**Scalability, I/O and All That**

AOS 2009.10.16

Peter Chubb
peter.chubb@nicta.com.au

---

**Scalability**

- Essentially the same kernel across wide range of hardware or not?
- Performance scales with hardware or does it?

---

**Phase Changes**

- Processor change
- Uni → SMP
- UP → SMT

---

Same kernel source, same core algorithms (in general); conditional compilation does change some features.

---

**NUMA**

- Cluster
Scalability Issues

- Number of Processors
- Number of Spindles
- Amount of Memory
- Power Consumption...
- Scalability depends on Workload

System Performance

The ideal system performance curve has three parts. At the left, throughput is determined by the rate at which work requests arrive; there is little or no queueing, and latency is governed by how long a job takes to do. In the middle, where the curve is, jobs sometimes arrive while a previous job is still running. Latency is governed by both the time a job takes, and the time it spends queued – which depends on how long previously queued jobs take. At the right of the curve, latency is determined almost entirely by queueing time; and throughput is determined solely by the time the server takes to do a job. In this state, one or more resources on the server are bottlenecked; removing the bottleneck will move the curve up and the bent part of the curve to the right.

System Performance

Gunther’s law:

\[ C(d) = \frac{d}{1 + \alpha(d-1) + \beta d(d-1)} \]

where:
- \( d \) is demand
- \( \alpha \) is the amount of serialisation: represents Amdahl’s law
- \( \beta \) is the contention in the system.
Scalability Considerations

- On-chip cache (ns)
- Local L2 cache (10s of ns)
- Local RAM (100s of ns)
- Other processor’s cache (μs)
- Disc backing store (10s of ms)

Other factors
- Locking ‘grain’
- Interrupts — to one processor or to any?
- Disc and other I/O parallelism
- Cache fights
- TLB contention

Goals of NUMA

1. Better performance at lower cost (two processors are cheaper than one twice-performance processor — and when the twice-performance processor is developed it can be plugged into the earlier architecture)
2. Hide architecture from user-mode programs
3. Scalable performance enhancement

NUMA Concepts

RAD the Resource-affinity-domain, (aka ‘Cell’)
SMP Symmetric Multi-Processing — All processors have equal status
Master-Slave Some system calls, interrupts must run on Master processor.
NUMA CONCEPTS

- ‘Bristled’ hypercube (e.g., SGI Altix).
- Plain hypercube (e.g., Compaq).
- $O(\sqrt[n]{n})$ hops for $n$ nodes.
- Each node has local memory.
- Some nodes have I/O (e.g., Disc, framebuffer, serial, parallel, etc)
- Clocks may be synchronised with NTP, or be delivered from a central master.
- Nodes may not be exactly homogeneous.

MEMORY HIERARCHY RE-VISITED

- On-chip cache (ns)
- Local L2 cache (10s of ns)
- Local RAM (100s of ns)
- Other local processor’s cache ($\mu$s)
- Remote processor’s RAM or cache (ms)
- Disc backing store (10s of ms)

PROCESS STATE

PIDs
- Want a single global PID pool
- Can manage in two ways
  - Single pool-manager, used by all, or
  - Use RAD ID as part of PID.
- Either way need a globally-consistent mapping $\text{pid} \rightarrow \text{proc struct}$

SCHEDULING AND DISPATCHING

- Memory switching is expensive
- Therefore try to schedule memory-sharing threads at same time.
- Two approaches
  1. Gang Scheduling
  2. Job models
SCHEDULING AND DISPATCHING

Dispatching

- Per-processor run queue
- Thread queued to same processor as last time
- Thread runs in the same cell as last time, if possible.
- Otherwise either goes onto queue for idle processor, or onto global queue.
- At dispatch time, threads taken from local pushback queue, or per-processor queue or global queue.

Gang Scheduling

- Threads run in a memory space
- Schedule memory spaces, let threads fend for themselves (RR or FIFO depending on user selection)
- Try to spread threads over ‘nearby’ cells.

Job Currency scheduling

- SGI/Cray Irix concept
- ‘Job’ is set of threads+processes that are scheduled together.
- Each job given a ‘weight’ in proportion to priority
- At regular intervals, time is handed out to jobs in proportion to weights
- Jobs divided into ‘over entitlement’ and ‘under entitlement’
- Threads belonging to under-entitlement jobs have

Job Currency Scheduling (cont)

- Scales to a few hundred processors only!
- Not as good as gang for many purposes
- Needs combination with topology scheduler.

General Principles

- Locality of reference (use per-processor data area where appropriate)
- Replicate read-only data, including user text segment for MT progs;
- use ‘const’ in kernel code; linker support for replicated RO-sections.
- Try to move shared elements ‘close’ to users; stripe shared memory.
- Use fine-grain MR-locks where appropriate
- Use fetch-and-op operations rather than locked sections
**GENERAL PRINCIPLES**

where possible.

**CACHE-FRIENDLY DATA LAYOUT**

- Group data in cache blocks by usage
- keep data consumers to the minimum
  - per thread
  - per processor
  - per proc
  - per RAD
- Replicate read-mostly data

**MORE GENERAL GUIDELINES**

- Keep differently locked-data in different cache blocks
- Group read-mostly data into a single cache-block
- Put read-write data in the same cache block as its lock
- Group data according to how frequently it is accessed
- Minimise number of cache blocks for most frequent code paths.

**TOPOLOGY SCHEDULING**

(e.g., SGI’s ‘Miser’ program, Compaq’s Processor Sets)

- Reserve RADs for particular job(s)
- Can reserve nearby memory in other RADs as well if nec.
- (pinning, processor sets)

**SOME CONCLUSIONS**

- Can get scalable performance out of ccNUMA
- Needs careful attention to detail in OS
- Tuning for one architecture can break others.

**REFERENCES**

- Schimmel, ‘Unix Systems for Modern Architectures’
On an HP RX2600 with two processors, the OSDL Aim7 benchmark is perfectly scalable.

But on the Altix, the scalability is awful. What’s going wrong? We don’t have access as root (borrowed machine from UWaterloo), and may not install our own instrumented kernels to see what’s going on, so let’s speculate that there’s an I/O problem (as we know the benchmark scales with processors on an SMP machine), and try with a faster disc:

Better, but we still don’t see the near-perfect scalability we expect. So try with a ram-disk:
Even here there is a problem. It looks like there’s a major problem in the I/O subsystem on a NUMA machine in Linux 2.4.21-sgi.

So let’s try 2.6.11, on an HP Olympia with 12 processors spread across 4 nodes. Again, a CPU-intensive, no Disk I/O workload shows reasonable scalability with number of processors.

This shows very poor scalability. The 12-way actually performs worse than the 8-way.
Microstate accounting showed much time on the CPU; q-syscollect showed much time in ia64.spinlock_contention, so we ran with lockmetering:

```c
struct page * find lock page (struct address space * mapping, unsigned long offset) {
    struct page * page;
    spin lock irq(&mapping->tree lock);
    repeat:
        page = radix tree lookup (&mapping>page tree, offset);
        if (page) {
            page cache get (page);
            if (TestSetPageLocked (page)) {
                spin unlock irq(&mapping>tree lock);
                lock page (page);
                spin lock irq(&mapping>tree lock);
            }
        }
    ...}
```

So let's replace the spinlock with a rwlock...
And bingo, it scales much better.

- Slope still positive
  - Spare capacity available
  - Latency a problem

Other File Systems On Ram disk

ext3  xfs

Microstate accounting output, ext3

Scalability by spindles

ext3  xfs

- Find the bottleneck
- fix or work around it
- check performance doesn’t suffer too much on the low end.
- Experiment with different algorithms
INTERRUPTS

- Steal time from current process
- Therefore can cause negative scalability.

SCHEDULERS AND DISPATCHERS

Scheduler

Priorities

- Real-Time
- Time share schedulers
- Entitlement Schedulers
- Batch Schedulers

ISOCHRONOUS SCHEDULERS

- For Multi-Media apps
- Guarantee a proportion of the processor.

Dynamic Window CPU Scheduler (DWCS)
Entitlement Schedulers

- Aim to share resources in proportion to entitlement
- Shares
- Or proportions (must be \( < 100\% \))

October 28, 2009  Scalability, I/O and All That...  56

Shares

- Allow entitlement holders to come and go.
- Each user is given shares
- Entitlement = \( \frac{S_u}{\sum_i S_i} \)
- where \( i \) runs over all active users.

October 28, 2009  Scalability, I/O and All That...  58

Simple Entitlement Scheduler

```text
0\% 0\% 0\% 0\% 0\%
```

October 28, 2009  Scalability, I/O and All That...  60

Simple Entitlement Scheduler

```text
0\% 0\% 0\% 0\% 0\%
```

October 28, 2009  Scalability, I/O and All That...  61
**SIMPLE ENTITLEMENT SCHEDULER**

- Each entity has some tickets
- Randomly select a ticket at each decision point
- Entities get time in proportion to how many tickets they hold.

**LOTTERY SCHEDULERS**

- P1 has 2 tickets
- P2 has 3 tickets
- P3 has one ticket

**PROBLEMS**

- Infinities.
- Processes that sleep.
- Processes too low level.

**INACTIVE ENTITY PROBLEM**

- Need feedback loop
- Boost under-utilising processes
- Slow down over-utilising processes
- Can lead to starvation
• Use Kalman estimator as usage
  \[ u_i = \alpha u_{i-1} + (1 - \alpha)\rho \]
  where the half life of the system is \( \ln(0.5) \).
• Can update once after multiple time steps:
  \[ u_m = \alpha^{m-i} u_i + (1 - \alpha^m) + (1 - \alpha)\rho \]
• Priority is then \( u \) where \( \epsilon \) is entitlement.
• Hierarchical possible: \( \epsilon = \epsilon_u \epsilon_p \epsilon_t \), providing
  \[ \forall x \in u, p, t : \sum x \epsilon_x = 1 \]

results:
• Patent granted
• Forms basis of ARMtech (used in Citrix).
• Earlier version in Solaris Resource Manager
• Free version was posted to LKML.
• Works very well, with low overhead.

• heads, tracks, sectors
• seek is very expensive