Hardware Profiling of Kernels

Andrew McRae – Megadata Pty Ltd.

ABSTRACT

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 with cheap and readily available hardware. Other profiling methods are touched upon, and why these methods were rejected. Some goals are stated, and a proposed hardware/software solution is described. In a case study, a 386BSD kernel is evaluated, and the results of this exercise are presented, demonstrating how tracing of software in real time highlights optimal or non-optimal code paths. The solution also provides for effective and flexible kernel debugging.

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

Introduction

Michael Jackson 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 learn an important lesson:

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. However, with a poor design, the best optimising compilers are usually of little benefit. Experience has shown that if a piece of software is not performing, reviewing the design is the best, and sometimes only, way of obtaining significant improvement. Sometimes performance is not a major goal of software; other issues such as maintainability, correctness under all conditions, and robustness are more important. Other times it is important that a piece of software not only runs correctly, but runs fast as well. The ideal is to have the best design, and then apply optimisation so that the implementation can perform well.

It is a common mistake to expend effort optimising code that intuitively seems to be slow, but contributes only a small portion to the overall total, and not optimising where most of the time is spent. This is oft referred to as the *10/90 rule*, where if a piece of software was improved in speed by 10%, and that software contributed only 10% of the running cost, an overall gain of only 1% is obtained; if the 10% improvement were applied to software that was 90% of the running cost, then a 9% overall gain is gained.

The key to optimisation is to understand where or how it is applied (or whether it should be applied at all), and therefore the Golden Rule of Optimisation is:

Measure BEFORE you optimise.

Optimisation in UNIX

UNIX has a number of tools to help in this area; compiler profiling allows time based and function entry/exit profiling to be incorporated into programs, which allow operating statistics to be extracted and analysed. Generally this is sufficient for most programs, as the programs are not usually interacting with, or affected by, real world events. Simulators also have been used to good effect by providing a higher degree of granularity to profiling, allowing tracing of code paths etc.

Kernel Profiling

Kernels are a special case in that they must interface to real world entities, such as devices, networks, memories, clocks etc. Subtle and complex interactions occur between device drivers, processes and external events, as anyone who has attempted to remedy bugs caused by these interactions will appreciate. It is likewise difficult to obtain hard data to guide kernel optimisation, mainly due to the difficulty in obtaining fine-grained kernel performance measurements.

Kernel measurement has been considered a Black Art in the past. A number of techniques have been devised that allow various degrees of accuracy. Virtually all kernels keep event statistics and counters that allow a rough idea of the overall performance; these counters can be reset or logged at specific intervals to give a broad understanding of system activity. Examples include paging rates, network packet inputs/outputs, disc block transfers etc. The main drawback to relying on event statistics is the poor granularity and lack of detail concerning where the kernel time is spent. Keeping a large

number of statistics also takes up memory, and sometimes requires a not insignificant amount of CPU time to update them.

A more common approach is to measure the overall system performance by using an external benchmark package, or by timing the throughput or response time of the kernel by running specialist programs, e.g., *ttcp* (networking), *iozone* (file system) etc. Others run a sample of the intended applications so that a true idea is obtained of the system performance in that environment. Whilst these are the ultimate in kernel measurement (by definition), they do not aid in discovering where optimisation should be employed, except perhaps in a general sense ('the network code needs to be faster...'. 'But where in the network code?').

Some areas of kernels can be measured in the same way as user programs, using function counting and gross clock profiling. If a psuedo-random or skewed clock is available, then it is possible to improve the clock profiling so that other clock-related activity is not missed. These measurements are useful but suffer from a trade-off in granularity and accuracy; the finer the granularity, the more time is spent running the profiling clock and not actually running the kernel, which may perturb the kernel's activity. The coarser the granularity, the less effect on the kernel activity, but then the resolution becomes too low to perform useful measurement. Memory also has to be reserved to store the profiling clock data, and having clock profiling running often may cause instruction and data cacheing to be adversely affected (though with larger caches becoming more common this may not be significant).

But 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 like 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 likelihood 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, normally out of the league of the casual kernel hacker. 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. Microprocessor designers are becoming aware of the need to measure and trace processor activity even when running in cache, and newer designs often have pins dedicated to providing indication of the state of the processor.

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 alter the timing characteristics.

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 allowed external responses to be measured, and certainly helped when hardware drivers were being tested.

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 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. It also is not trivial to connect and operate a logic analyser for most software engineers. Special logic analyser software can be used to perform time based profiling, but the sampling granularity was generally too coarse to be of any use, and since it operated on physical addresses, this was difficult to relate back to the actual software.

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, and then integrate the information with the source code.

I still had a desire to find out what was really happening *inside* these kernels, 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 attempted several methods of getting the data, with limited success, but in the meantime 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 performing 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 sections 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 cacheing enabled (as any 'production' code would run the cache enabled).
- Granularity to a source code function level (however short the function is) should be the worst case; however it would be desirable to profile within functions if possible.
- Any method of profiling should be portable between different computer architectures.

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. Cheap enough that any person who wished to profile and debug their home PC would be able to put it together, but useful enough so that design decisions could be made in confidence as a result of accurate measurement. It attempts to meet the above goals, and also be simple and cheap enough to build without great effort (even a software engineer could probably manage it).

The Profiler

Three basic elements 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 back-trace of events and relate it to the source code.

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 is purposely 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 one or two hundred dollars than the Profiler could be built by just about anybody.

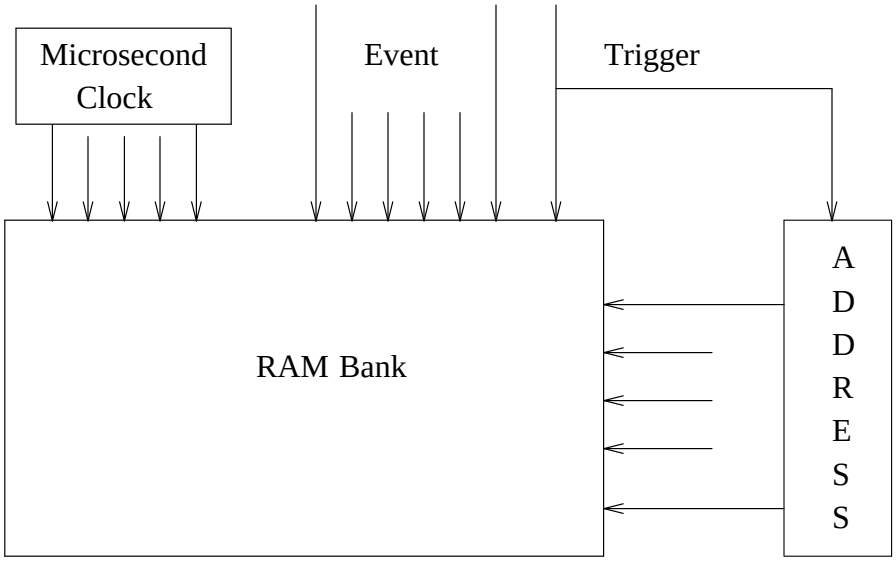

Figure 1: Profiler block diagram

Finally the Profiler is simple because I hate wire wrapping; it's so much more tedious than writing software.

Commonly available components were used, and the hardware prototyped on a breadboard using wire wrapping. A single electrically erasable PAL is used for the logic and timing functions; the final cost of the parts totaled 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 meant that moving to equipment that used different methods of accessing the Profiler could be handled by different PAL equations. It also allowed extra facilities to be incorporated such as some display LEDs and control switches.

A block diagram appears in Figure 1. The Profiler consists of a block of RAM which is 40 bits wide, an incrementing address counter, a free running counter clocking at 1 Megahertz, and some control logic. The RAM is split into two sections, one holding an identification code (**event tag**) which is 16 bits in width, and the other 24 bit wide section connected to the microsecond clock. When an event tag is presented to the Profiler, it stores this code along with the microsecond counter value into RAM. The RAM address is automatically incremented every time an event is stored, essentially storing the event and time in a large list. The list is currently 16384 events long, but there is no inherent limit to the total number of events stored except the maximum amount of memory designed into the Profiler.

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 Simple Matter Of Hardware), but how to generate the event code, which must come from the equipment being measured. It was clear that some software assist was required to generate these event tags in an orderly fashion. Another problem was how to connect the Profiler to a working system.

An elegant solution presented itself when I realised that most of the computing equipment that the Profiler was designed for has one or more EPROM sockets fitted for boot code or board drivers. This presented itself as a simple method of connecting the Profiler to the equipment, by piggybacking a EPROM socket onto some cable, and using the socket to bring the appropriate signals into the Profiler. The original boot EPROM would plug into the piggy-back socket, if indeed it was required. The event trigger would be the access of the EPROM, and the address of the EPROM access could be the event tag data.

In this case, only 18 signal lines needed to be brought into the Profiler (16 address lines and the EPROM ChipEnable and OutputEnable signals). This allowed a simple and easy method for 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 switch exists on the Profiler that initiates the profiling recording; this allows the Profiler to be synchronised with execution of test programs, network activity etc. Two LEDs exists in the card giving some indication of its state; the first indicates that the Profiler is active and storing data, the second indicates that the address counter has overflowed and the Profiler has automatically ceased storing data.

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. How does the data then get retrieved? The data RAMs are mounted via battery-backed SmartSockets™, and when the profiling samples are stored, the timing data is retrieved by transferring the RAMs into another networked embedded host, and copying the profile data to a UNIX host for processing.

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 a lot of signal hooks, and the software trigger was minimal enough not to intrude very much in the timing of the kernel.

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 generate 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; in this manner 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. Besides, many functions have multiple exit points, and often functions contain some initialisation as part of their local variable declarations which would be performed before the trigger; this would give skewed 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. On a 68000 system, this effectively added one instruction in the function prologue, and one instruction in the function epilogue, e.g., a function would now contain:

```
.globl _myfunction
_myfunction:
    tstb 1386
    link a6,#-8
       ...
    unlk a6
    tstb 1387
    rts
```

If a higher granularity of profiling is required with a function, then a macro may be used to generate an *inline* trigger via a compiler *asm* function. Assembler routines may have event tag trigger instructions added via an include file and a preprocessor macro.

The trigger value is taken from a file containing the function names and values, of which a sample is shown below:

```
main/502
hardclock/510
gatherstats/512
softclock/514
timeout/516
untimeout/518
swtch/600!
MGET/1002=
```

The insertion of Profiler event tag instructions is enabled by a compiler option indicating the name of the file containing the functions names and event tag values. This file is automatically extended by the compiler when it generates new event tags for functions that do not already exist in the file; the event tag for the added functions is taken as the next available value (i.e the next value higher than the current highest in the file). The name/event tag file may be generated from scratch, with an initial dummy entry indicating the starting tag number to use. Once generated, the same profile tags are used to allow recompilation without having different profile tags assigned to a function. Multiple name/tag files may exist, and may be concatenated to provide a complete list of profiled functions. Inline and assembler trigger names and values may be manually added to the file.

Special character modifiers may be appended to any of the name/tag values that indicate special processing of this particular tag when analysing the results; a '!' character indicates a function that causes a processor context switch, which the analysing software must treat specially. The '=' modifier indicates an inline tag, as opposed to a tag representing the entry or exit of a function.

Adding event tag triggers to software will have a small impact on performance; this has been calculated at around 1 to 1.2% extra CPU cycles, which is a small penalty to pay for profiling. In absolute terms this equates to about 400 nanoseconds per function for a 40 MHz 386. The size of the software also increases by the overhead of two instructions per function; it is hard to quantify this increase as a percentage as it depends on the number and size of each function, and also on the number of inline triggers used.

Connecting to a PC

The initial platform for testing was a 68020 board designed for embedded applications. Since it was of Megadata design and manufacture, it was

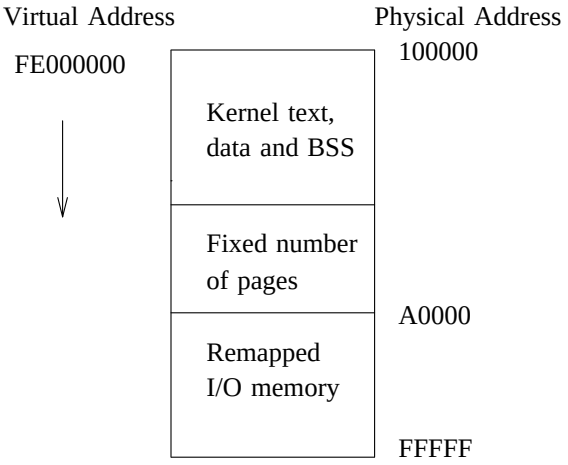

Figure 2: Virtual memory remapping

easy and safe to develop and test the Profiler hardware on this platform.

Once the concept was tried and proven, it was decided to connect the Profiler to a *real* kernel, namely the freely available 386BSD release 0.1 running on a 40 Megahertz 386 PC with 8 Megabytes of memory. Since the point of interface was a common JEDEC EPROM socket, it was simple to connect the Profiler to the PC via a spare EPROM socket on a Western Digital WD8003E Ethernet controller. Any ROM socket could have been used as long as it was at a known fixed address and was accessed as a 8 bit wide device, such a VGA BIOS ROM socket etc. The address space of the ROM falls somewhere in the ISA bus memory address space, between (hex) A0000 and 100000.

Changes were made to the 386BSD C compiler (based on gcc 1.39) to accommodate the Profiler event tag additions. A snag was hit when it was realised that the 386BSD kernel remapped the kernel's view of ISA bus memory into kernel virtual address space, and so an absolute address could not be easily used.

After initial loading, the 386BSD kernel remaps the physical memory addressing to new virtual locations as shown in Figure 2.

In effect, the kernel is remapped to absolute location FE000000; the last location of the kernel is rounded to a page boundary, and a fixed number of pages are allocated for the kernel stack, a proto u-dot area and other virtual memory requirements. The ISA memory address space is then remapped to follow this kernel address space; the virtual address that this memory is mapped at may vary depending on the size of the kernel.

The Profiler event tag instructions added by the compiler require an absolute address within the EPROM address range starting at a fixed EPROM location somewhere in the ISA bus memory address space. But since this EPROM location may vary depending on the kernel size, it cannot be resolved at compile time. It would be unreasonable to have to recompile all source code modules just to update the event tag instructions. Fortunately it can be resolved at link time with a little extra effort; the compiler modifications generate function entry and exit event tag instructions thus:

```
.globl _myfunction
_myfunction:
    movb _ProfileBase+1386,%al
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    ...
    leave
    movb _ProfileBase+1387,%cl
    ret
```

The global label *_ProfileBase* is set in an assembler file as a result of a two stage kernel linking process. The kernel is first linked with a dummy of _ProfileBase, then a shell script is automatically used to extract the size from the kernel and recompile the assembler file with the real value of _ProfileBase, which is then linked with the kernel. If the physical address of the Profiler EPROM location is changed, then only this assembler file has to be modified to cater for the new position of the EPROM. This scheme worked very well in providing a correct run time virtual address of the Profiler's physical memory address.

Profiling the Kernel

A total of 16384 event tags and time values may be stored in the Profiler before the RAM addressing overflows. Whilst this allows a considerable amount of data to be gathered, if a particular subsection of the kernel was to be examined in finer detail, then some form of selective profiling should take place. This is easy to set up, as all that needed to take place was to compile those modules of interest with profiling enabled, and to compile the rest of the kernel without profiling. This allowed highly selective profiling to take place without losing resolution, but without filling the Profiler RAM with events in which there was no interest.

This selective profiling allowed two broad categories of profiling to take place, macro-profiling and micro-profiling. Macro-profiling takes place when certain key modules such as the system call handlers and VNODE interface routines are profiled. Virtually all kernel code paths traverse these higher level routines, so it is possible to get a broad-brush view of system performance to answer questions like, "How long does it take to fork/exec a process?" Or "How long does it take to read this file?" Or "How long does it take to open a TCP connection?" This view of the kernel is very instructive as the overall code path through the kernel can be easily seen and traced, and can give a guide to where further profiling should take place.

Micro-profiling takes place when a particular subset of the kernel is examined in detail. Interrupt handlers, clock routines, assembler subroutines can be profiled as well, allowing complete snapshots to be taken of a particular kernel code path. For example, the file system buffer cache, file system code and disk driver routines can be profiled, so that whenever the kernel enters these areas, the code path is traced. No other code paths are profiled, allowing a detailed and unobstructed view of that section. Similar subgroupings may be made with the networking code, the Network File System (NFS) code, the virtual memory subsystem, various drivers (SCSI, tty, IDE) etc.

After repeated micro-profiling of the various kernel subsystems, it is possible to eventually

construct a highly detailed and accurate mosaic of the kernel performance. As a result, quantitative comparison may guide design and implementation improvements as performance bottlenecks are highlighted in the kernel, and accurate before and after measurements may be made to test the success of such changes.

Analysing the data

Once the triggers are generated in the object code, and the Profiler has 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. The raw data appears as a list of event tags and times. How then is the data processed to gain the maximum useful information out of it?

Identification of function entry and exit points allow a code path trace to be constructed with timing information at each call and return point. Subroutine depth is easily discovered by matching exits with entries, with event tags between a function's entry and exit indicating subroutine calls within that function.

This works well when used when the control flow follows a simple subroutine call/return model, but when the target being profiled is a kernel this model is inadequate to describe the thread of control. The essential difference is that the kernel is multiplexing many processes, and context switches occur to change the control flow to a different process. This appears in the profiling data as a discontinuous change in the subroutine call/return model, where it appears a different subroutine is being exited than was called. Some extra information must be given to the analysing software to indicate where context switches may occur.

386BSD context switches occur in the *swtch()* function; upon entry to *swtch* the current process context is saved, and the run queue is checked for the next process to run. If none are ready, then an idle loop is entered.

The analysis software must detect when *swtch* is entered so that each process's code path may be analysed separately. The *swtch* function is tagged in the name file with a modifier to indicate this special processing. The time between the exit of a call to *swtch* and the entry to the next call of *swtch* is analysed as a contiguous block of processor activity. The time in *swtch* itself is counted as CPU idle time, except when device interrupts occur. The separation of idle and active CPU time provides accurate calculation of CPU usage, both as a overall ratio and on a per function basis.

Currently two different analyses can be generated; the first is a summary of each function's statistics, sorted by highest to lowest net CPU usage, headed by an overall summary of the profiling data, see Figure 3.

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 (*% net*).

These statistics give accurate and concise summaries of the processor activity, and can quickly highlight bottlenecks or subroutines that are heavily used. As can be seen in the example, it is obvious

```
Elapsed time = 0 sec 497272 us (28060 tags)
Accumulated run time = 0 sec 492248 us (98.99%)
Idle time = 0 sec 5024 us ( 1.01%)
-----------------------------------------------
  Elapsed  Net   # calls  (max/avg/min)  % real  % net
  166218 165343     889     (1089/185/2)  33.25% 33.59%   bcopy
  152382 151700     514     (901/295/23)  30.51% 30.82%   in_cksum
   26359  26359    2474        (13/10/8)   5.30%  5.35%   splnet
  442031  16391     166      (125/98/87)   3.30%  3.33%   soreceive
    9963   9913    2782         (19/3/3)   1.99%  2.01%   splx
   16069   9855     433       (36/22/18)   1.98%  2.00%   malloc
  202651   9132      86     (193/106/28)   1.84%  1.86%   werint
  183830   7989     170       (98/46/18)   1.61%  1.62%   weget
   13646   7576     423       (23/17/15)   1.52%  1.54%   free
   19467   7189     218       (78/32/12)   1.45%  1.46%   westart
   ...
```

Figure 3: Summary of profiling data

immediately that the CPU is completely saturated, and most of its time is spent in *bcopy*.

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 in Figure 4.

Accumulated and net elapsed times are shown for each function, e.g., the *tcp_input* function takes 318 microseconds total elapsed time, but only 92 microseconds was actually spent in the *tcp_input* routine; the other 226 microseconds were spent in subroutines called from within *tcp_input*. Inline triggers are marked using '==>'. Modifiers in the names file allow detection of context switches, which are flagged in the code path trace.

From the function summary report *bcopy* is a likely target for more investigation; each invocation of bcopy can be examined by looking at the code path trace, and some idea can be obtained why this function is causing high CPU usage.

Much of the effort going into the Profiler now centres upon processing the raw data in many more useful ways, such as graphically representing the code path or building histograms of the function time and usage for easy detection of bottlenecks.

User Code Profiling

The hardware profiling solution can be readily adopted to user level profiling with similar results. A driver stub may be configured in the kernel that reserves the Profiler's physical memory address space; a modified profiling *crt.o* initialises the process for profiling by opening the driver and calling *mmap* to memory map the Profiler's address space into a fixed location within the process address space.

There is no reason why a mixture of kernel and user level profiling cannot take place concurrently, or profiling several user processes at the same time to closely monitor and analyse interactions occurring via the interprocess communications facilities. This approach is especially applicable in debugging and tuning communication protocol stacks, where the network and link layers are implemented in the kernel, and the transport layer and higher layers are implemented in user libraries and application code.

Case Studies

The first platform that the profiler was tried on was a 68020 based embedded system running a Megadata kernel incorporating the 4.3 BSD Tahoe release networking code. A number of profiling

```
0:002 671 -> ISAINTR (31 us, 778 total)
0:002 679     -> weintr (50 us, 292 total)
0:002 704         -> werint (70 us, 215 total)
0:002 739             -> weread (11 us, 145 total)
            ...
0:003 458                     -> bcopy (1073 us)
            ...
0:004 996     -> ipintr (55 us, 424 total)
0:004 998         -> splnet (10 us)
0:005 012         -> splx (4 us)
0:005 031         -> in_cksum (23 us)
0:005 074         -> tcp_input (92 us, 318 total)
0:005 082             -> in_cksum (38 us)
0:005 138             -> in_pcblookup (9 us)
            ...
0:005 424     -> spl0 (21 us)
0:005 449 <-
             ---- Context switch in ----
0:005 488                 <- swtch
0:005 492                 -> splx (3 us)
0:005 513             <- tsleep (22 us, 25 total)
0:005 520             -> falloc (22 us, 83 total)
0:005 523                 -> fdalloc (13 us, 18 total)
0:005 528                     -> min (5 us)
0:005 541                 <-
0:005 547                 -> malloc (29 us, 43 total)
            ...
```

Figure 4: Code path traces

studies helped greatly in identifying key performance problem areas in the kernel, and in one case the recoding of an Ethernet driver doubled the network throughput.

A SNMP client based on the CMU SNMP code was profiled, highlighting a major bottleneck in searching the MIB table linearly; redesigning the data structure to use a B-tree to hold the MIB data reduced the CPU cycles required to respond to SNMP requests by an order of magnitude.

Since the embedded system contained no memory management hardware, no user/kernel boundary existed except as an artifice of the system interface, thus it was easy to trace activity right from application level code down to the kernel code through to driver code.

The next step was to begin profiling the 386BSD kernel, which provided much more comprehensive and interesting results. These results are presented in several sections, the first being an overall impression of performance, and the other sections taking one kernel subsystem and describing the results of profiling each one in turn.

386BSD Overall Performance

The profiled kernel contains 1392 functions, so 2784 event tag trigger points were automatically added to the code. 35 assembler routines had trigger points added, so a total of 1427 possible functions could be profiled. Depending on the nature of kernel activity, the Profiler RAM could be filled (a total of 16384 events) in as short a time as 300 milliseconds. No noticeable difference can be detected between a profiled and a non-profiled kernel. After profiling a number of the key areas of the kernel, some impressions emerged concerning the kernel performance. These fall into three main categories; CPU performance, I/O performance and virtual memory management.

Firstly, I was pleasantly surprised to note the oft maligned Intel architecture did indeed run fast, especially at a clock speed at 40 Megahertz and employing 64 KB of external cache. Moving data through the kernel to user space was faster than expected, and it was clear that function call and return was also speedy. It would be instructive to profile other microprocessor types running at a similar speed using the same software to do a side-by-side comparison. Undoubtedly memory speed and cache effects have a major impact on performance, as data throughput dropped markedly whenever memory was accessed on the ISA bus as opposed to main memory. More on this later. Profiling the interrupt code showed that the regular clock tick interrupt took on average 94 microseconds to execute; unfortunately the hardware architecture does not provide for Asynchronous System Traps (commonly known as software interrupts), so the interrupt code has to work extra hard to emulate this facility. The interrupt code overhead to do this is around 24 microseconds per interrupt; it is hard to judge whether this has a significant impact on system performance.

Due to the interrupt architecture of the bus and the processor, it was evident that more time was spent ensuring correct synchronisation and interrupt lockouts than would normally be required on a multi-priority interrupt level processor such as 680x0; on the average it took 11 microseconds per *splnet* call, which may not seem a long time, but the *spl** routines get called a great deal, and it all adds up to a significant amount. In one test, 9% of the total CPU time was spent in *splnet, splx, splhigh* and *spl0*. Unfortunately it is hard to see how this could be improved, given the nature of the interrupt architecture.

Some sample functions are shown in Table 1, along with their measured average execution times (the times are inclusive of subroutines that are called).

Function	Microseconds
vm_fault	410
kmem_alloc	801
malloc	37
free	32
splnet	11
spl0	25
copyinstr	170

Table 1: Sample function timings

When some tests were performed where input/output activity was heavy, it was clear that a major bottleneck in system performance is the use of the ISA bus. This was especially noticeable on the Ethernet adaptor, which is a 8 bit wide controller. To transfer similar amounts of data, the ISA bus is up to 20 times slower than main memory transfers.

It would be instructive to profile different controller cards to determine where each performed best; when support for EISA cards is available it would be interesting to see what performance gain would be obtained using the higher bandwidth bus.

Whilst the CPU performs reasonably well, overall performance is crippled by the poor I/O bandwidth, and the interrupt architecture of the 386 and the ISA bus also contributes to reduced performance.

The virtual memory management subsystem of 386BSD was derived from the Mach memory management code; a member of the CRSG has been heard to say that the old BSD VM code was ripped from the kernel, and the Mach memory management code placed next to the kernel and hot glue poured down the middle. Following code path traces of various virtual memory functions seem to support this

model, and it seems the glue is fairly thick in some places and thin in others. Some functions seem to run surprisingly fast; the routine that handles page faults and enables new pages to be accessed (*vm_fault*) takes about 400 microseconds, which seems reasonably low overhead. On the other hand, an excessive number of page faults seem to occur at times. Where the real performance problems lie is in creating new VM contexts for new processes, as explained in the next section.

Fork/exec Profiling

A common operation of UNIX is to *fork* a process and create a child copy of the process, which then *execs* a new process image. For UNIX to perform well, these two operations must be reasonably fast, since some UNIX operations rely on a low cost of process creation.

The current situation looks fairly abysmal; it takes some 24 milliseconds to perform a *vfork* operation, and it takes about 28 milliseconds to perform an *execve* system call. This adds to about 52 milliseconds to perform a combined fork/exec operation. Note that these times do not include any disk activity, as the process image was already cached. Where is this time being used? In figure 5 a summary of the highest cost subroutines is shown.

Most of the CPU time occurs within a small number of routines; it is clear that the *pmap* module is a bottleneck when manipulation of the virtual memory is required (the *bcopyb* call relates to scrolling of the console screen, so it should be ignored for the purpose of the exercise). Over 50% of the time is being spent in the virtual memory routines shown above. Examination of the code path trace shows that *pmap_pte* is called 1053 times when a *fork* is executed, and a similar amount when an *exec* is done. Further analysis of the code path shows the exact progress of the fork operation, and each subsection can be examined in detail to see the amount of time it is taking, and whether significant optimisation can take place. There is a major amount of cross-calling between the *pmap* module, and the rest of the virtual memory subsystem, so it is envisaged that a major performance benefit would occur if some of that glue could be trimmed back and some sculpting of the interface performed.

Network Performance

Profiling was performed on the TCP/IP and socket code by running a program that listened on a socket and when another host connected, read and discard the data. A Sun Sparcstation 2 was used as the host to send the data, as I was sure it could fill the available network bandwidth to the PC over an ethernet.

This was the only test that caused the PC to be totally CPU bound, so that essentially the CPU was busy 100% of the time. It was obvious that the PC could not process the data from the network at anywhere near Ethernet speed. Examining the code path trace and function summary showed that 33.6% of the time was spent in *bcopy*, and that 30.8% of the time was spent in *in_cksum*. Again, *splnet*, *splx* and *spl0* contributed around 9% of the time.

Delving further into the code path trace, it was clear that a major bottleneck occurs because the Ethernet driver for the card must copy that data from the onboard controller memory across the bus; each TCP data packet that was received (i.e a full Ethernet packet) took about 1045 microseconds to process at the driver level. This alone is only about 20% more data throughput than Ethernet itself, so it is unlikely that Ethernet data rates through to the network applications can be achieved using this 8 bit controller card, unless the rest of the software has been tuned for minimum overhead. One approach to solve this copying overhead is to make the buffers on the controller memory external *mbuf* memory, so that all the driver has to do is link the received packet(s) to *mbuf* headers, and then the double copying would be avoided (once from the controller

Elapsed	Net	# calls	(max/avg/min)	% real	% net	name
77603	58913	67	(14061/879/2)	5.02%	28.22%	pmap_remove
22283	22148	5549	(66/3/2)	1.89%	10.61%	pmap_pte
12938	12938	1215	(13/10/9)	1.10%	6.20%	splnet
10912	10874	3	(3634/3624/3613)	0.93%	5.21%	bcopyb
33435	10134	453	(40/22/21)	0.86%	4.85%	spl0
15963	7876	8	(3862/984/3)	0.67%	3.77%	pmap_protect
5657	5657	77	(244/73/3)	0.48%	2.71%	bcopy
47723	4889	115	(64/42/27)	0.42%	2.34%	vm_fault
4759	4759	1349	(5/3/3)	0.41%	2.28%	splx
7836	4361	236	(29/18/13)	0.37%	2.09%	vm_page_lookup
7320	3489	119	(39/29/12)	0.30%	1.67%	pmap_enter
3457	3457	38	(132/90/2)	0.29%	1.66%	bzero

Figure 5: High cost subroutines

memory to mbufs, and then via *copyout* to the user data space).

Would this help? Contrary to intuition, this would actually decrease the performance, and using the accurate timing provided by the Profiler, a close estimate of the impact can be calculated. It takes *bcopy* around 1045 microseconds to copy 1500 bytes from the controller; *copyout* takes about 40 microseconds to copy a 1Kbyte mbuf cluster to the user data space. If the controller memory were accessed only once, then collapsing *bcopy* and *copyout* would give at most a gain of 60 microseconds (less than 6%). But other routines access the network packet as well, such as the TCP and IP input processing routines, and most importantly the IP checksum routine. Checksumming the packet whilst in the controller's memory would add at least an extra 980 microseconds to the overall processing of the packet. The time to process a packet would increase from 2000 microseconds to around 3000 microseconds, a big loss. It is now obvious that if you have slow controller memory, it is a big win to get it out of that memory as soon as possible into faster main store.

The other major CPU user was the checksum routine itself, which was almost a big an overhead as the driver packet copy. This was surprising at first, as the packet was now in main memory, and the checksumming should be close to memory-to-memory copying speeds. To checksum a 1 Kbyte packet was taking 843 microseconds. It was discovered that the *in_cksum* routine has not been optimally coded (e.g., like other architectures where it is done in assembler), and recoding this routine should provide a reduction in packet processing from 2000 microseconds to perhaps 1200 microseconds; this would provide a major improvement in network performance, and the limiting factor would become the memory bandwidth available to the network controller across the ISA bus.

Another conclusion that can drawn is that a much faster I/O architecture is required before serious data throughput can be expected, but I think we all knew that.

Filesystems

Separate profiling studies have been performed on the BSD Fast File System (FFS) code and the Network File System code. Due to the network performance problems discussed in the previous section, any performance issues in the actual NFS implementation are totally swamped by the I/O bandwidth limitations. An interesting situation arises due to the fact that UDP checksums are usually turned off with NFS; since the checksum routine contributed a large proportion to the CPU overhead, NFS actually provides less overhead and better throughput than an FTP style connection!

Given the tracing capabilities of the Profiler, it was easy to get accurate measurements of the network turn around time with NFS RPC calls, and to see how long to formulate the request, send it and then how long to process the reply.

The disc controller used in the target PC was an IDE controller on a Seagate ST3144 disc. The FFS profiling showed how disc seek times impact the I/O throughput. Each read of the disc varied from 18 milliseconds up to 26 milliseconds. Each write interrupt took about 200 microseconds in total, with about 149 microseconds of that being actual transfer time of the data to the controller. Interrupts seemed to be close together most of the time (< 100 microseconds), so the disc driver may well be improved by waiting a short time after transferring the data to see if the controller is ready to accept another block straight away.

Overall, the CPU was only busy for 28% of the time when doing a large number of writes, so the disc seek times are still the major influence in determining disc throughput. It was interesting to see that out of that 28%, at least 6% was spent in the *spl** routines. It would be interesting to use a different type of controller (maybe one with DMA) and see what difference it makes.

Conclusions and Future Work

The major conclusion about the performance of 386BSD is that there are a small number of areas that need addressing, that when fixed should improve the performance considerably. The hardest area to address is the virtual memory subsystem. The easiest area would the IP checksumming. The grossest area of mismatch between the hardware architecture and UNIX is the interrupt priority control and lack of software interrupts.

It was also clear that the hardware I/O performance is a major factor, and that the platform the profiling was performed on is crippled in I/O bandwidth.

Even in its simple prototype form, the Profiler has proved to be an invaluable tool for looking under the hood while the engine is running. One clumsy aspect that remains is the uploading of the Profiler data to a host for processing; currently this is manually performed, which slows down the profiling process somewhat. I am considering a new improved Profiler hardware design with more memory and some extra facilities. A higher clock precision has been considered, especially if the Profiler were connected to a upmarket workstation architecture such as a Sun or DEC; this would entail fitting a wider RAM module for accepting more clock data bits. It is unclear at this stage whether a higher clock rate is really needed, though.

The method of connection via EPROM socket has proved to be so useful that it is hard to see how

it could be improved. The next step is to bring in the EPROM data lines as well, and have a Zero Insertion Force socket for the EPROM on the Profiler itself. Then once the Profiler has been used to collect the data, each of the storage RAMs in turn can be multiplexed into the EPROM address space, and the data can be read as if it were an EPROM. This would allow fast turnaround for processing the Profiler data.

Since the raw data comes in a simple package, a lot of analysis can be applied to the raw data. Further work in this area hopefully will yield sophisticated tools that allow statistical processing of the data, groupings of functions into separate subsystems, and other ways to process the data.

Acknowledgements

Thanks must go to Bryan Grayson, who slaved with me over a hot logic analyser. Without his invaluable assistance the Profiler would still just be an idea.

Thanks also to Megadata, who suffer me being distracted from doing *real* work for long enough to try out new ideas.

Finally thanks must go to William Jolitz and the UCB Computer Science Research Group, who have brought dreams into reality for a lot of people who have always wanted to hack on kernels.

Author Information

Andrew McRae is a software engineer with the Australian company Megadata Pty Ltd, where for the last 9 years he has worked on real time supervisory and control systems. His responsibilities include communications and embedded systems, and a range of other areas such as Unix applications and drivers. He has been involved with the Australian Open System Users Group (AUUG) since 1986. Prior to joining Megadata he co-founded a company specialising in motion control special effects and computer graphics for film and television. He can be reached via electronic mail at andrew@megadata.mega.oz.au, or via snail mail at Megadata, PO Box 1687, Macquarie Centre, NSW 2113, Australia.