the need for a standardized performance monitoring interface

Stéphane Eranian
HP Labs
February 2005
HPCA11, San Francisco, CA - USA

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice
The situation today

- **Hardware situation:**
  - All processors include PMU, more than just counters
  - by nature, PMU is very implementation specific

- **Operating system situation:**
  - All major OS have a specific interface
  - Linux is worse: multiple interfaces available, not integrated
  - large functionality variations between interfaces

- **Monitoring software is diverse:**
  - needs include counting, sampling, per-thread/system-wide
  - used for application tuning, dynamic optimization

- **Portability of tools affected by OS interface variations**
  - user level toolkits (e.g., PAPI) cannot hide everything
why a standard interface?

• Hardware diversity likely to remain
  – can be managed by software packages (e.g. PAPI, PCL)

• OS interface diversity creates an artificial barrier
  – adds useless complexity to monitoring tools
  – slows down large-scale development of tools

• benefits of a standard interface are multiple:
  – uniform level of functionalities across platforms
  – increases chances of adoption by OS distributors, ISVs
  – developer focus on tools and not supporting infrastructure
  – increases code reuse, avoid duplication of effort
  – faster development, smaller learning curve

• added value is in tools/PMU not in OS interface
interface functionality requirements

• built-in, robust, secure, documented, regular user access
• access to all PMU features of present and future CPUs
• generic, flexible, efficient (minimize overhead)
• simple counting, sampling
• per-thread, system-wide
• self-monitoring, unmodified binary, attach/detach
• monitoring of multi-threaded, multi-process workloads
• scale to large NUMA-style machines
• support for existing tools
  — on Linux: Oprofile-based, VTUNE™, HP Caliper, perfctr-based
interface design choices

• exploit common characteristics: register interface

• provide simple read/write operations on registers

• provide uniform view of PMU across platforms:
  − use generic PMU register names, e.g., PMC and PMD
  − export all counters as 64-bit

• focus on PMU access and **NOT** PMU programming
  − PMU specific knowledge in user level libraries (e.g. events)

• avoid kernel bloat
  − no feature integrated unless required for speed or by HW

• use a system-call rather than device driver model
  − built-in, flexible, support for per-thread monitoring
perfmon2 experience

- generic monitoring interface to access PMU
  - implemented in the 2.6 kernel series for Linux/ia64
  - nothing specific to Itanium, ports to other architectures possible

- PMU is key to achieving performance on Itanium
  - performance depends a lot of code quality

- Itanium PMU framework is specified by architecture:
  - up to 256 PMC and 256 PMD, 2 events, start/stop, ovfl. intr.
  - room for extensions within framework:
    - Itanium®: 175 events, 4 counters (32bits), BTB, EARS, range restrictions, opc. match
    - Itanium® 2: 400 events, 4 counters (47bits), BTB, EARS, range restrictions, opc. match

- Itanium is good tesbed for interface
perfmon2 interface

- **perfmon context** encapsulates all PMU state
- Each context uniquely identified by file descriptor
- logical PMU: set of control (PMC) and data (PMD) reg.
- kernel level sampling buffer
- support for event sets and multiplexing

```c
int perfmonctl(int fd, int cmd, void *arg, int narg)
```

<table>
<thead>
<tr>
<th>PFM_CREATE_CONTEXT</th>
<th>PFM_READ_PMDS</th>
<th>PFM_START</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFM_WRITE_PMCS</td>
<td>PFM_LOAD_CONTEXT</td>
<td>PFM_STOP</td>
</tr>
<tr>
<td>PFM_WRITE_PMDS</td>
<td>PFM_UNLOAD_CONTEXT</td>
<td>PFM_RESTART</td>
</tr>
<tr>
<td>PFM_CREATE_EVTSET</td>
<td>PFM_DELETE_EVTSET</td>
<td>PFM_GETINFO_EVTSET</td>
</tr>
<tr>
<td>PFM_GETINFO_PMCS</td>
<td>PFM_GETINFO_PMDS</td>
<td>PFM_GET_CONFIG</td>
</tr>
</tbody>
</table>
challenges for sampling support

• overflow-based sampling needs kernel support
  – period uses counter overflow interrupt mechanism
  – kernel notification on counter overflow

• message-based notification mechanism
  – signal required for self-monitoring

• number of periods = number of counters
  – allows overlapping measurements (e.g., q-tools)

• kernel level sampling buffer for efficiency
  – to amortize cost of notification
  – must provide randomization to avoid biased samples

• customizable buffer format via kernel modules:
  – controls what to record, how to record, how to export
sampling format examples

• want to sample kernel level call stack on cache misses:
  – combine PMU-based sampling with kernel stack unwinder
  – requires zero changes to perfmon2 interface or kernel core
  – new buffer format: about 300 lines of C
  – leverage: PMU programming, buffer remapping, notifications

• OProfile: 20 lines, full reuse of existing implementation

• focus on tool, not kernel support

$ pfmon -e l3_misses --long-smpl-periods=2000 --smpl-periods-random=0xff:10 -k \ 
  --smpl-module=kcall-stack-ia64 --resolve-addr --system-wide
  __copy_user,file_read_actor,do_generic_mapping_read,___generic_file_aio_read, \ 
generic_file_aio_read,do_sync_read,vfs_read,sys_read,ia64_ret_from_syscall
  do_anonymous_page,do_no_page,handle_mm_fault,ia64_do_page_fault, \ 
ia64_leave_kernel

clear_page,do_anonymous_page,do_no_page,handle_mm_fault,ia64_do_page_fault, \ 
ia64_leave_kernel
efficiency: event set support

- PMU limitations:
  - limited number of counters, constraints on events/counters
  - certain measurements may require multiple runs

- solution:
  - create sets of up to n events when PMU has n counters
  - multiplex sets on PMU
  - scale counts (not without danger)

- can be implemented at the user level
  - incurs high overhead because of context switches

- much faster when integrated in the interface:
  - switching occurs within context of monitored thread
  - flexibility: time and overflow-based switching
Conclusions

- Monitoring is key to achieving world-class performance
- PMU is a vital tool to pinpoint performance problems
- PMU hardware is diverse and likely to remain so
- Monitoring tools can really benefit from standardized kernel interface to access PMU
- Linux is a very good testbed for developing interface
- We must unify all Linux interfaces
- perfmon2 interface can serve as a basis for discussion

For PMU HW designers:
- Architect PMU frame in ISA, improve documentation
- More generic counters, faster access to counters
resources

- Itanium Processor Family (IPF) PMU architecture:
  http://developer.intel.com/design/itanium/

- perfmon2 specification:

- PAPI toolkit:
  http://icl.cs.utk.edu/papi

- Oprofile:
  - http://oprofile.sf.net

- VTUNE (Intel):
  http://www.intel.com/software/products/vtune

- Caliper (HP):
  - http://www.hp.com/go/caliper

- Perfctr (Mikael Pettersson)
  - http://user.it.uu.se/~mikpe/linux/perfctr/