Hardware Profiling of Kernels.

Andrew McRae

Megadata Pty Ltd.
2/37 Waterloo Rd
North Ryde
andrew@megadata.mega.oz.au

Or: How to look under the Hood while the Engine is Running.

This paper describes a method of accurately measuring and profiling kernel code in real time. Some background is covered, which describes other more common, and easier, methods of profiling, and why these methods were rejected. Some goals are stated, and a proposed hardware/software solution is described. As a case study, the profiling method is used to evaluate a kernel incorporating the Berkeley TCP/IP networking code; the results of this exercise are presented, showing how tracing of network software in real time highlights optimal or non-optimal code paths.

Warning to software people: this paper contains some descriptions of hardware.

Warning to non-kernel-hackers: this paper has lots of kernel hacking in it.

1. Optimisations.

Michael Jackson (the safe [non-dangerous] one) has made some pertinent remarks about optimisation.

Jackson's First Rule of Optimisation:

Don't do it.

Jackson's Second Rule of Optimisation (for very experienced programmers):

Think about it, then don't do it.

This expresses a well founded caution, often ignored by the naive, who would do well to remember the Prime Directive for Optimisation (as espoused by Kernighan and Plauger):

Make it right before you make it fast.

Even so, much effort goes into making programs as fast as possible, leading to a plethora of optimising pre-processors, compilers, assemblers etc. Given a poor or slow design, however, the best optimising compilers are generally not of great benefit. Experience has shown that if a piece of software is not performing, reviewing the design is often the best, sometimes only, way of obtaining significant improvement.

Testing compiler optimisations is relatively easy; for a given set of test cases, the resultant instruction stream can be analysed to check for near-optimal code size or length of code path. Even given perfect translators, how can we test our designs in the same way? Generally programs are comprised of a number of algorithms, often interacting. How do we decide which algorithm is slow, or if a particular design section is far from optimal?

The key to all optimisation is to understand where or if it should be applied, and therefore the Golden Rule of Optimisation is:

Measure BEFORE you optimise.

UNIX† has a number of tools to help in this area; compiler profiling allows time based and function

† UNIX is a registered trademark of AT&T UNIX System Laboratories.

entry/exit profiling to be incorporated into programs, which then allow operating statistics to be extracted and analysed. Generally this is good enough for most programs, as the programs are not usually interacting with, or affected by, real world events. Simulators have also been used to effect by providing a much higher degree of granularity to profiling, allowing tracing of code paths etc. But often setting up and executing a simulation is difficult, or the simulator cannot provide for interaction with ‘real time’ events.

2. Kernel Measurement.

Kernels are a special case in that they must interface to certain real world entities, such as devices, networks, memories, clocks etc. Subtle and complex interactions occur between device drivers, processes and external events, and anyone who has attempted to fix bugs caused by these interactions will appreciate the difficulty in ascertaining hard data as to where optimisation is best applied. Kernel measurement has progressed to the point where it could be labelled A Black Art, instead of the realm of Wild Guesses. Kernels can be profiled, and through the use of statistics to count specific events, it is possible to do a fair job of deciding where a kernel is very slow, slow or not-as-slow-as-the-rest.

Some areas of kernels can be measured in the same way as user programs, using function counting and gross clock profiling, but these are not the interesting areas. What happens if one wishes to profile the clock interrupt code itself? What happens if you wish to measure the time taken to process character input interrupts, or discover the optimal code path taken for processing back-to-back packets through a certain protocol stack, checking the time to reply with acknowledgements?

The fly in the ointment is that kernel profiling is in the same vein as the Heisenberg Uncertainty Principle i.e the more accurate your measurements, the more you are perturbing the environment in which the kernel is running, and the less likely of getting data which reflects the actual state of the unprofiled kernel. Other methods are available which are non-intrusive, such as connecting large amounts of hardware to record the instruction stream; this is expensive and requires specialised hardware. Another problem with this method is that it often does not cope with cache effects; instruction caches must be turned off, thus ruining the non-intrusive nature of the measurement.

So we are faced with a dilemma; in order to rationally test kernel designs and code, we need accurate measurements, but in obtaining these measurements we change the environment of the kernel, and possibly introduce erroneous measurands (and consequently make wrong design decisions). Any kernel profiling system must be as non-intrusive as possible, or at least keep the effect of measurement to a minimum so that it does not grossly change the timing characteristics.

3. The Goals.

As a result of much software written in an embedded environment, a great deal of it driver and kernel related, I became increasingly interested in being able to easily measure and profile the software, and so make rational and informed judgements concerning algorithms and coding techniques. Faced with the regular need to discover why things were not responding at the expected speed, it quickly became clear that the human brain is not a good enough simulator to handle the complex timing interactions occurring within a kernel. Some early solutions to the problem was to use statistic counters, but this was usually too gross a measurement to help. Another favourite method was to press-gang a hardware engineer to connect an oscilloscope to the equipment; this enabled finger-in-the-wind type measurements, and certainly helped when external hardware was being controlled.

Sophisticated tools such as logic analysers provided a major benefit, as whole sequences of events could be trapped and examined in the cold light of day. More intelligent software within the analysers also allowed instruction disassembly, which made easier work of following code paths, but this was generally tedious and unfriendly because of the difficulty in relating the raw instruction stream back to the source code. Other software can be made to perform time based profiling, but the sampling granularity was generally too coarse to be of any use.

In-circuit Emulators generally are considered the top of the heap for embedded development, and come with complete suites of cross-compilers, assemblers, remote debuggers and hardware which allows all manner of tracing and measuring programs. They also come with Rolls Royce price tags. Unfortunately they tend to be black boxes when it comes to analysing the data; it is often difficult to extract the desired information from the raw timing data.

I still had a need to find out what was happening *inside* our embedded computers, but I had a limited budget. I wanted to do the equivalent of what our local car mechanic does, to open the hood, listen to the engine running, judge the revolutions, feel the temperature, and so forth.

By now I had tried several ways of getting the data, and I had formulated a wish list to describe what I wanted.

- Fine granularity of measurement, so that accurate profiling may be obtained.
- Little or no intrusiveness, so that taking the measurement will not affect the timing of the kernel.
- Integration with development tools or program source so that source level code paths may be traced with ease.
- Profiling to occur for all kernel operations within a selected interval, including clock interrupts, device interrupts, even periods when processor interrupts were locked out.
- If some hardware assists were to be employed, then some easy and portable method of connection should be used e.g not having to connect 96 separate clips to a PCB.
- Immune to instruction cache effects. In fact it should still work as expected with instruction caching enabled (as any ‘production’ code would run the cache enabled).
- Granularity to a function level (however short the function is) should be the worst case; however it is desirable to also profile within functions if possible.
- Whilst the inital target should be a 68020 system, future systems employing other types of microprocessors should be capable of being profiled.

It became clear that it is impossible to fulfill these goals with software alone. It is also clear that complex hardware did not offer an elegant (or cheap) alternative This paper describes a solution to this problem which is a better alternative to software only kernel profiling, and much cheaper than specialised and complex ICE hardware measurements of kernel operation. It attempts to meet the above goals, and also be simple and cheap enough to build without much effort (even a software engineer could manage it).

4. The Profiler.

Three basic building blocks are used in the profiling system proposed; the first is a hardware device that is used to record time and event data into a RAM block. The second is a modified C compiler that allows event triggering code to be inserted into key locations, and finally the last building block is analysis software that is used to decode the backtrace of events and relate it to the source code.

4.1. The Hardware.

The role of the hardware in the Profiler is very simple. Its job is to store timing information and some identification value. It purposely is as simple as possible, primarily because it was a first attempt at exploring what the basic hardware requirements were for meeting the goals. A lesser goal was cost minimisation; as long as the cost could be held to something below several hundreds of dollars than the Profiler could be built by just about anybody. Finally the Profiler is simple because I hate wire wrapping; it’s so much more tedious than writing software.

Commonly available parts were used, and the hardware prototyped on a simple breadboard using wire wrapping. A single electrically erasable PAL is used for the logic and timing functions; total parts cost less than $100 dollars. It has a chip count of 5 static RAMs, 5 counters, 1 PAL, 1 oscillator and 1 delay line. Having an EE PAL turned out to be a great boon, as it meant quite a bit of experimenting could take place to get the logic right, and also allowed extra facilities to be incorporated such as some display LEDs and control switches.

A block diagram appears below.

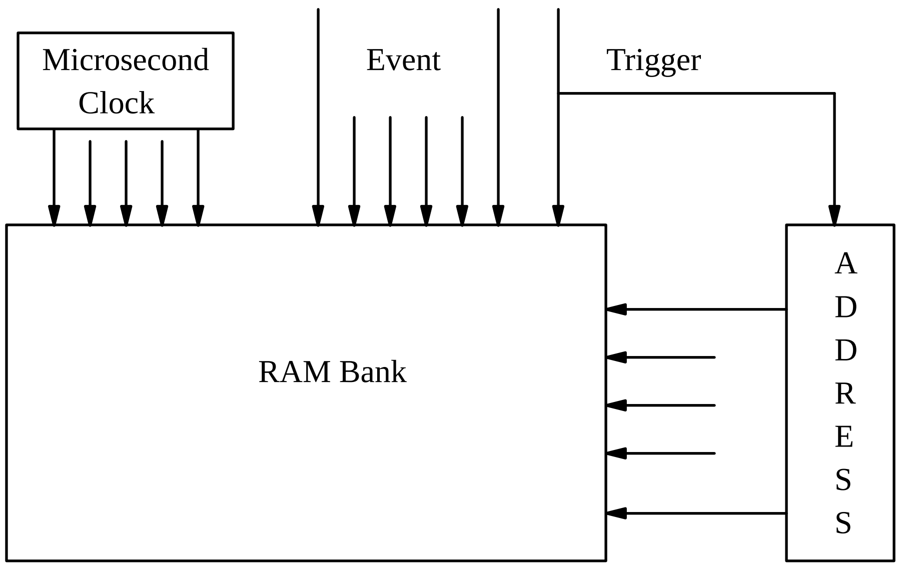

Figure 1

When the identification code (‘event’) is presented to the Profiler, then it stores this code along with a microsecond clock value into RAM. The RAM address is automatically incremented every time an event is stored, essentially storing the event/time in a large list. The list is 16K events long. The microsecond timer is 24 bits long, allowing a maximum time of 16 seconds between events before the time is wrapped around and information is lost. Note that this is the maximum time between events, not the total time that can be profiled - the analysis software only uses the timer value as an interval time, not as an absolute time. The event tag is 16 bits, allowing 65536 unique event tags.

The trick in this scheme is not the gathering or storing of the event/time data (a SMOH [Simple Matter Of Hardware]), but how to generate the event code, which must come from the equipment being measured. At first thought I had planned to detect specific instruction codes that represented function entry and exit points (e.g on the 680x0 every C function begins with a LINK statement, which translates to a hex instruction code of 4E56 - a return from subroutine translates to 4E75); this scheme had the advantage that is was totally non-instrusive but suffered from a number of shortcomings, such as inability to profile within subroutines.

It was clear that some software assist was required to actually trigger the profile events. One way was to place code at specific points to write the event value to a special location, but this assumes the equipment can actually perform the addressing required; the code translated to an instruction like:

```
MOVE.W    #val,EVENT_ADDR
```

All up this instruction cost 8 bytes, 4 read memory cycles and 1 write cycle (16 bit memory). It wasn’t really a minimal trigger.

An elegant solution presented itself when I realised that the target equipment always has a boot EPROM located at location 0. Generally this was not used once the kernel relocated itself to RAM. It also presented itself as a simple method of connecting the Profiler to the equipment, just by piggy-backing a EPROM socket onto some cable, using the cable to bring the appropriate signals into the Profiler and then plugging the boot EPROM into the socket. The selecting of the boot EPROM is the event trigger, and the address presented is stored as the event data.

Thus the event could be stored using the instruction:

```
TST.B    #event_val
```

Because `event_val` is accessing EPROM at location 0, it can use 16 bit absolute addressing. Total instruction length is 4 bytes, 2 16 bit reads, 1 8 bit read in total. On a 32 bit memory bus there is minimal overhead in using this event trigger.

Only 18 signal lines needed to be brought into the Profiler (16 address lines, the EPROM CE signal, and ground). This simple and easy method of connection allowed the Profiler to connect to **any** piece of equipment that contained a standard EPROM socket, without other connections. Power is obtained from the EPROM socket, so the Profiler is self contained.

A toggle switch provided simple control of the Profiler; when off it disabled all Profiler functions, when switched on it started the Profiler by zeroing the microsecond and address counter and enabled trigger values to be stored. When the address counter overflows the Profiler is automatically disabled, preserving the RAM contents from being overwritten from the address wraparound. A LED is activated when this occurs, indicating the end of a profiling run. Another LED is active whenever the Profiler is enabled, indicating data is being stored.

Once the data is in the RAM, how do we get it out? In the prototype, static RAM chips were plugged into SmartSockets™ (i.e battery backup in the RAM sockets), so once the data was collected, the RAM can be relocated to another board and uploaded to a host. This was sufficient for the prototype, but obviously tedious to do when performing a number of profiling runs. Plans for the next version provide for a much better method of extracting the data i.e all EPROM signals will be connected to the Profiler, and each bank of RAM will be multiplexed to the EPROM in turn, allowing the timing data to be automatically extracted via a user program or device driver.

The profiling scenario is now clear; simple software triggers are sprinkled in strategic locations throughout the target software. Each time one of the triggers is executed the time and trigger value is recorded. Once enough samples are stored, the timing data is retrieved and correlated back to the source code, allowing meaningful and accurate evalutation to take place.

And so I had a workable hardware/software scheme that could record with accuracy specific events occurring, was easy to connect to a piece of equipment, didn't require lots of fiddly signal hooks, and the software trigger was minimal enough not to intrude very much in the timing of the kernel.

4.2. Generating the Triggers.

The next problem was how to manage the event triggers i.e how to automatically generate them in the target code, and how to manage the event value so that it could relate back to functions and points within functions. It seemed natural to place a trigger at the entry and exit of each function; that way code paths could be traced, and accumulated times calculated for each subroutine. It isn't really practicable to modify the source code to explicitly add the triggers; this would mean that a macro would have to be used so that the profiling could be turned off, and it would also mean manual allocation of a trigger value to each function, something that is tedious and error prone to manage. Besides, many functions have separate exit points, and often functions contain some initialization as part of their local variable declarations which would be performed before the trigger; this would give inaccurate timing results.

So it was decided to modify the compiler to add the trigger points; the Free Software Foundation's GNU C compiler was modified to generate the triggers at the start and end of every function. For ease of processing and identification, each function is assigned a trigger value that is an even number, and that number + 1 is used as the function exit trigger. The trigger value is taken from a file containing the function names and values. The profiling was enabled via a compiler option thus:

```
gcc -O -c -mprofile=name_file x.c
```

The option argument *name_file* contains function names and corresponding trigger values. If functions are compiled that do not appear in the file, they are allocated the next sequential value and the name appended to the file. The user can start with an empty file and allow the compiler to generate all trigger values. The name file can also be manually edited to tune the profiling in various ways, as described below.

A segment of a sample name file is show below:

```
duartint/100
splclock/106
splimp/110
```

```
        softint/112
        vec0/116
        sw_sys/200$
        sw_task/202*
        MGET/222=
        MCLGET/224=
        MCLFREE/226=
        MFREE/228=
        soo_rw/5038
        soo_ioctl/5040
        soo_select/5042
```

If a function's trigger value is 0 then no trigger will be generated for that function. Each trigger may have a single character modifier appended indicating whether the trigger deserves special treatment. An inline trigger is indicated as an equals sign; a dollar sign and an asterisk indicate that these functions are involved in context switching, and the flow of control is not obvious. These modifiers are necessary so that context switching is taken into account when analysing the data.

Inline triggers can be easily added to functions by including a profiling header file. The header file contains:

```
#ifdef     PROF
#define    PROFILE(x)    asm("tstb " #x);
#else
#define    PROFILE(x)
#endif
```

The inline triggers can then be added to code by adding the name into the name file, allocating a trigger value, and placing the macro with the trigger value as the argument into the desired location.

4.3. Analysing the data.

Once the triggers were generated in the object code, and the Profiler captured some events, the raw data is then uploaded to a Unix host. The data is processed by matching the event data (with the microsecond time values) with the function names as listed in the name file. Currently two different analyses can be generated; the first is a summary of each function's statistics. Preceding this is an overall summary of the profiling data. An example is shown:

```
Elapsed time = 1 sec 586839 us
Accumulated run time = 0 sec 815785 us (51.41%)
Idle time = 0 sec 771054 us (48.59%)
-----------------------------------------------
Elapsed  Net   # calls   (max/avg/min)  % real  % net     name
  3641   3432    1323         (16/2/2)   0.22%  0.42%   splnet
367833   3365     274       (27/12/11)   0.21%  0.41%   softint
   715    715     320          (3/2/2)   0.05%  0.09%   setsoftnet
 55571   3539     440          (9/8/5)   0.22%  0.43%   vec1
   100     86       5       (18/17/17)   0.01%  0.01%   m_get
    49     16       1       (16/16/16)   0.00%  0.00%   soaccept
721953  45293     267    (236/169/136)   2.85%  5.55%   soreceive
   151     34       1       (34/34/34)   0.00%  0.00%   soisconnected
   349     39       1       (39/39/39)   0.00%  0.00%   sonewconn
  ...
```

The elapsed time for each function is the accumulated interval time recorded between the function entry and exit. The net time is the accumulated time minus the accumulated time of all subroutines that are called from this function, giving an overall time for this function alone. The count of calls to each function is calculated, as well as the maximum, minimum and average time spent in each function. The net time is expressed as a percentage of the absolute elapsed time for the entire run (*% real*), and also as a percentage of the total time the processor was not sitting in the idle loop.

These statistics give accurate and concise summaries of the processor activity, and can quickly highlight bottlenecks or subroutines that are heavily used.

The second report shows a real time code path trace, along with accumulated and separate function timings. Subroutines are shown as nested where necessary to allow easy following of the code path; a sample is shown below:

```
0:760 522 -> softint (13 us, 30 total)
0:760 530     -> ipintr (14 us, 17 total)
0:760 535         -> splimp (3 us)
0:760 547     <-
0:760 552 <-
0:760 574 ---- Context switch in ----
0:760 588         <- sleep (4 us, 14 total)
0:760 596         -> selscan (399 us, 497 total)
0:760 631             -> soo_select (19 us, 27 total)
0:760 636                 -> splnet (2 us)
0:760 647                 -> sbselqueue (6 us)
0:760 658             <-
           ...
0:761 093         <-
0:761 119     <- select (34 us, 545 total)
0:761 151     -> accept (70 us, 256 total)
0:761 158         -> getsock (8 us, 13 total)
0:761 161             -> getf (5 us)
0:761 171         <-
0:761 175         -> splnet (2 us)
0:761 186         -> ufalloc (27 us)
0:761 217         -> falloc (32 us)
0:761 254         -> soqremque (13 us)
0:761 273         -> m_get (17 us, 20 total)
0:761 276         == MGET (3 us)
           ...
```

Accumulated and net elapsed times are shown for each function e.g the *accept* call is shown as taking 256 microseconds total elapsed time, but only 70 microseconds was actually spent in the *accept* routine; the other 186 microseconds was spent in subroutines called within *accept*. Inline triggers are marked using '=='. The modifiers in the names file allow detection of context switches, which are marked accordingly.

5. A Case Study.

How well does the Profiler operate in real world situations? The first experiment performed was aimed at evaluating the performance of the Berkeley TCP/IP code in conjunction with the Megadata embedded kernel. The hardware was a 68020 running at 20 MHz, using a separate Ethernet board over a slow/medium speed bus. The Ethernet board had local memory into which a AMD LANCE would transfer packets. A process was set up that listened on a TCP port, and when a connection was made a process was spawned that acted as a data sink. A Sun IPX was used to open a connection to the TCP port, and dump an arbitary amount of null data across the connection.

A summary of the profiling is shown below (most functions are not shown for brevities sake).

```
Elapsed  Net   # calls  (max/avg/min)  % real  % net     name
298926   2677     218       (26/12/11)   0.06%  0.34%   softint
257602   2745     339          (9/8/7)   0.06%  0.35%   vec1
  5365   3959     190       (25/20/19)   0.09%  0.51%   m_free
 27871  16303     251       (83/64/63)   0.35%  2.08%   m_pullup
512108  51347     285     (239/180/27)   1.12%  6.56%   soreceive
 17511   2318     302         (16/7/3)   0.05%  0.30%   sbwakeup
 22412   2080     302         (12/6/5)   0.05%  0.27%   sowakeup
 34851   3686     289        (21/12/8)   0.08%  0.47%   sbappend
 29078  15991     288       (90/55/28)   0.35%  2.04%   sbcompress
107598  86938     850     (277/102/27)   1.89% 11.10%   in_cksum
290913  21596     188     (294/114/13)   0.47%  2.76%   ipintr
 82042   7727     131       (72/58/57)   0.17%  0.99%   ip_output
218625  37394     294    (185/127/124)   0.81%  4.78%   tcp_input
157383  43991     579      (185/75/43)   0.96%  5.62%   tcp_output
143131   8107     290       (47/27/27)   0.18%  1.04%   tcp_usrreq
 23871  23785     921       (40/25/24)   0.52%  3.04%   clock
157250 113515     856     (369/132/33)   2.47% 14.50%   uiomove
 56805   7375     131       (71/56/54)   0.16%  0.94%   leoutput
 14207  13846     262       (128/52/8)   0.30%  1.77%   lestart
254857 249139     339    (3372/734/28)   5.41% 31.82%   lanisr
```

As expected, routines which traverse the incoming data take up the bulk of the time (e.g *in_cksum*, *uiomove*). A big surprise was the time of the Ethernet driver's interrupt handler. Some of this could be explained as slow memory requests over the external bus, but a glance at the maximum and average times showed that at least one interrupt took 3.3 milliseconds to process! The next step was to examine the code path trace and actually see what was going on. An inline trigger was placed at the start of processing for each received ethernet packet. The code path seen was thus:

```
1:169 991 -> vec1 (8 us, 2183 total)
1:169 996     -> lanisr (2153 us, 2175 total)
1:170 014     == le_rxpkt (18 us)
1:170 024     == MGET (28 us)
1:170 026         -> splimp (3 us)
1:170 043     == MCLGET (47 us)
1:170 045         -> splimp (3 us)
1:170 996     == MGET (1000 us)
1:170 998         -> splimp (3 us)
1:171 104         -> setsoftnet (2 us)
1:171 128     == le_rxpkt (1132 us)
1:171 138     == MGET (1142 us)
1:171 140         -> splimp (3 us)
1:171 157     == MCLGET (1161 us)
1:171 159         -> splimp (3 us)
1:172 037     == MGET (2041 us)
1:172 039         -> splimp (3 us)
1:172 145         -> setsoftnet (2 us)
1:172 171     <-
1:172 174 <-
```

The inline triggers showed that back-to-back packets were being received, and each packet was being transferred into an mbuf cluster and an ordinary mbuf. The time taken was in the copying of the packet from the local board memory to the internal mbufs. From this a maximum bandwidth could be calculated. This was considered unacceptable, and some options were examined that might remove this bottleneck. It was seen also that after the packet was copied into the mbufs, the data would eventually be copied out from the mbuf space into application buffers via read, so a good solution would attempt to minimise the amount of data copying.

One scheme attempted to optimise the copy from the ethernet board via loop unrolling, which would be usually an acceptable solution, but again the Profiler showed it made little difference to actual time. So some design modifications were made to the mbuf handling so that a a ‘foreign’ cluster buffer could be accomodated (Sun has a similar scheme); received packets are stored in 1518 byte buffers in the ethernet controller’s local memory, and these buffers are then ‘loaned’ to the network code. The network code then processes these buffers as it would any other mbuf cluster, and when the mbuf cluster is freed, it calls a handler that returns the buffer back to the owner. Thus the only copying should be the transfer from the ethernet board’s buffer to the application process (an unavoidable copy).

Would this improve things? If so, by how much? Previously it would be very difficult to answer these questions, but not only can these questions be answered using the Profiler, but it can answer them to two decimal places.

The new summary looked thus:

```
Elapsed  Net   # calls  (max/avg/min)  % real  % net     name
367833   3365     274      (27/12/11)   0.21%  0.41%   softint
 55571   3539     440         (9/8/5)   0.22%  0.43%   vec1
 30608  28609     307     (118/93/86)   1.80%  3.51%   m_pullup
721953  45293     267    (236/169/136)   2.85%  5.55%   soreceive
 13082   3913     304       (19/12/8)   0.25%  0.48%   sbappend
  9023   8877     304      (33/29/28)   0.56%  1.09%   sbcompress
230786 207568     880     (993/235/27)  13.08% 25.44%   in_cksum
360987  23587     267     (270/88/13)   1.49%  2.89%   ipintr
296221  38807     307    (181/126/122)   2.45%  4.76%   tcp_input
 96127  44059     575     (185/76/43)   2.78%  5.40%   tcp_output
  8223   8197     318      (46/25/25)   0.52%  1.00%   clock
504229 245434     752    (1071/326/33)  15.47% 30.09%   uiomove
 32791   7309     133      (57/54/53)   0.46%  0.90%   leoutput
 14348  14001     266      (126/52/9)   0.88%  1.72%   lestart
 52032  34538     440     (235/78/26)   2.18%  4.23%   lanisr
```

Some interesting effects can be seen; the amount of time taken by the ethernet interrupt handler is reduced to an average of 78 microseconds, but the average time spent in *in_cksum* and *uiomove* has more than doubled! Overall there is a major improvement because of the elimination of one set of data copying, but because the TCP data packet is now accessed over a slower bus there is a noticeable degradation in computing the packet’s checksum and moving it to the application buffer. This raises the difficult question of whether it would be more economical to copy the data once from the slower bus memory to the fast internal mbuf memory, or take the extra delays in bus access.

The major bottleneck is now clearly seen to be the external bus acesses, and design decisions may be made to increase the bus speed now that there is clear evidence of poor performance.

The important lesson here is that **quantitative** measurement allows these optimisation and designs issues to be tested rigorously, and accurate comparisons made, which just goes to prove:

Measure BEFORE you optimise.

6. Further Applications and Conclusions.

The next (and hopefully much more fruitful) exercise is to apply the analysis to the OSI stack incorporated in the recent Second Berkeley Networking Release, which is the basis of BSD 4.4 (yet to be released). Due to the untuned nature of this code, and to the lack of long term experience with complete OSI implementations, it is expected that the Profiler will be a valuable tool in providing detailed information about how to optimise the code paths within the networking software.

Another goal is to connect the Profiler to a PC running the freely available 386BSD software, so that measurement may take place of a *real* UNIX kernel, and profiling of the various subsystems of the kernel may take place. This will provide interesting data on the performance of virtual memory systems, file system support, and sophisticated networking applications such as NFS etc.

It is hoped that a case study of this kernel can be presented at the AUUG Winter Conference.