More Haste, Less Speed

David S. H. Rosenthal

Sun Microsystems
2550 Garcia Ave,
Mountain View, CA 94043

ABSTRACT

In the good old days, six years ago, I was teaching at the Universiteit van Amsterdam and sharing a 1MIPS, 4Meg machine with up to 60 other users. At times, it was irritatingly slow. Now, I have a 1.5MIPS, 4Meg machine all to myself, and at times it's irritatingly slow. Where did all the extra power go? Are we doomed always to be frustrated at the performance of our Unix systems? What can we do to improve the performance of the systems we use and the code we write?

''What we created
has separated
a past we hated
from what we are feeling today
'cause everything's different now.'' 'Til Tuesday *Everything's Different Now*

1. Introduction

Six years ago I was teaching Computer Science in the Universiteit van Amsterdam. The department had lots of students, and not much money, so we configured a low-cost, high capacity UNIX† timesharing system. It was a second-hand PDP11/70 with 4 Megabytes of memory, two Massbuss disk controllers with 5 SMD disks, and 64 lines of DH11 DMA asynchronous line multiplexers, running 2BSD.

The system proved very popular, and I frequently found myself sharing it with around 60 other users. I won't claim that the performance was great, in fact at times it was irritatingly slow, but I could get my work done at the cost of some frustration.

Now, I have a Sun 3/75 with 4 Megabytes of memory and a SCSI disk on my desk*. It has the same amount of memory, and a processor that is around 50% faster than the PDP11/70. I have it all to myself instead of sharing it with others. I won't claim that the performance is great, in fact it's irritatingly slow at times, but I can get my work done at the cost of some frustration.

A long time ago, C. Northcote Parkinson[5] observed a similar phenomenon and enunciated Parkinson's Law to describe it:

> *Work expands to fill the time available for its completion.*

It seems as if an analogue of this law operates for UNIX

> *The hardware is never powerful enough to keep the user happy.*

Other have made the same observation,[1] but they normally blame it on the growth of applications. I contend that there are non-linearities in the behaviour of UNIX systems and their users that interact together to make the statement true, reinforcing the effect of application growth. It isn't enough to adopt a reactionary attitude, and complain that networking, window systems, virtual memory and all the other things that have

---

† UNIX is a trademark of Bell Laboratories.

* The fact that this is hardly a state-of-the-art machine doesn't, alas, affect my conclusions. I have observed the same phenomena on much more powerful machines, they're just easier to see on my machine.

been added to UNIX since the golden age of V7 burn more CPU cycles than they're worth. If that were true, at some point the machines would be fast enough to afford all these goodies. I contend that there are fundamental mechanisms that mean that simply adding MIPS will *never* keep the user happy.

This is not a ''scientific'' paper. I have only fragmentary evidence for my hypothesis, and no clear idea of how to test it. I am simply presenting some of the ideas I'm spending some of my time thinking about.

2. An Experiment

Figure 1 shows the results of a simple performance experiment. The SunView user interface toolkit, and the underlying SunWindows window system, were modified to allow a script of user actions to be recorded and played back as fast as the system would accept input. The object was to simulate an interactive user limited by overall system performance. The time to complete the script was recorded in a series of runs; on each run the UNIX kernel's memory self-sizing routine was faked to return a different amount of system memory.

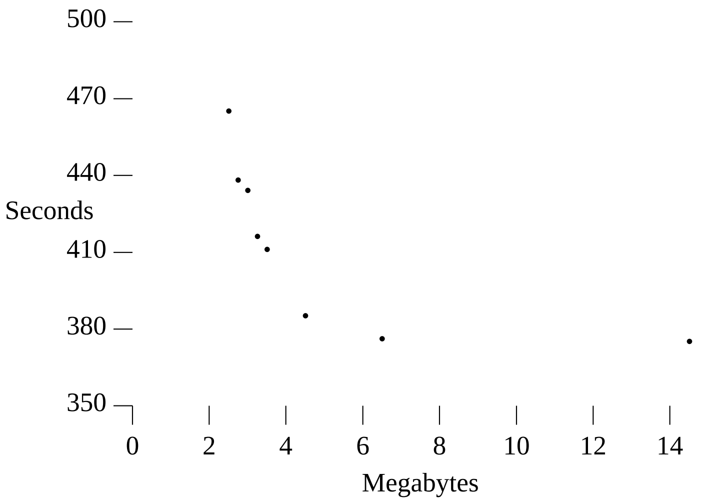

Figure 1: Task duration vs. available memory

The exact numbers aren't important, its the shape of the graph that is telling us something. What it is telling us is hardly unexpected. As you add memory to a system with very little memory, the time to complete the script falls very rapidly. But after a certain point is reached, adding more memory has little effect. In this case, the knee in the curve is around 4.5M of available memory.

3. What is Going On?

Like fluid flow and other dynamical systems, operating systems with multiple processes and virtual memory exhibit transitions between ordered and chaotic states.[2] As the system is driven harder and harder, response time gets gradually worse and paging gradually increases. At some point, response gets sharply worse and paging increases dramatically as the system starts *thrashing*. Thrashing is inevitable; introducing virtual memory provides the system with a feedback loop that necessarily allows chaotic behaviour and introducing multiple processes allows the user an easy way to modulate system load.

Another way of looking at this is that the system oscillates between a state in which it is saturating the CPU, and a state in which it is saturating the disk. The system is stable in the CPU-saturated state, and unstable in the disk-saturated state, in the sense that small changes in the load in the ordered state produce small changes in response time, whereas in the chaotic state small changes in load can produce large changes in response.

4. What Does It Mean?

Since it appears that the shape of this graph is a consequence of multiple processes and virtual memory, two of the reasons why we want to use UNIX rather than MS-DOS or the Mac ‘‘OS’’, spending some time thinking about what the shape implies looks like a good idea. So far, I have come up with the following implications:

- Is the goal *good* performance or *predictable* performance? They aren’t the same thing.
- The goal is obviously to keep the system operating in the ordered region. Adding memory helps but there may be other things just as good.
- Virtual memory is *not* just like real memory, only slower. We need VM-aware design for good performance.
- It is impossible to discuss the performance of a UNIX workstation without discussing how the user is driving it. To keep the system in the ordered region, we need to affect what the user does as well as the way the system responds.
- Simplistic benchmarks can do more harm than good.

4.1. Good Performance or Predictable Performance?

Comparisons between the performance of workstations and ‘‘personal computers’’ such as the IBM PC and the Mac are often unflattering to the workstations despite their much greater horsepower. The difference is largely due to the fact that PC operating systems have only a single process, and do nothing unless the user asks them to. The workstations, on the other hand, have multi-process systems and perform many activities in parallel autonomously.

As these processes compete against each other for resources such as CPU and memory, they reduce the *predictability* of the response of the workstation. On a PC, a given operation will always take exactly the same time. On my workstation, the time a given operation will take depends on whether, at the time I invoke it, someone else is sending me mail, or one of the file servers I use is heavily loaded, or there is a

```
make vmunix
```

underway in another window.

If the machine is thrashing, its response will be much less predictable. This makes the contrast with the PCs even more marked. Not merely is the system inherently less predictable, but the more you use it the less predictable it gets!

And, worse still, workstations typically use network facilities. Sharing network bandwidth and resources with other workstations adds to the unpredictability, and there’s little that can be done to reduce this.

Given a multi-process system, users use it to get more work done. I can still remember the amazing productivity boost I got when I switched from Control Data timesharing systems to 6^{th} Edition UNIX and I discovered the effects of ‘&’. The effect of this, and even more so of multiple windows, is that users are free to load up the system until the response drops to a point they deem unacceptable. For obvious reasons, the shape of the curve means that users will end up close to the knee of the curve.

Most users will spend most of their time on the flat part of the graph close to the knee. Every so often, they will push the system over into the chaotic state and experience a long pause. The system will exhibit a kind of *hysteresis*; once it gets into the chaotic region it will take some time to recover. The spectrum of responses the user sees will be bi-modal, many short pauses and a few very long pauses. Since the users cannot predict the results of their actions, the long pauses will be unexpected, and thus memorable. The combination of the shape of the curve and the natural way to use a multi-process, virtual memory workstation means that the user will be unhappy.

There’s another aspect of the user’s behaviour that makes the transitions to the chaotic region more unpleasant. What do you do when the system stops responding for a while? You prod it in some way to see if it’s still there, perhaps by moving the cursor, hitting return in some shell window, or popping a menu. All these things make the problem worse, because they cause processes to be scheduled and increase the demand for pages.

4.2. Adding Memory is Good, but ...

The natural first reaction to the graph is to add memory immediately. Adding memory not merely improves performance, it improves predictability as well. Although memory is wonderful stuff, it is expensive (in low-cost workstations the memory is often worth more than everything else in the box put together) and tends to come in large chunks (the minimum memory increment in current technology is typically 4 megabytes and it will soon be 8).

Notice that the graph is labelled ‘‘Task duration vs. available memory’’. What is the memory being used for, and why isn’t some of it available? When Berkeley Unix is booted, it normally reports two numbers; on my machine they are:

```
mem = 4096K (0x400000)
avail mem = 3325952
```

The second of the numbers doesn’t mean what it says. In the good old days on the PDP11 kernel memory was statically allocated, but nowadays the kernel allocates much of its memory dynamically through a version of *malloc()*,[6] and this makes it more difficult to work out how many non-pageable (locked-down) pages the kernel is really using. Table 1 shows the kernel’s typical use of locked-down pages on my machine;

Table 1 – Non-pageable memory usage	
Used for	8K Pages
text + data	77
heap	35
mbufs	13
UFS bufs	10
valloc space	10
u structs	17
locked-down total	162
+ ‘‘avail mem’’	406
total	568

if you believe these numbers my machine has grown an extra half megabyte of memory! The discrepancy represents the memory the kernel has dynamically allocated and thereby stolen from the pageable page pool. In fact, my machine normally has about 350 pages in the page pool.

4.3. Saving Memory is Much Cheaper

A 4M machine with 8K pages not merely has very little memory, but it also has very few pages in the page pool. So:

- Adding even a few pages to the page pool will have a big effect.
- Using the pages in the pool effectively is important.

The best thing you can do to add pages to the page pool isn’t to buy more memory, it’s to configure your kernel carefully. We find that people often continue to run the (huge) generic kernel we supply to get the boot process started. This is like beating your head against the wall; its wonderful when you stop.

But even following the manufacturer’s instructions isn’t always enough. The *SDST160* SunOS4.0 kernel Sun advised its customers to run on a 3/75 had only around 290 pages in its page pool on my machine. By carefully eliminating all unneeded options and resizing kernel data structures downward, I was able to free up a lot of pages, and since then the advice has been changed.[3]

On my machine, the kernel is wiring down around 1.25M. The data points on the left of the graph are 0.25M apart, so they correspond to 20% of the kernel memory. Relatively small changes to the configuration of a typical workstation kernel can make a 20% difference to the amount of memory it uses, and thus a significant performance difference.

Other good things you can do include:

- Using shared libraries. Experiments measuring the age of pages (the length of time they have survived without being paged out) in my system show that the oldest pages almost always come from the shared *libc*. Old pages are being used effectively, normally because they are shared by several processes.
- Reducing your program's demand for space. Excessive *malloc()*-ing is bad for you; it creates pages that are non-shared by definition. On the other hand, although the stack is equally private, its pages are almost always in memory and tend to be re-used more effectively.
- Thinking about locality of reference for text. Re-ordering modules in libraries can pay big dividends. So can moving infrequently used code, such as error reporting, onto other pages, or even into other libraries.
- Thinking about locality of reference for data. Putting a layer over *malloc()* that arranges that data objects that will be referenced together appear adjacent in memory can make a big difference.

4.4. Adding Disk Bandwidth is Important Too

Why could my PDP11/70 support 60 times as many users as my Sun 3/75? I admit that these days I'm much more of a load on the system than I was then, since I now expect my machine to paint on a megapixel display, track a mouse every 1/30 second, support a window system, network file access, and, of course, *sendmail*. But a factor of 60? The key difference between the systems is I/O. The PDP11's SMD disks were faster than the Sun's SCSI disk, and above all it had several disks and used them effectively. With 5 drives on two controllers, it often had two transfers in progress *and* another two drives seeking, while the CPU was running a user process. Simply comparing the transfer rates of the two drives gives no idea of the PDP11's huge advantage in overall disk bandwidth.

We ran a blind taste-test experiment, in which users were asked to complete a complex task using the window system and a number of applications three times each, comparing the subjective performance. All the systems appeared identical, with the same software, display and keyboard. Although all the machines had 4M of memory, unknown to the subjects the machines had different CPU architectures, MIPS ratings from 3 to 5, different versions of the operating system, disks, and so on.

In general, users were unable to detect any difference between the systems. The only difference they could reliably detect was between disks; some of the systems had a 70M SCSI drive and some a 140M SCSI drive with twice the transfer rate. The systems were all operating in the disk-saturated region of the curve, and adding disk bandwidth was the *only* thing that made any difference. It is very easy for a customer to compare MIPS ratings of workstations, but very difficult to compare disk bandwidth. Given the increase in program size, and thus the increasing demand for disk bandwidth, it is clear that the effective disk bandwidth of typical Unix systems has declined in recent years while their MIPS rating have increased.

4.5. VM-aware Design

When they have something to do, processes compete with each other for the pages in the pool. By being constantly aware of this, we can design systems that use the page pool more effectively by scheduling fewer processes less often.

For example, there is long-running debate among the users of UNIX window systems between the advocates of *click-to-type* focus management and the advocates of *focus-follows-cursor* mode. Note that this debate doesn't happen among Mac users; the only window that is really active is the front one, and focus-follows-cursor simply isn't a concept. Users want some indication of the window that has the input focus, so each application that gets the focus wakes up and makes some subtle change to its appearance to indicate that it now has the focus.

Take a focus-follows-cursor system and a screen crowded with windows, scrub the mouse around without typing anything and watch the performance monitor's paging activity graph. It will rise sharply even though the system isn't really doing anything, because of all the processes waking up and saying ''I've got the focus''. In fact, the situation is even worse than that. Suppose the screen is being driven by an X server.[7,8] Each time the mouse moves from an *xterm* window (client A) to another *xterm* window (client B):

- The X server gets the CPU, generates the window crossing and focus transfer events, sends them to the clients (making both A and B runnable), and goes to sleep.
- Client A (or B) gets the CPU, examines the event, and sends the protocol requests to change its cursor from solid to hollow (or vice versa) to the server (making it runnable), and goes to sleep.
- Client B (or A) gets the CPU, examines the event, and sends the protocol requests to change its cursor from hollow to solid (or vice versa) to the server, and goes to sleep.
- The X server gets the CPU, reads the two sets of requests, adjusts the screen accordingly, and goes to sleep.

For one mouse movement, we have a best case involving four schedulings of three separate processes. The worst case is much worse, since the window manager process may also get involved, and the server may get the CPU and process one client's requests before the next client gets it. It appears that in a VM-aware user interface, click-to-type focus management is the way to go.

Another feature of many UNIX window systems is the need to wake up processes to repair damage to their windows. When the window tree is changed in such a way as to damage some windows, the X server sends Expose events to all the clients which need to repaint some of their image. They all become runnable together, and start competing for the pages in the page pool. The sudden spike in demand for pages may well be enough to push the system into the chaotic region of the graph.

We did an experiment in which we modified the kernel-based SunWindows system to request a repaint only from the process with the front-most damaged window, and to wait until the repaint was complete before requesting a repaint from the next window back in the stack. This was a big improvement:

- Even with surplus memory, the visual impact of having the windows repainted in a predictable order, and having each window repaint completely before another window painted, was much better.
- Even with surplus memory, on average more pixels were updated sooner. This is because, on average, the front window has more pixels exposed than windows deeper in the stack.
- On limited-memory machines, the page traffic generated by each re-arrangement of the screen was much reduced.

Reducing the impact of a given set of damage to windows isn't the only thing we can do. We can also change the user interface to reduce the damage-prone-ness of the window layout. An extreme example of this is the *tiling* window layout used, for example, in Carnegie-Mellon's *Andrew* system.[4] Windows are arranged in columns; they never overlap and no window layout change can cause more than one window to repaint. In the worst case for an X tiling window manager, only the window server, the window manager, and a single client would be competing for the pages in the pool.

There is an index of damage-prone-ness; the average number of processes that are woken up by a single to-level window manipulation. It is possible for a window manager to estimate this:

```
index = 0
for (each mapped top-level window A) {
    if (A is at least partly visible) {
        for (each mapped top-level window B that A overlays) {
            if (there is at least one pixel in B which would
                be visible if A were unmapped) {
                    index++
            }
        }
    }
}
```

Window managers might use this estimate as input to their window layout policies, striving to minimise the index. Even in an overlapped window environment, there are things that can be done. For example, the window manager might automatically iconify windows that are deep in the stack and do not have the input focus.

4.6. The User is Part of the System

In the good old days of timesharing, the system load was statistical in nature. It resulted from a process of averaging the actions of a large number of users. In a workstation, the system load is largely the result of the actions of a single user, and is highly correlated with the actions of the system.

While the system is operating in the flat part of the graph, load is additive. Adding a small constant amount of load will degrade response by a small constant amount. Users can build a reliable mental model of system performance, and can predict the effect of their actions. Once you get to the knee of the curve, load is no longer additive. Adding a small constant amount of load will degrade response by wildly different amounts depending on where on the curve the system is. Users cannot predict the effects of their actions.

Because the load that the user imposes on a workstation is closely related to the system's behaviour, unpredictability in the system's response will lead to variations in the imposed load as the user's estimates of the load it can absorb prove wrong. These spikes in demand will push the system into the chaotic part of the graph, causing abnormally long delays, reinforcing the unpredictability, and the user's unhappiness.

It's a common observation that experienced workstation users often run a performance monitor, an EEG-like trace of system parameters such as CPU load and disk traffic. Why do they add to the system load in this way? It prevents the need to make the response worse in order to obtain reassurance that response hasn't stopped completely.

The performance monitor also allows the user to find the knee in the curve more accurately, and predict the systems behaviour better. Interpreting its output takes skill, and it may well be that the information available could be presented in a more effective way. A better presentation should have measurable effects on throughput as the user manages the workload to stay closer to the edge of the ordered region; experiments are needed to investigate this effect.

A VM-aware system should also attempt to prevent the user getting into trouble. Systems that fork subtasks might monitor the load before attempting the fork, and putting up an alert to warn the user if the load is heavy enough that adding the sub-task would go over the knee. We did an experiment in which the *suntools* program's startup phase was modified to delay starting new applications if the paging load caused by earlier applications was too heavy. The paradoxical result of delaying the start of some of a set of applications was that the set as a whole got started sooner. The effect is similar to that of the repaint-front-to-back strategy.

Once again, this experiment shows that expectations based on linear behaviour are wrong. In fact, we are faced with another paradox. Adding memory (or reducing kernel memory demand) to a system that is around the knee of the curve (and most will be) will probably increase throughput, but the user will probably not be any happier. They will simply add load to the system to bring it back to the knee in the curve, and as a result will continue to see occasional long pauses.

4.7. Simplistic Benchmarks Are Harmful

As we saw earlier, MIPS ratings may have no correlation with perceived performance. But they are easy to measure, and they have driven the workstation market for the past few years. In order to achieve better satisfaction from their systems, users (customers) need a benchmark measure that correlates better with perceived performance.

Unfortunately, we have also seen that the user is an essential part of the feedback loop that determines system performance. Building a benchmark that modulates system load as a real user would is very hard. It seems improbable that one could be developed in the near future; much more research into the nature of the relationship between system performance and user actions is needed before understanding sufficient to do this would be available.

In the meantime, using a script-style interactive benchmark to draw the task duration vs. available memory graph for a range of systems would allow a somewhat better prediction of likely perceived performance than MIPS or the (only marginally more sophisticated) benchmarks in current magazine product reviews.[9] Some of the facilities needed in X servers to do this are being agreed under the auspices of the group looking at X testing and validation.

5. Conclusion

Virtual memory systems running script-style interactive tasks exhibit a task duration vs. available memory graph with a characteristic shape. This combined with the freedom a multi-process system gives the user to modulate system load, and the tight correlation between system and user actions in a single-user workstation environment, leads to a region of unpredictable and thus irritating system response around the knee in the curve.

Normal user responses to systems that are far from the knee in the curve move them closer to the knee, by adding load or memory as the case may be. Thus, most users spend most of their time close to the knee, experience unpredictable response, and feel dissatisfied. There are two possibilities to explore in an attempt to improve the user's satisfaction with the system:

- Improve the system's behaviour close to the knee. Among the tactics that can be employed are eliminating spikes in the demand for memory, changing the scheduling and paging policies, and exporting to the applications more relevant measures of system load (such as the number of processes in short-term I/O wait).
- Improve the user's behaviour close to the knee. Among the tactics that can be employed are performance monitors displaying more detailed and relevant system load parameters, applications that estimate the load that user commands will impose and warn the user before executing commands that are likely to push the system over the knee.

It is difficult to develop benchmarks that show this problem; because it is directly related to the way a user uses the system. The task duration vs. available memory graph can be used as a rough indication of likely system behaviour, but much more work is needed to develop benchmarks that model the user's behaviour properly.

Acknowledgements

Special thanks are due to Mark Liu, Marty Itzkowitz and the other members of Sun's Software Performance Engineering Group, who did some of the experiments. Thanks are also due to Sue Knapp, Joe Provino and Sun's East Coast system software team for doing most of the other experiments. Above all, thanks to Gary Sager for giving me a diskless 4 megabyte Sun 3/75 for my new office and thereby re-awakening my interest in performance.

References

1. Steven M. Christensen, ‘‘Monstrous Performance,’’ *Unix Review*, vol. 6, no. 12, pp. 61-67, December 1988.
2. James Gleick, *Chaos: making a new science*, Viking Penguin, New York, 1987.
3. Sun Microsystems, ‘‘SunOS4.0 Read This First, Appendix E,’’ 800-1737-17, Rev. A., November 1988.
4. James H. Morris and others, ‘‘Andrew: A Distributed Personal Computing Environment,’’ *Comm. Assoc. Comput. Mach.*, vol. 29, no. 3, pp. 184-201, March 1986.
5. C. Northcote Parkinson, *Parkinson's Law and other studies in administration*, Houghton Mifflin, Boston MA, 1957.
6. Robert Rodriguez and others, *A Dynamic UNIX Operating System*, pp. 305-319, USENIX Conference, San Francisco, CA, June 1988.
7. Robert W. Scheifler, James Gettys, and Ron Newman, *X Window System: C Library and Protocol Reference*, Digital Press, 1988.
8. Robert W. Scheifler and Jim Gettys, ‘‘The X Window System,’’ *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1987.
9. David Wilson, ‘‘Tested Mettle,’’ *Unix Review*, vol. 6, no. 12, pp. 87-95, December 1988.