Context cont.

• Many HPC apps. are bulk-synchronous:
  – Iterative computation-phases, separated by barriers
  – Typically run on a dedicated partition

• Problem: poor scalability if granularity is fine
  – Grain size can be ≤ 1ms

```c
while(...) {
    compute;
    barrier;
}
```
Data from LLNL’s ASCI-White

From Jones et al, "Improving the scalability of parallel jobs by adding parallel awareness to the OS", SC’03 (with permission of authors)

Data from LANL’s ASCI-Q

From Petrini et al, "The case of the missing supercomputer performance", SC’03 (with permission of authors)
The reason: system noise

- **Noise** = sporadic per-node system activity
  - Unrelated to app & but deprives app from CPU
- One late task holds up thousands of peers to which is synchronizes
- More nodes => increased noise probability

![Diagram of processes and time]

<table>
<thead>
<tr>
<th>FAST OS</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colony</td>
<td>H</td>
</tr>
<tr>
<td>Config</td>
<td>H</td>
</tr>
<tr>
<td>DAiSES</td>
<td>M</td>
</tr>
<tr>
<td>K42</td>
<td>M</td>
</tr>
<tr>
<td>MOLAR</td>
<td>M</td>
</tr>
<tr>
<td>Peta-Scale SSI</td>
<td>H</td>
</tr>
<tr>
<td>Rightweight</td>
<td>M</td>
</tr>
<tr>
<td>Scalable FT</td>
<td>H</td>
</tr>
<tr>
<td>SmartApps</td>
<td>H</td>
</tr>
<tr>
<td>ZeptoOS</td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtualization</strong></td>
<td>H</td>
<td></td>
</tr>
<tr>
<td><strong>Adaptability</strong></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td><strong>Usage Models</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fault Handling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Common API</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Collective RT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I/O</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS Noise</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| H | High |
| M | Medium |
Sources of noise

• Nodes typically run a general-purpose OS (see Top500) in which there’s …

• **Process noise:**
  – native OS daemons (kswapd, ntpd, …)
  – Cluster daemons

• **Non-process noise (interrupts):**
  – Network interrupts
  – Periodic clock interrupts = **Ticks**
  – …

OS periodic clock interrupts = **Ticks**

• **Every few ms the OS wakes up**
  – CPU accounting, preemption, pending signals

• **Ticks always happen**
  – Even if the system is otherwise idle

• **Ticks are widespread**
  – Used by **ALL** mainstream GPOSs (Windows*, UNIX*)

• **Tick-resolution: until early 2000s, 100 Hz**
  – Nowadays: 250/1000 Hz (Linux, FreeBSD, …)
What’s next

1. Explain linearity
2. Quantify tick-noise impact, as affected by
   – Platforms
   – Grain sizes
   – Tick frequencies
   – Operating systems
3. Establish a general case against ticks
   – Not just ticks
4. Suggest how to deal with ticks

Why is perf. degradation linear?

*The “noise law”*

• $n = \text{number of nodes in the cluster}$
• $p = \text{probability a task is delayed:}$
  – On a single node
  – Within a compute-phase
  
  • Job’s per-phase no-delay prob: $(1-p)^n$
  • Job’s per-phase delay prob: $d_p(n) = 1 - (1-p)^n$
  • Can show that if: $p \leq 1/10n$
    then: $d_p(n) \approx pn \Rightarrow \text{linear}$
A desirable value for $p$

- $n \leq 10,000$:
  - Need $p \leq 10^{-5}$
  - For $d_p(n) \leq 10\%$
  - Need $p \leq 10^{-6}$
  - For $d_p(n) \approx 0$

- $n > 10,000$:
  (BlueGene/L)
  - Need $p \leq 10^{-6}$

Measuring the noise

- Micro benchmark
  - Pentium-IV, 2.8GHz, Linux-2.6.9
  - Default settings (1000 Hz, daemons, …)

```c
for( i = 1 \ldots 10^6 ) :  
  start = cycle_counter  
  for(...) /* one ms work */  
  ;
  end = cycle_counter  
  time[i] = end - start
```

- Analyze the $time[]$ distribution
Impact of interrupt-noise

<table>
<thead>
<tr>
<th>POSIX Priority Class</th>
<th>Avg. [usec]</th>
<th>Desktop Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default [SCHED_OTHER]</td>
<td>1060</td>
<td>6%</td>
</tr>
<tr>
<td>Realtime [SCHED_FIFO]</td>
<td>1080</td>
<td>8%</td>
</tr>
</tbody>
</table>

Reason: indirect overheads (cache)

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Interrupt/sec</th>
<th>Direct overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticks</td>
<td>~1000</td>
<td>0.78%</td>
</tr>
<tr>
<td>network</td>
<td>~6</td>
<td>0.06%</td>
</tr>
<tr>
<td>Sum</td>
<td>~1006</td>
<td>0.84%</td>
</tr>
</tbody>
</table>

Reducing Hz [CDF]

No Cache [1-CDF]
### Granularity & Platform

![Graph showing delay over minimal phase (%) for different timescales (1μs, 10μs, 100μs, 1ms, 10ms).]

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock MHz</th>
<th>Cache Kuops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>main mem</td>
<td>bus</td>
</tr>
<tr>
<td>Pentium-IV-2.4</td>
<td>266</td>
<td>533</td>
</tr>
<tr>
<td>Pentium-IV-2.8</td>
<td>400</td>
<td>800</td>
</tr>
<tr>
<td>Pentium-IV-3.0</td>
<td>400</td>
<td>800</td>
</tr>
</tbody>
</table>

### Platform & Tick-Frequency

![Graph showing delay over minimal phase (%) for different timescales (1μs, 10μs, 100μs, 1ms, 10ms).]

![Bar graph showing L1 and L2 misses at 100Hz and 1000Hz for different timescales (1μs, 10μs, 100μs, 1ms, 10ms).]
Cache misses: user vs. kernel

Different operating systems
Observation

“Periodic ticks are a stupid idea, that’s why K42 doesn’t use them.” ~

Orran Krieger, Jan 2006

The alternative: One-shot timers

• Eliminate periodic ticks
• Set timer only for the next (earliest) event
• Examples:
  – ~K42, Pebble, Pegasus project, ~KURT, ~Firm Timers
  – Common in realtime context
So why do GPOSs use ticks?

• There are a few reasons…

Why use ticks? **Historical Reasons**

• **[Dijkstra’1968]:** First known reference
  “The Structure of the THE Multiprogramming System” in *CACM*

• **Specifies two reasons:**
  – Only way to implement multitasking (back then...)
  – Allows for “exhaustive testing”
    “…[Had we made] the central processor react directly upon any weird time succession of interrupts, the number of relevant states would have exploded to such a height that exhaustive testing would have been an illusion.”

• **Older HW:** **Expensive timer reprogramming**
  – E.g., for Intel, until Pentium-II (1997; APIC since)
Why use ticks? *Inertia*

- This is the way it has always been done…
  - 40 years old design decision
  - All mainstream GPOSs are affected
- Legacy
  - Kernel subsystems rely on it
  - User space too 😎 (e.g. `top`)
- Simplicity
  - Polling is simpler than being event-driven
- It works!
  - If it ain’t broken…

⇒ For a GPOS developer, question is:

  “why NOT??”

(HPC noise is an insufficient incentive)
Why use ticks? **Objective Reasons**

- **Bounds Overhead**
  - Handling clock interrupts
    $\Rightarrow$ Context switch: to kernel & back
  - Ticks aggregate nearby events
- **When taking this argument to the extrem**
  - With one-shot timers
    $\Rightarrow$ **Any user can bring the system down:**

```c
for(int ns=1/*nanosec*/; 1; ns++)
setitimer( shortly + ns );
```

To truly motivate a change in GPOSs should address…

<table>
<thead>
<tr>
<th></th>
<th><strong>Why use ticks?</strong></th>
<th><strong>Counter argument</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no reason not to +</td>
<td>supply sufficient</td>
</tr>
<tr>
<td></td>
<td>hard to change (legacy)</td>
<td>motivation</td>
</tr>
<tr>
<td>2</td>
<td>robustness +</td>
<td>supply a better</td>
</tr>
<tr>
<td></td>
<td>overhead containment</td>
<td>mechanism</td>
</tr>
</tbody>
</table>
Drawback #1: HPC noise

- Dramatic performance loss
  - Noise effect amplified ("noise law")
  - Reports of up to 2/3 performance degradation for real code

Drawback #1: HPC noise

- Two traditional solutions
  1. Designate one core for dealing with noise (app is not allowed to use it)
  2. Synchronize the noise (see next slide)
Synchronizing the noise

Uncoordinated

Coscheduled

Not always appropriate for ticks + only treating the symptom…

Drawback 2#: Desktop Slowdown

<table>
<thead>
<tr>
<th>priority</th>
<th>time stats [usec]</th>
<th>avg slowdown vs. base</th>
<th>vs. min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>base: 1000</td>
<td>avg: 1060, min: 668</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>Realtime: 1080</td>
<td>avg: 778</td>
<td>8%</td>
</tr>
</tbody>
</table>
**Drawback #3: Power consumption**

- Power consumption of an idle laptop
- (In busy a laptop there’s no difference, but run takes longer...)

**Drawback #4: Security breach**
Implement #1: External process

program

fork/exec

cheat client
[default; 1000Hz]

cheat server
[10,000Hz]

SIGSTOP

SIGCONT

sleep epsilon

request msg 80%

select

select

select

SIGSTOP

SIGCONT

sleep epsilon

send msg

request msg 80%

send msg

request msg 80%

One low-end server can “steal” an entire cluster

sum of honest processes [10 per node]

sum of cheaters [one per node] [cheat 60%]
Implement #2: No external process

• Assume app. is an event-driven simulator

```c
start = cycle_counter;

for( i = 0,1, ... )
    now = cycle_counter;
    if( i==0 || (now-start > 0.9tick) )
        sleep epsilon; // wakeup at start of next tick
    start = cycle_counter;
execute_event(i);
```

• Can do this without changing app using binary instrumentation (I’ve done it with pin)

Drawback 5#: VM overhead

• IBM’s S/390

“We are facing the problem that we want to start many (> 1000) Linux images on a big S/390 machine. Every image has its own 100HZ timer.

…

You quickly end up with 100% CPU only for the timer interrupts of otherwise idle images”

Martin Schwidefsky, Apr 2001, LKML
Drawback 6#: poor resolution

- Other soft-realtime apps.:
  - Crash-tests, rate-monotonic net, file-system benchmarking, various sensors, etc.
- SIGMETRICS’03 [“Effects of Clock Resolution…”]:
  - Increasing the tick rate is beneficial but overheads are high.

Solution: Go tickless + bound overhead

- Implementation:
  - Events: quantum-expiration, process-alarms, ...
  - Event time are aligned on Hz boundaries
  - Use one-shot timers (as x86’s APIC)
  - Hz is a settable parameter
    - Hz=0 (pure one-shot), Hz=100 (most GPOSs), Hz>1000 (soft RT)
  - Accounting upon each kernel entry, etc.
- Benefits:
  - Reduced noise (1 runnable process => no ticks)
  - Reduced overheads
  - Faster computation (even for desktops)
  - Reduced power consumption
  - Improved security & robustness
  - Improved resolution (high Hz without useless ticking)
Bottom line

• The general-purpose OS, becomes more general
  – As more applications can enjoy it