Linux Scalability — from the micro to the HUGE

Peter Chubb and Darren Williams

National ICT Australia
and
Gelato Project, School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
peterc,dsw@gelato.unsw.edu.au

April, 2005

Abstract

Linux is possibly the most scalable operating system *ever*. From the Linux Watch to the 1024 and more processor ALTIX machines from SGI; one kernel runs on them all.

The kinds of challenges that are presented by the very small are in some way related to the challenges of the very large, but for different reasons. For example, memory footprint is very important for both: on small systems you just don't *have* much memory; on large systems, caching effects dominate performance — if you can keep cache lines local, you win.

Likewise, power management is extremely important for both ends of the scale. At the low end, to maximise battery life, at the high end, to minimise air-conditioning costs.

However, the mechanisms available, and the specific goals differ.

I shall attempt to explain what scalability *is* as far as an operating system kernel is concerned, explain where Linux has come from and is now, and then give a possible vision for the future, as to how Linux can maintain and enhance its already good scalability. I'll also present benchmarks on a variety of machines to show where the problems are now.

1 Introduction

Linux runs from essentially the same source code base on a very wide variety of machines, from Ricoh's RDC-i700 digital camera [OV04], and the Linux watch [GK02], to the NASA 'Thunder' system of 10240 64-bit Itanium processors and close to a terabyte of real memory. In this sense one could say that it is 'scalable' — the operating system runs on all of these machines, presenting essentially the same programming API to user programs on all of them.

This kind of scalability is provided by phase changes: the binary that runs on the watch is different from the binary that runs on the supercomputer. Not only are the processors different, but the Linux kernel is highly configurable, allowing support for different features, or the use of different algorithms, while providing the same API.

However, core algorithms (like memory management and the processor scheduler), while they are adjusted slightly for the two extremes (both have to be aware of the NUMA topology on high-end machines; the page table has fewer levels on low end machines), are essentially the same across all platforms.

2 What is scalability anyway?

A computer has some amount of resources — so much memory at such and such a bandwidth, so many processors, so much disc space etc., etc.

One can think of this as a set of orthogonal vectors in N-space, where each axis is a different resource. A given machine configuration becomes a point in that space. A simple way to visualise this is the spiderweb diagram — see figure 1. Each arm is a different resource; distance from the centre is the amount of that resource available from the hardware. A particular machine can be plotted as a line joining points on the arms

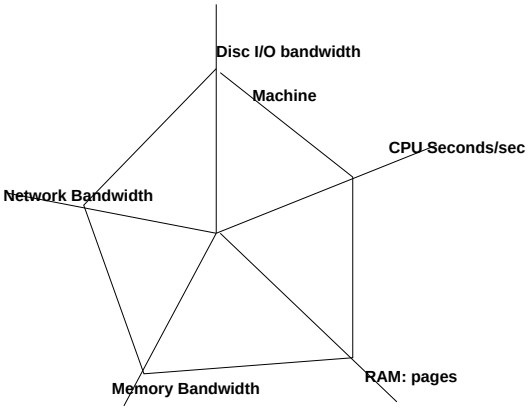

Figure 1: Scalability diagram, showing resources available to a particular machine configuration

of the web.

Different workloads use different amounts of these resources. If the operating system is perfectly scalable, then adding more resources will lead to more resources being made available to the workloads.

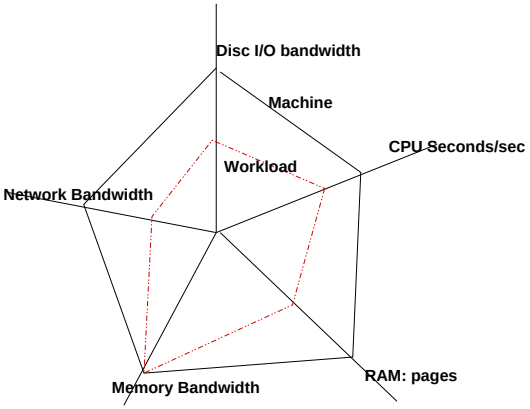

Figure 2:

A workload's resource needs can be plotted on a spiderweb too: for example, the workload in figure 2 is limited by the memory bandwidth available; by spreading its memory across multiple nodes in a NUMA machine it may speed up.

Assuming a scalable workload (one that uses all available resources, so that it will speed up if more resources are available), one can plot work achieved against work presented. The expected graph is in figure 3. As the available work increases, the throughput increases until some resource is exhausted, whereupon attempts to do more work are ineffectual. On a machine with twice the resource (on the figure, number of processors) the horizontal 'resource exhausted' part of the graph will be twice as high if the operating system and hardware is perfectly scalable.

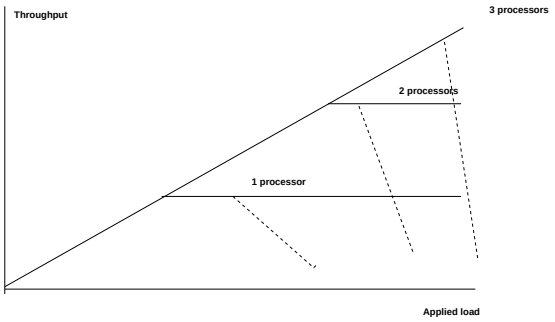

Figure 3: Ideal scalability: work achieved vs work presented

In general, the lines shown as horizontal will droop a bit, because of increased overheads in handling the presented load. In extreme cases, 'livelock' — shown by the dotted lines — will result. Livelock occurs where the extra time taken to process and discard more presented work takes away from the time given to perform real work.

Unfortunately, real operating systems, workloads and hardware are not perfectly scalable. Overheads and bottlenecks within the OS itself usually mean that doubling a resource makes less than twice that resource available to user processes.

Our work has concentrated on scaling two resources: processors and memory.

3 Scaling processors

Multiprocessors with small numbers of processors tend to be fully symmetric (SMP): each processor has the same latency to memory as any other processor. However, some processor packages have internal multithreading (SMT) — for example, the Intel XEON — where a single processor looks like two or more processors that share cache.

When the number of processors becomes large, manufacturers switch to a non-uniform memory architecture (NUMA). Processors and memory are distributed into several nodes; each node's processors can access both any memory on the local node, and, at a penalty, memory on more distant nodes.

Thus when scaling the number of processors there are four different architectures to consider:

1. Single processor
2. SMT processor
3. SMP
4. NUMA

The single processor case is the simplest. There is only one processor to schedule work onto; and there is only one thing happening on a processor at a time, so there is no need for fine-grain locking in the kernel.

The other cases all require locking. Moreover, for good performance, the scheduler needs to be aware of the underlying machine architecture.

For good performance in an SMT system, unrelated threads should be scheduled onto the same processor — so that computational units unused by one thread can be used by another. For good performance in an SMP system, related threads should be scheduled at the same time on different processors. For good performance on a NUMA system, related threads should be scheduled on nearby processors, preferably in a 'gang'.

Fortunately, many of the issues in rescheduling a thread are the same for all cases: as a general rule, a thread should be scheduled onto the same processor it ran on last time. For SMT and SMP systems this is because, if a thread ran recently on a processor, it will still have data in the processor's cache. (On some architectures, such as ARM, this will not be true, as those processors have to flush their cache on every context switch; such systems are normally uniprocessors).

The same is true, of course, for NUMA systems; but on a NUMA system it is even more important that processes stay where they were first put, and that processes be spread appropriately around the system. Linux provides an interface (`libnuma.a`) to allow manual placement of processes and the memory used by processes — the *best* placement is usually dependent on the workload, and cannot easily be intuited by the kernel.

4 Scaling Memory

Linux in one form or another runs on machines with from less than ten megabytes of memory up to machines with terabytes of real memory.

4.1 Memory Amount

On small machines, the amount of memory used is vitally important. Small machines do not have memory to waste. Techniques used to reduce memory consumption include:

- omission of rarely-used features (for example, there's no point in including the PS/2 driver on an embedded machine without a keyboard)
- packing data structures to remove padding (for good performance where memory is plentiful, data items are often aligned on cache-line boundaries; for memory-short systems, the alignment padding can be removed)
- reducing the size of integers used to store data (for example, using a 32-bit integer to hold the size of a disc instead of a 64-bit one).
- Scaling data structures by memory size. For example, there is a utility function, `alloc_large_system_hash()` for allocating hash tables such as for the `dentry` cache, which automatically scales the size of the hash table to the size of directly-addressable memory.

On large machines, it is acceptable to waste a small amount of memory for improved performance. The key to good performance on a multiprocessor is, on one hand, to eliminate unnecessary cache-line sharing, and on the other, to try to fetch several items used together into cache at once. Linux adds padding to data structures so that they start on cache line boundaries, which can lead to performance improvements, as can packing closely related items into a single cache line. The exception is the spin lock — moving a spinlock into a separate cacheline from the data it protects can lead to a major performance gain when the lock is contended.

Data structure scaling is a separate issue: while linear scaling works reasonably well for small memory machines (less than say 16G of memory), on machines with more memory (and particularly NUMA machines) this can result in tables being far too large. Moreover, on a NUMA machine, although the total amount of memory may be in the terabytes, the amount on each node is relatively small. The algorithm for allocating memory at boot time is fairly simplistic, so in a NUMA machine, memory in cell 0 can become exhausted before any user programs even start running.

4.2 Memory Placement

This is a general problem with NUMA: data structure placement needs to be thought out carefully. Accessing a data structure in local memory has lower latency than if the data structure is in another cell's memory (of course, after the cache line is fetched, subsequent accesses are *much* cheaper).

Linux uses a strategy for memory placement called 'first touch' — when a process first tries to use a page of memory, the kernel tries to allocate it on the same node that the process is running on. This, combined with a 'balance-on-exec' strategy, works reasonably well for mixed workloads of many small processes. It is also the strategy assumed by many MPI programs (and programmers!) — MPI programs often start by allocating a large amount of memory, then split into threads (which, it is hoped, will be spread around the machine by the scheduler), then each thread touches part of the large shared mapping to ensure that it's local to that thread.

There are, however, a number of problems with 'first touch':

- Linux caches pages read from disc. A common pattern for multithread processes is to start by reading the data, then many threads cooperate in working on that data. 'First touch' means that all the data resides on the node where the read happened; a large file can easily take up more than the available node-local memory (see [BBH+04] for details, and possible solutions).
- Although latency to remote nodes is higher than latency to local memory, the total bandwidth available by striping data structures across multiple nodes is far greater than one node's bandwidth. Hence, for some access patterns it makes sense to move data to remote nodes, and to use deep prefetch to hide latency. This is particularly true on Itanium, where the compiler can easily issue speculative data load instructions to fetch data well ahead of where it is used, hiding the additional latency.
- Heavily multithreaded workloads fork (or create threads) many times; all the children remain on the processor their parent was on until other, idle, processors steal them. Thus the memory they use all comes from the first node.

The proposed solution to all these problems is to make it possible to control memory and process placement from user space. While this certainly does solve the problem, it creates new ones: in particular the programmer has to become aware of the topology of the machine, and so it becomes harder to write portable code. In fact, it's common to have to experiment with different placements to find out what works best.

In addition, the new `libnuma` API is Linux-specific. This of course reduces portability.

4.3 Other Memory Issues

As available real memories continue to increase in size, there will be new problems.

As a general rule, memory bandwidth is increasing at around 27% per year, but latencies are improving by only 7% per year [Pat04]. Larger memories in general have longer latencies, because of higher capacitances on the buses, and longer data lines. To try to hide latency, manufacturers are increasing the sizes of on-chip cache (the new Itaniums have 9M of L3 cache, Xeon-MP have 8MB), but this works only for workloads with good locality-of-reference. Streaming workloads such as particle physics data capture or video processing do not benefit very much from the larger caches alone; however, by prefetching data in the same way that can be done in a NUMA system, latency can be hidden.

Also, RAS (Reliability, Availability, Serviceability) issues arise: in a machine with a petabyte

of real memory, several chips will probably be broken at any given time.

5 The Power Issue

Much embedded computer equipment is designed also to have low power consumption. There are several reasons for this:

- If running on batteries, the lower the power dissipation of the device, the longer the batteries last.
- If placed remotely, one can multiplex a small amount of power on data cables (phantom power), and so save cabling costs. To make this work, the peak current drawn for the device must be kept low — typically less than one or two hundred milliamps.
- Equipment embedded into other machinery may not be able to use fans or similar to remove excess heat. So it is important not to generate any.

What is less often recognised, is that efficient power management is also important at the high end. Every watt generated by a computer part first has to be ducted away from the part, then away from the room the machine is in. Thermal management in a computer is a major part of the engineering design effort, and air-conditioning a major cost of running at the high end.

So far, the power management work in Linux has been aimed primarily at laptop computers, where battery life is the main consideration. And the main area that works in 2.6 is processor frequency scaling.

Interesting areas that could be addressed in future include memory hotplug, where banks of memory can be put offline and into a low-power state; processor hotplug (the same, but for processors), and PCI hotplug. The user space at present assumes that if something's plugged in, it should be powered up and ready to use; but this could be changed.

6 Scalability Results

Previous work (see [BBH+04]) has shown good scalability for processor-intensive workloads on SGI's ALTIX platform, which is NUMA. When we repeated the tests on a borrowed 16-way ALTIX with a mixture of I/O and CPU-bound workloads, scalability wasn't so impressive. Unfortunately, access to the remote ALTIX machine was limited, so we could not run profiling etc., to find the real problems.

6.1 The Benchmark machine

Fortunately, HP lent us access to a 16-way Olympia. This machine is a 4-cell NUMA machine, each cell having 4 Itanium-2 processors, and 8G memory. Unfortunately, one of the cells died shortly after we started, so the results are only up to 12-way.

The Olympia is interesting in that it can be configured in one of two modes: memory can be local to each cell (NUMA) or the memory on each cell can be interleaved on a cache-line basis across all the cells, making it look like a pure SMP machine.

6.2 AIM-7 results

We used the OSDL AIM7 benchmark to see how throughput scaled with number of processors. This benchmark attempts to simulate a large number of users doing different things: it can be tuned to be either purely compute-bound, purely I/O bound or mixed. In order to explore scalability in the filesystems and I/O paths, while not being bottlenecked on a single spindle (the machine we had access to was not overly endowed with high-performance disc), we created a largeish (4G) ram disk, and then put different filesystems on it.

The benchmark was modified slightly so that we could plot throughput vs number of clients (number of clients is proportional to the multiprogramming level), rather than allow the benchmark scripts to find a sweet spot and report just that.

We ran two workloads: a CPU-intensive workload to stress the scheduler, and an I/O intensive workload to stress the file systems. We used a ram disk to avoid the problem of long disk I/O latency masking multiprogramming issues.

The I/O intensive benchmark was run with tmpfs, and xfs on a ram disk. Theoretically

tmpfs should perform the best of these, as it avoids most of the the overheads involved in laying out data structures on a disc — tmpfs keeps data in the page cache directly. Unfortunately, bugs in XFS prevented our getting good results — see below. We also tried ext3, but the machine crashed at relatively low levels of multiprogramming.

6.2.1 CPU scalability

As expected, a CPU-bound workload with almost no I/O scaled almost linearly with number of processors (see figure 4). As you can see, the horizontal sections of the graph are regularly spaced, and the end point is a little farther to the right for each increment in number of processors. This benchmark was run with the machine in SMP mode (memory from the 3 cells interleaved at cache-line granularity). The 8-way case is interesting — it falls off a lot more rapidly than the other cases. However, we haven't yet had time to investigate this case, and the maximum jobs per minute scales beautifully — see figure 5 for a different representation of the same data.

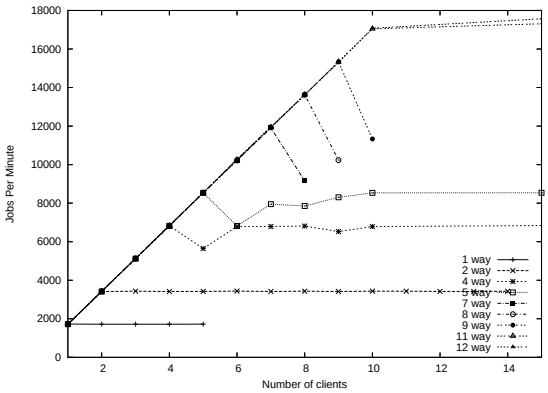

Figure 4: OSDL AIM-7 results for varying numbers of on-line processors on a 12-way HP Olympia: jobs-per-minute against multiprogramming level

6.2.2 I/O scalability on tmpfs

The graph in figure 6 shows OSDL AIM-7 results for an I/O-intensive load, using tmpfs as the underlying filesystem. The graph shows very poor scalability. Even at very low levels of multiprogramming, the graph is curved, showing

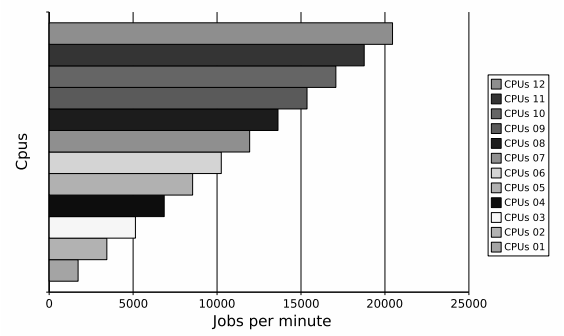

Figure 5: OSDL AIM-7 results for varying numbers of on-line processors on a 12-way HP Olympia: peak performance against number of processors

that there is a bottleneck that is independent of the benchmark. And by the time one gets to 10 to 12 processors, the individual curves overlap, showing negative scalability.

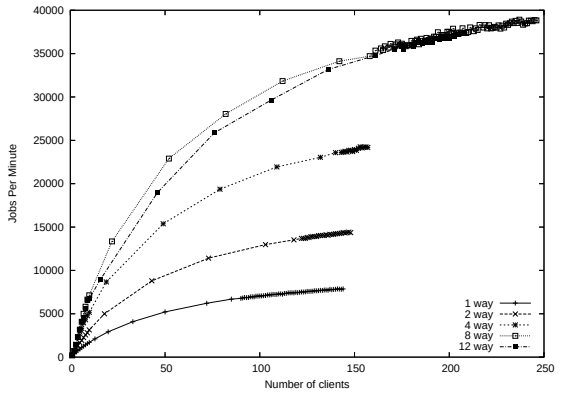

Figure 6: OSDL AIM-7 results for an I/O-intensive workload, using tmpfs.

Profiling at the 100-client point showed that the problem lies in the VM system, not in the filesystem. Using SGI's lockmetering patch showed that one problem was the spinlock in `struct as` protecting the radix tree. Replacing this with a multi-reader lock to allow concurrent read access changed the results to look like 7. These results are still not good. Notice that the slope is still positive on the right hand end of the graph, implying that there is still spare capacity that can be used. In addition, doubling the concurrency available (by doubling the cpu count) gives linear improvement in throughput. This effect is still under investigation.

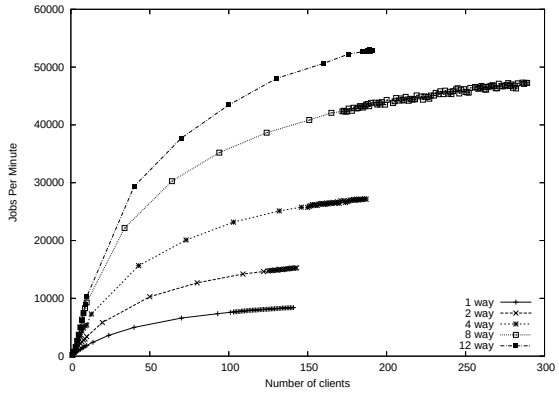

Figure 7: OSDL AIM-7 results for an I/O-intensive workload, using tmpfs, with a rwlock instead of a spinlock protecting the radix tree.

6.2.3 I/O scalability on ext2

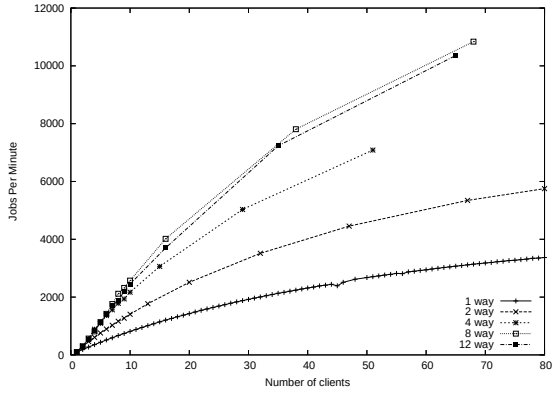

Figure 8: OSDL AIM-7 results for an I/O intensive workload, using ext2 on a ram disk

The results for ext2 also show poor scalability above four processors. Eight and twelve way results overlap, and are less than twice the four-way results — see figure 8. Lockmetering shows contention on `inode_lock`, a spin lock that protects the lists of dirty and clean inodes in the system. If I/O happened to real disks, the time between acquisitions of this lock would be limited by the speed of the attached storage, so one would expect the contention to lower.

Even with the contention, the total throughput is fairly good.

6.2.4 I/O scalability on ext3

The ext3 filesystem had severe bugs causing lockups when we began to benchmark. When these were fixed, it became apparent that ext3

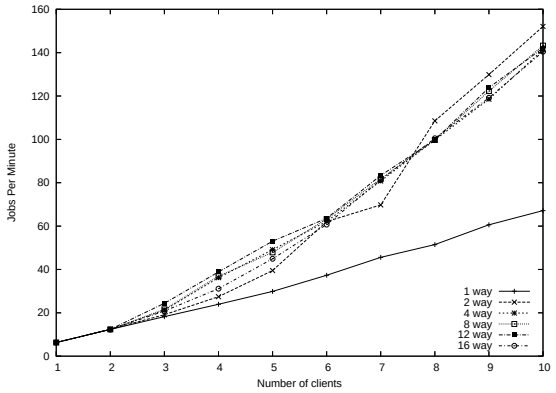

Figure 9: OSDL AIM-7 results for an I/O intensive workload, using ext3 on a ram disk

is single-threaded on kjournald, and doesn't really scale well with number of processors (kjournald is a daemon that handles the journal for the filesystem). The benchmarks that ran in a few minutes on tmpfs took several hours on ext3 on a ram disk.

On a real disk, kjournald will be slowed to the I/O speed of the attached storage, so this is probably not a problem except for *really* fast storage such as a SSD.

6.2.5 I/O scalability on XFS

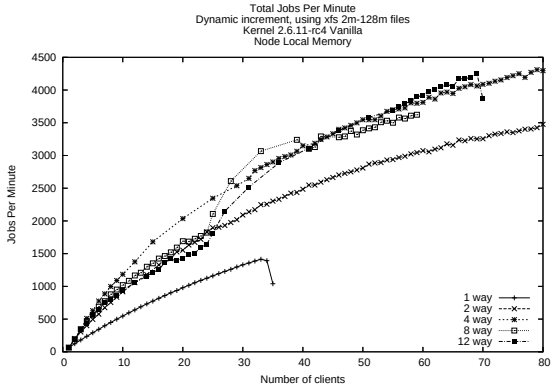

Figure 10: OSDL AIM-7 scalability, on XFS.

XFS scalability also seems poor, although its absolute throughput is far better than ext3. In particular, there is a race condition somewhere (still under investigation) that occasionally causes lockups under heavy write loads. We did manage to get a few results, seen in figure 10. The poor scalability appears at initial analysis to be because of single threading in the journalling code when allocating more space for

the log, but more analysis is needed. Interestingly enough, the xfs results also show significant time spent rebalancing load across the cpus (in `sched_migrate_task()`), so perhaps on a more realistic workload it would behave better.

7 Future Work

The next steps are firstly to try to find some more realistic benchmarks (we're planning on trying OSDL's DBT-[123] benchmarks ([OSD]) if we can get them to perform properly on IA64), and to do soem measuring with a real disc array.

8 Some conclusions: One size *doesn't* fit all

There are two reasons that Linux scales as well as it does. Firstly, it is highly *configurable*. Features (such as four-level page tables) that are not needed for a particular installation can be compiled out. Unusual features (like a NUMA-aware scheduler) can be compiled in.

Secondly, a lot of ad-hoc work has been done benchmarking and working around bottlenecks in the code, as Linux has advanced from a x86, single processor only operating system to something that spans a couple of dozen architectures and three orders of magnitude of processor count.

However, it can be argued that we're reaching the limits of what can be done by tweaking existing algorithms. Recently, pluggable I/O schedulers have been introduced to the mainline kernel, so that, for example, small systems that have no disc can use a very simple scheduler; desktop systems can use a scheduler that minimises latency; and Enterprise systems can use a scheduler that provides fairness between competing users of the underlying storage.

For scalability, it's likely that pluggable CPU schedulers and page tables will be next. The existing 3 (or 4) level pagetable is optimum for processes with a small number of dense mappings. Efforts are currently being made (by Christoph Lameter at SGI, among other people) to improve scaling of the existing page tables, but it's unclear how far that can go without either hurting performance for small machines, or making the code very convoluted.

There is an out-of-tree patch for a pluggable scheduler interface (by Con Kolivas). In its current form it is unlikely to be merged. However, the approach is very useful, as it allows different, streamlined, schedulers for different expected workloads. For example, a fair-share scheduler for the enterprise, a multi-level hierarchical scheduler for HPC, and a low-latency scheduler for the desktop.

Being able to use different pagetables or schedulers would give the same kinds of flexibility one can have now with filesystems. However, to do this would require a major internal interface cleanup. It would also perhaps have one drawback: one of the reason for the present system's relative stability is that the same core codebase is used throughout. Scheduler or pagetable components that are less popular would tend to be tested less, and so allow bugs to persist for longer (and have more bugs introduced as normal kernel development proceeds), just as happens with filesystems today.

References

[BBH+04] Ray Bryant, Jesse Barnes, John Hawkes, Jeremy Higdon, and Jack Steiner. Scaling Linux to the extreme: From 64 to 512 processors. In *Ottawa Linux Symp.*, pages 133–148, Jul 2004.

[GK02] Dinakar Guniguntala and Vishal Kulkarni. IBM's Linux Watch: The challenge of miniaturization. *IEEE Comp.*, pages 33–41, January 2002.

[OSD] Open Source Development Labs. *Database Test Suite*. http://www.osdl.org/lab_activities/kernel_testing/

[OV04] Shigeki Ouchi and Alain Volmat. Linux porting onto a digital camera. In *Linux 2004 Conference*. UKUUG, 2004. .

[Pat04] David A. Patterson. Latency lags bandwidth. *CACM*, 47(10):71–75, Oct 2004.