.

```
total:  1460      in kernel  text:  1460     outside  kernel  text:  0
KTZERO  80100000
ms         %      sym
130        8.9    markheap
110        7.5    iunlock
100        6.8    xec
90         6.1    rungc
80         5.4    chartorune
70         4.7    c2string
60         4.1    numbconv
50         3.4    string2c
40         2.7    sched
40         2.7    wakeup
30         2.0    splhi
30         2.0    freeptrs
30         2.0    memset
30         2.0    dodiv
30         2.0    _divvu
30         2.0    strtoll
20         1.3    _mulv
20         1.3    vmachine
20         1.3    unlock
20         1.3    runproc
20         1.3    DEA
10         0.6    kwrite
```

Interpreting the Results

In order to progress we need to look more closely at what is happening inside the call to sys->read. As far as the kernel is concerned, there are two kinds of *mcall* operations; those which result in a call to a builtin function and those which result in a call to a non-builtin function in another module. Sys->read is of the former kind, it is a system builtin whose definition lies in the function *Sys_read*. When executing *mcall* for a normal function the virtual machine primes the register set so that on the next iteration the operations will be fetched from the code associated with the new function. When executing *mcall* for a builtin the virtual machine actually executes the associated function before returning to the next operation. The code for *Sys_read* looks like this:

```
void
Sys_read(void *fp)
{
        int n;
        F_Sys_read *f;

        f = fp;
        n = f->n;
        if(f->buf == (Array*)H) {
                *f->ret = 0;
                return;
        }
        if(n > f->buf->len)
                n = f->buf->len;

        release();
        *f->ret = kread(fdchk(f->fd), f->buf->data, n);
        acquire();
}
```

The important thing to note is that in common with all builtin functions which must perform some kernel function *Sys_read* releases the interpreter before carrying out its task and then acquires it again before returning to the body of the *mcall* operation. Releasing the interpreter means removing this Dis thread from the list of threads to be run and making ready another virtual machine from the list of kernel processes waiting to use the interpreter, if necessary by first creating a new one. Once the interpreter has been released the process carries out its operation, in this case a call to *kread*, and then attempts to reacquire the interpreter. If no other process is using the interpreter it will get it back immediately, otherwise the process sets its state to *Queueing* and calls sched() to wait for its turn. How long it has to wait depends on how many other threads are running and whether they are compiled or interpreted. If another process wants to use the interpreter then it may run for a short period of time before making a call to *rungc()* to perform a small amount of garbage collection and then yielding control to our process. If the other process is running an interpreted thread then it may execute for up to 2048 Dis instructions or for a more variable time if it is a compiled thread. We can see the likely effect of this by timing calls to *rungc*. The following code does this inside *devbench*:

```
log("GCSCHED:1d:Observation:n:Time:us");
for(i=0; i<1000; i++) {
        (*ts)(&t0);
        rungc(head);
        (*ts)(&t1);
        log(" %.2f", ts2us(t1-t0));
        release();
        acquire();
}
log("\n");
```

The calls to release() and acquire() allow other Dis threads to run and so exercise the memory allocation code in much the same way that the same calls in tstamp() do. On each call to rungc() the garbage collector visits 50 blocks in the heap. The times for 1000 calls to rungc are shown in Figure 5. Running

```
cat results | bcut 1-201 | plot
```

This benchmark is called GCSCHED in the *devbench* benchmark suite.

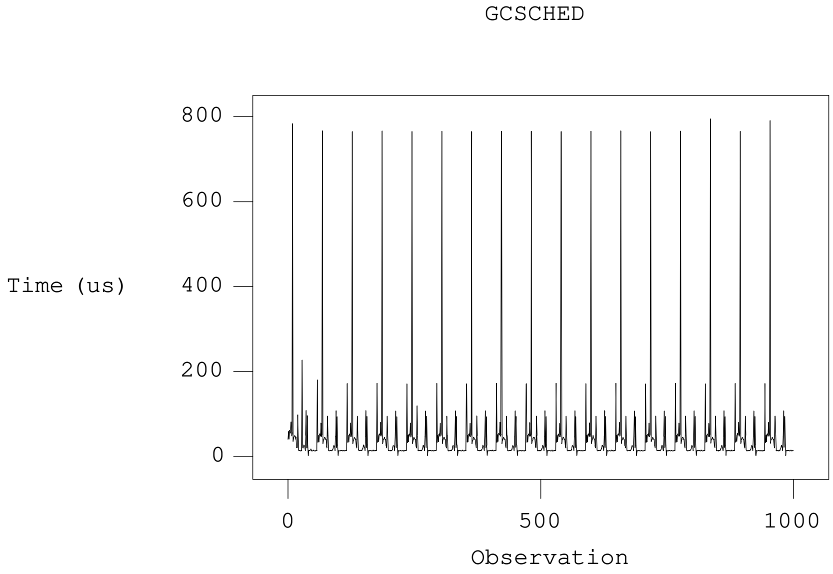

Figure 5. Times for 1000 calls to rungc().

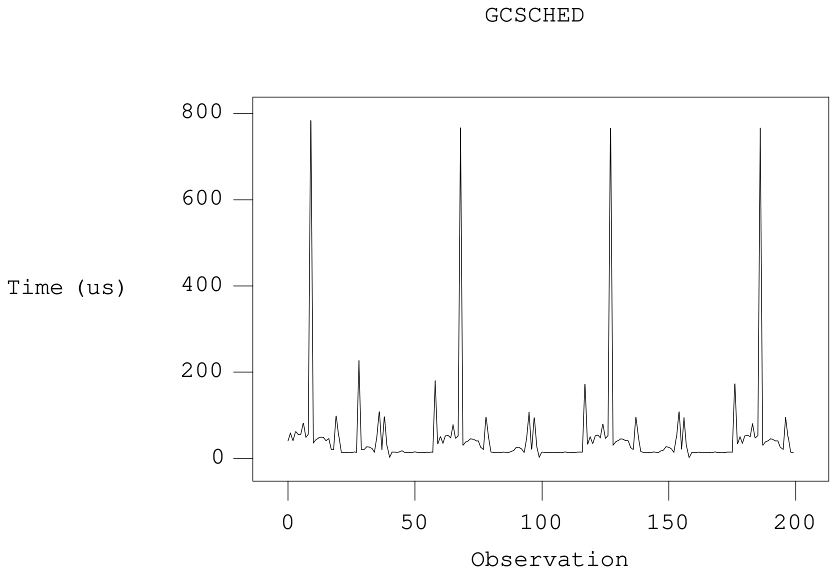

Figure 6. Times for the first 200 calls to rungc().

will give us just the first 200 data points in a little more detail; these are shown in Figure 6. We can see that the time for calls to rungc has a very repeatable pattern, but there is no sign of the large 6 millisecond delay.

Garbage collection is also done during idle virtual machine cycles. If no thread is ready to run the interpreter calls execatidle() and then sleeps until there is something to do. Execatidle() attempts to complete 3 epochs of garbage collection, checking after each call to rungc() to see if any thread is ready to run; if a thread is ready, normal interpretation is resumed immediately.

The following code in *devbench* is used to determine how long it might take the garbage collector to run 3 epochs :

```
int
idlegc(void *p)
{
        int done;
        Prog *head;
        vlong t0, t1, tot;
        USED(p);

        head = progn(0);        /* isched.head */
        done = gccolor + 3;
        tot = 0;
        while(gccolor < done && gcruns()) {
                if(tready(nil))
                        break;
                (*ts)(&t0);
                rungc(head);
                (*ts)(&t1);
                tot += t1-t0;
        }
        log(" %.2f", ts2us(tot));
        nidle--;
        if(nidle == 0) {
                log("0);
                return 1;
        }
        return 0;
}
```

The code is invoked by assigning 100 to nidle and then calling

```
atidle(idlegc, 0);
```

so that the interpreter will call this code when idle in the same way that it calls the normal garbage collection code.

The function tready() is defined as

```
tready(void *a)
{
        return isched.runhd != nil || isched.yield != 0;
}
```

returning *true* whenever there is a thread in the run queue or when another interpreter kernel process has called *acquire()* to request that control of the interpreter be *yielded* to it. In fact, the code mirrors the garbage collection code in the interpreter, breaking when another thread is ready to be run.

Figure 7 shows the times for performing three epochs of garbage collection. It is clear from these results that idle garbage collection can take some considerable time even when the system is relatively quiet. The times for completing 3 epochs of garbage collection were of the same order of magnitude as the large delays we were noticing with the timestamping earlier. However, the system is clearly designed to relinquish control whenever another thread needs to run. It was not, therefore, clear that this could be the cause of these delays.

This benchmark is called GCIDLE in the *devbench* benchmark suite.

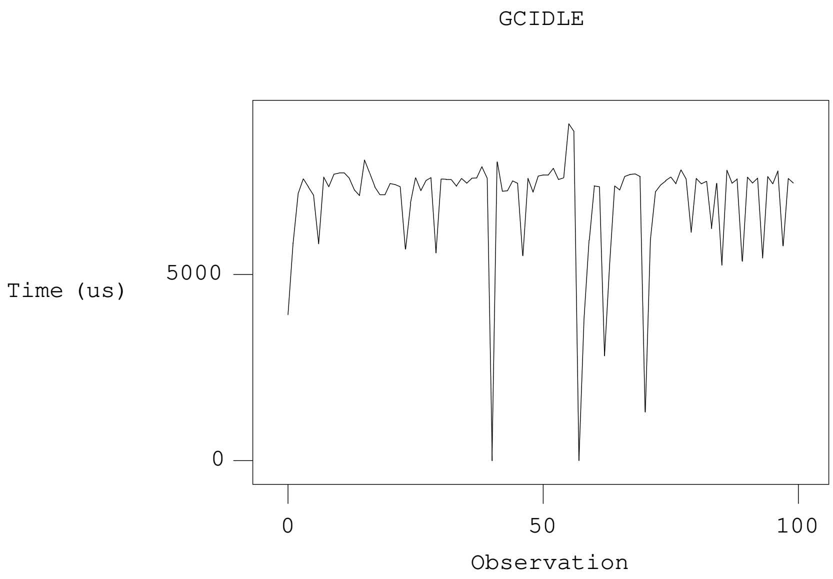

Figure 7. Times for performing three epochs of garbage collection.

An Explanation

Clock interrupts happen on this system at regular intervals of 10 milliseconds. At each such interrupt the handler checks to see if any kernel processes are ready to run, and if so it calls *sched*. This gives the system its preemptive scheduling. Suppose we are executing a call to sys->read(). If after calling *release* but before returning from the call to *kread* and calling *acquire* a clock interrupt goes off then another process may grab the interpreter and complete a scheduler quantum. If no other threads are ready to run because, like us, they are still in the kernel then tready() will return *false* and so it will choose to perform idle garbage collection. There is an upper bound on how long this may go on for; it is the minimum of the time it takes to perform three epochs of garbage collection and the time to the next clock interrupt which may be as much as 10 milliseconds. This would give rise to the periodic high values we have seen and would likely result in delays of the same order of magnitude as those that we have seen.

One way to check this hypothesis would be to modify the condition upon which the idle garbage collection is halted to include a test for kernel processes that are ready to run.

The existing code in the function *port/dis.c:execatidle()* breaks from the process of garbage collection only when another interpreter requests control or when another thread is ready to run.

```
done = gccolor+3;
while(gccolor < done && gcruns()) {
        if(isched.yield != 0 || isched.runhd != isched.runtl)
                break;

        rungc(isched.head);
}
```

The result of this is that when a thread makes a call that takes it into the kernel it is effectively given a lower priority than the idle time garbage collection. This effect can only be seen if a clock interrupt happens which results in control being switched to an interpreter thread which completes its execution quantum and enters idle time garbage collection.

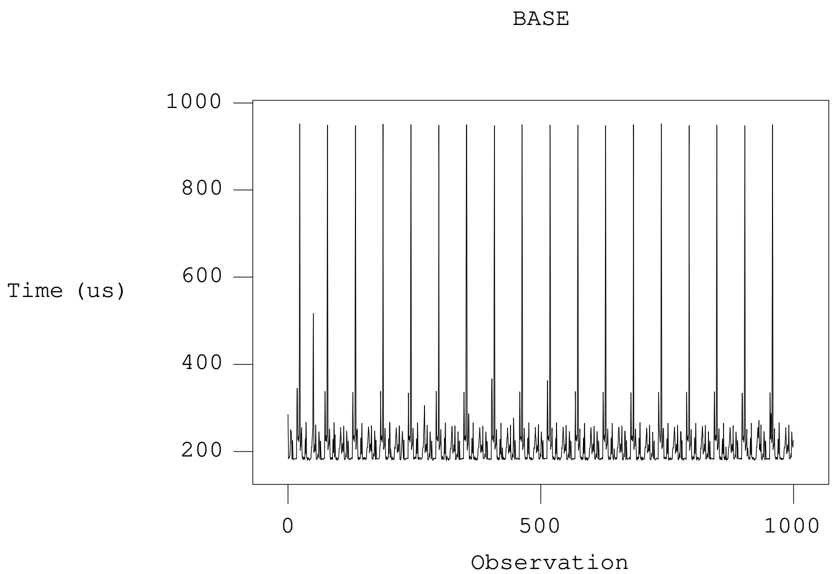

Figure 8. Improved results from idle garbage collection change

Ordinarily, a thread in the kernel will either run to completion or will, itself, call sched() while waiting for an event to occur.

This effect will be visible in any Inferno thread that makes a call to a system function which then results in a *release* of the interpreter—not just this benchmarking application.

Improving the Results

We have changed the code in *execatidle* to call *sched* on detection of ready kernel processes.

It now looks like this:

```
done = gccolor+3;
while(gccolor < done && gcruns()) {
        if(isched.yield != 0 || isched.runhd != isched.runtl)
                break;
        rungc(isched.head);
        sched();
}
```

Processes in the kernel running at the same priority as the collector are interleaved with the collector. If such a process returns from a kernel operation and attempts to *acquire* the interpreter then this loop will exit the next time around because isched.yield will be set. Kernel processes which do not require the interpreter, such as the network timers, will get to run without interrupting the garbage collection. The effect of this change can be, readily, seen in much smaller delays shown in Figure 8. The large 6 millisecond delays have disappeared but regular 1 millisecond delays still remain. Looking more closely at the first 200 times, in Figure 9, we can see that the pattern looks very similar to the pattern we know that we get from repeatedly calling *rungc()* . Indeed, this is most likely the cause of the regular pattern. The call to *rungc* after completing each

A first attempt checked the number of processes ready to run, making the call to *sched* conditional, but that did not correctly account for their priorities relative to the process running the collector.

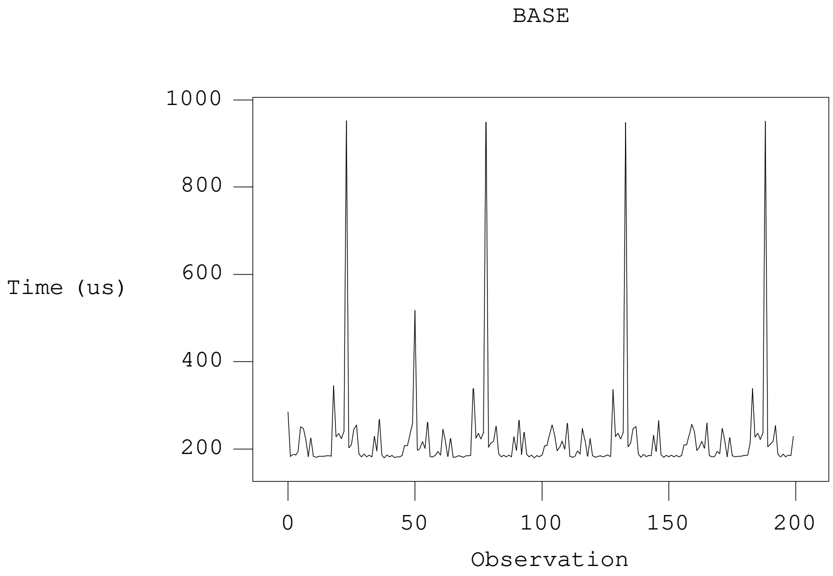

Figure 9. Improved results from idle garbage collection change – the first 200 values

quantum of scheduler activity results in quite a variable delay which means that the real time taken in executing tstamp() will be difficult to predict. This will be true for any Limbo function that calls a system builtin which in turn releases the interpreter.

I suspect that the single large delay will occur when the garbage collector completes an epoch and runs through marking each root as a propagator. There may be ways to improve the situation by smoothing this operation, perhaps by making it incremental or by temporarily disabling garbage collection or by optimising the conditions under which garbage collection is done. However, garbage collection has to be done sometime, and it is not clear that, in general, deferring it is always the best thing to do – we may get good results when the system is quiet but very bad results when garbage collection is required. It would seem likely that there may have been some work done on this aspect of the virtual machine in later incarnations of the Inferno system.

There are two ways forward. We could accept that garbage collection is an integral part of the system and not try to measure small units of time – instead, we could measure the time for a lot of operations and average them out and then subtract the known garbage collection background count from the overall time to get a feel for how much time taken performing our operation of interest. There are several problems with this approach. Firstly, it seems unreasonable not to be able to measure something small with Inferno; frustrating too, when the problem is caused by a visually regular disturbance. Secondly, measuring something over a long period of time gives ample opportunity for other factors to interfere with the operation of interest (for example, interrupts from external interfaces).

The second way forward is to modify our measurement process so that its interaction with the virtual machine is minimised in such a way that it can be used more predictably and, hence, is more reliable. We did this by providing a builtin function bench->microsec() and by providing an interface which allowed garbage collection to be disabled for a period of time.

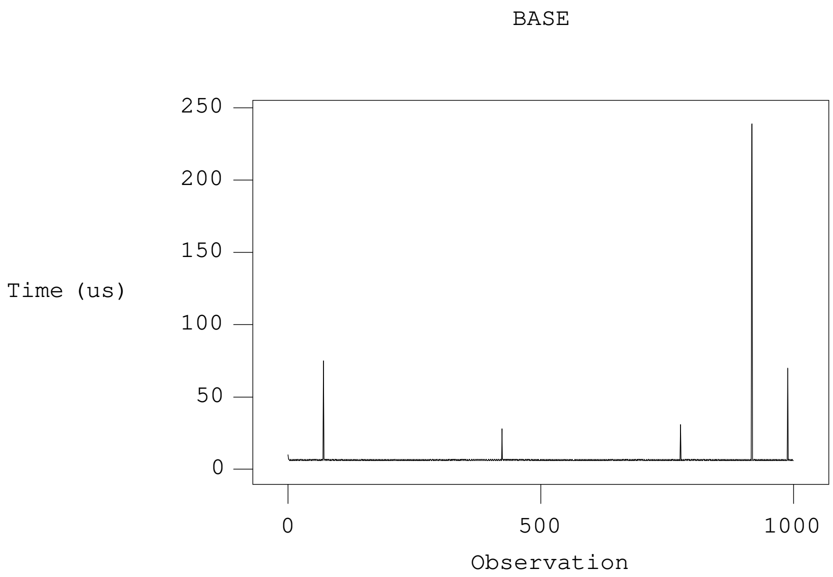

Figure 10. Further improvement by using a builtin timestamp.

The Limbo module prototype looks like this:

```
{
        PATH:    con  "$Bench";

        microsec:          fn():  big;
        disablegc:         fn();
        enablegc:          fn();
};
```

and the Limbo function to return a timestamp just calls the builtin.

```
xstamp():   big
{
        return   bench->microsec();
}
```

Using *xstamp()* in place of *tstamp()* without disabling garbage collection results in a significant improvement to the times returned by the BASE benchmark. These results are shown in Figure 10. I suspect that the few large values are as a result of external interrupts or scheduler quantum garbage collection. The distribution of these values is shown in Figure 11. Out of the 1000 measurements all but 6 of them were recorded at 6 microseconds.

The kernel used to make this recording (and all subsequent in this document) has had the modifications made to the idle garbage collection code described earlier. We found that switching to using a builtin with the old kernel did result in much better times but could still occasionally fall foul of clock interrupt.

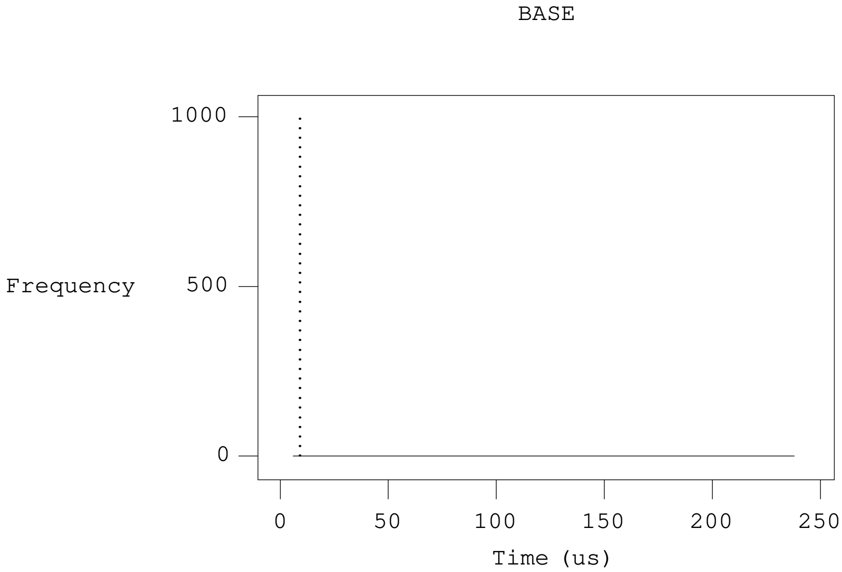

Figure 11. The distribution of the results shown in Figure 10

By wrapping the entire benchmark with calls to

```
bench->disablegc()
```

and

```
bench->enablegc()
```

which disable and then reenable all garbage collection the figures can be improved even further.

The histogram of timestamps obtained when garbage collection is completely disabled is shown in Figure 12. It is clear that the values we are now getting from *xstamp()* are much more acceptable. Out of the 1000 measurements, five were recorded at 12 microseconds, one at 8 microseconds and the remainder at either 6 or 7 microseconds.

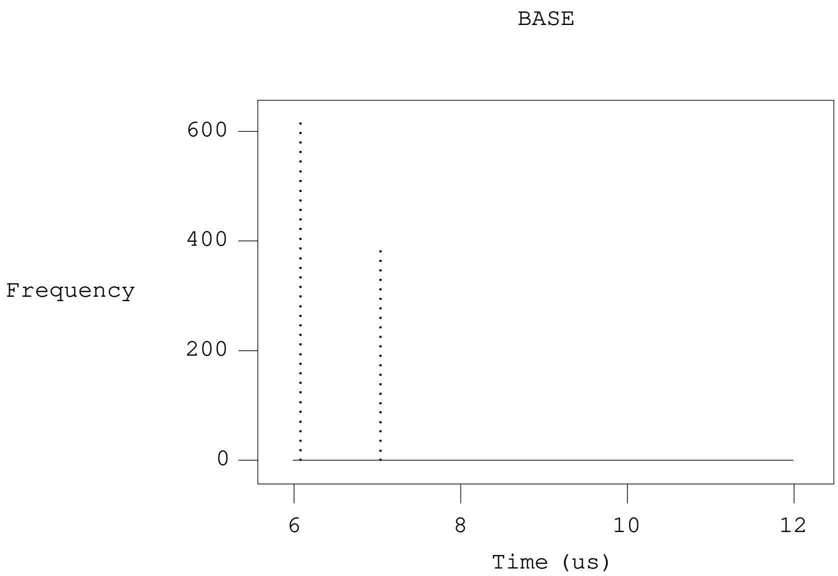

Figure 12. The distribution of results obtained when garbage collection is completely disabled.

Conclusion

Comparing the histogram for the original measurements with the one obtained by making a simple change to the interpreter and by using a builtin instead of using sys->read() it is clear that considerable improvements have been made. This behaviour will be visible in any Inferno thread that makes a call to a system function which, in turn, causes the interpreter to be released. These effects will be present in any Inferno application that uses, for example, sys->read(), sys->write(), sys->mount() and sys->bind() or any other such builtin system functions.

The decision to enable or disable garbage collection whilst benchmarking is a difficult one. Disabling it results in slightly better measurement, but is somewhat less realistic. The improvement gained by disabling it doesn t seem good enough to justify the likely criticism which might be levelled at the subsequent results.

It would appear that to enable reliable benchmarking of Limbo programs under Inferno two changes should be applied. Firstly, a simple change should be made to the interpreter to prevent idle garbage collection from grabbing a large time slot when scheduling occurs at a clock interrupt and one or more interpreter threads are in kernel operations. Secondly, making the timestamp function a builtin results in a simpler interface to the kernel and results in a measurement process whose execution time is reliable. Applied together, these changes make it possible to perform reliable benchmarking of Limbo programs under Inferno.