perfmon2: a standard performance monitoring interface for Linux

Stéphane Eranian
Agenda

- PMU-based performance monitoring
- Overview of the interface
- Current status
- Challenges ahead
What is performance monitoring?

The action of collecting information related to how an application or system performs

• Information obtained by instrumenting the code
  – extract program-level or system-level information
  – statically: compilers (-pg option), explicit code (LTTng, Xenmon)
  – dynamically (code rewrite): HP Caliper, Intel PIN tool, Kprobes
  – example: count basic-block execution, number of ctxsw/s

• Information obtained from CPU/chipset
  – extract micro-architectural level information
  – exploit hardware performance counters
  – example: count TLB misses, stall cycles, memory access latency
Performance Monitoring Unit (PMU)

- Piece of CPU HW collecting micro-architectural events:
  - from pipeline, system bus, caches, ...
- All modern CPU have a PMU
  - architected for IA-64, AMD64
  - now finally for Intel IA-32 (starting with Core Duo/Solo)
- PMU is highly specific to a CPU implementation
Diversity of PMU HW

- Dual-core Itanium 2: PMC, PMD, 12 counters (47 bits)
  - atomic freeze, opcode filters, range restrictions,
  - where cache/TLB misses occur, Branch Trace Buffer

- AMD64: MSR registers, 4 counters (40 bits)
  - no atomic freeze

- Pentium 4: MSR registers, 18 counters (40 bits)
  - no atomic freeze
  - Precise Event Based Sampling (PEBS)

- Intel Core: MSR registers, 5 counters (31 bits)
  - possible atomic freeze
  - fixed counters, PEBS
Diversity of usage models

• Type of measurement:
  – counting or sampling

• Scope of measurement:
  – system-wide: across all threads running on a CPU
  – per-thread: a designated thread (self-monitoring or unmodified)

• Scope of control:
  – from user level programs: monitoring tools, compilers, MRE
  – from the kernel: SystemTap or VMM

• Scope of processing:
  – offline: profile-guided optimization (PGO), manual tuning
  – online: dynamic optimization (DPGO)
Existing monitoring interfaces

• OProfile (John Levon):
  – included in mainline kernel and most distributions
  – system-wide profiling only, support all major platforms

• Perfctr (Mikael Pettersson)
  – separate kernel patch
  – provides per-thread, system-wide monitoring
  – designed for self-monitoring, basic sampling support
  – supports all IA-32, PowerPC

• VTUNE driver (Intel)
  – open-source driver specific to VTUNE

  no standard and generic interface exists
Why a standard interface?

• Current HW trend makes monitoring capabilities crucial
  − SW must evolve to exploit HW (multi-core, multi-thread, NUMA)
• Strong need for tools to understand SW performance
  − requires portable, flexible kernel-level infrastructure
• Users need portable tools
• Single interface is attractive for tool developers
  − improve code reuse
  − broader market for monitoring products
• Easier to get accepted in mainline kernel
  − no kernel patching, improved support
  − get into commercial distributions
Goals of the perfmon2 interface

• Provides a generic interface to access the PMU
  – designed using a bottom-up approach, no tool in mind
• Be portable across all PMU models/architectures
• Supports per-thread monitoring
  – self-monitoring, unmodified binaries, attach/detach
  – multi-threaded and multi-process workloads
• Supports system-wide monitoring
• Supports counting and sampling
• No special recompilation
• Builtin, efficient, robust, secure, documented
Perfmon2 interface (1)

- Core interface allows read/write of PMU registers
- Uses the \textbf{system call} approach (rather than driver)
- Perfmon2 \textbf{context} encapsulates all PMU state
  \begin{itemize}
  \item each context uniquely identified by file descriptor
  \item file sharing semantic applies for context access
  \end{itemize}
- Leverages existing mechanisms wherever possible
  \begin{itemize}
  \item e.g., file descriptors, signals, \texttt{mmap()}, \texttt{ptrace()}
  \end{itemize}

\begin{verbatim}
int pfm_create_context(pfarg_ctx_t *ctx, char *s, void *a, size_t sz);
int pfm_stop(int fd);
int pfm_restart(int fd);
int pfm_write_pmcs(int fd, pfarg_pmc_t *pmcs, int n);
int pfm_write_pmds(int fd, pfarg_pmd_t *pmcs, int n);
int pfm_read_pmds(int fd, pfarg_pmd_t *pmcs, int n);
int pfm_load_context(int fd, pfarg_load_t *ld);
int pfm_start(int fd, pfarg_start_t *st);
int close(int fd);
int pfm_create_evtsets(int fd, pfarg_setdesc_t *st, int n);
int pfm_delete_evtsets(int fd, pfarg_setdesc_t *st, int n);
int pfm_getinfo_evtsets(int fd, pfarg_setinfo_t *it, int n);
int pfm_unload_context(int fd);
\end{verbatim}
Perfmon2 interface (2)

- Uniformity makes it easier to write portable tools
- Counters are always exported as 64-bit wide
  - emulate via counter overflow interrupt capability if needed
- Exports logical view of PMU registers
  - PMC: configuration registers, write only
  - PMD: data registers (counters, buffers), read-write
- Mapping to actual registers depends on PMU model
  - defined by PMU description kernel module
  - visible in /sys/kernel/perfmon/pmu_desc
Perfmon2 interface (3)

• Same ABI between ILP32 and LP64 models
  – all exported structures use fixed-size data types
  – x86_64, ppc64: 32-bit tools run unmodified on 64-bit kernel

• Vector arguments for read/write of PMU registers
  – portable: decoupled PMC/PMD = no dependency knowledge
  – extensible: no knowledge of # registers of PMU
  – efficient and flexible: can write one or multiple regs per call
Per-thread session

- Thread = kernel visible thread (task)
- PMU state is saved/restored on context switch
  - multiple per-thread sessions can run concurrently
- Support one context per thread
- Thread must be stopped to access PMU state
  - except self-monitoring
- No inheritance across fork/pthread_create
  - `ptrace()` options (`PTRACE_O_TRACE*`) 
  - aggregation done by the tool, if needed
System-wide session

- Monitors across all threads running on one CPU
  - same programming sequence as per-thread
  - type selected when context is created
  - monitored CPU is current CPU in `pfm_load_context()`

- System-wide SMP built as union of CPU-wide sessions
  - flexibility: measure different metrics on different CPUs
  - scalability: strong affinity (processor, cache)
  - ready for HW buffer: Intel PEBS

- Mutual exclusion with per-thread session
Support for sampling

• Supports Event-Based Sampling (EBS)
  − period $p$ expressed as $2^{64} - p$ occurrences of an event
  − number of sampling periods = number of counters

• Can request notification when 64-bit counter overflows
  − notification = message, extracted via `read()`
  − support for `select/poll, SIGIO`

• Optional support for kernel level sampling buffer
  − amortize cost by notifying only when buffer full
  − buffer remapped read-only to user with `mmap()` : zero copy
  − periods can be randomized to avoid biased samples
  − per-counter list of PMDs to record/reset on overflow
Sampling buffer formats

• No single format can satisfy all needs
  – must keep complexity low and extensibility high
• Export kernel interface for plug-in formats
  – port existing tools/infrastructure: OProfile
  – support HW features: Intel PEBS, BTS buffers
• Each format provides at least:
  – string for identification (passed on context creation)
  – counter overflow handler
• Each format controls:
  – where and how samples are stored
  – what gets recorded, how the samples are exported
  – when a user notification must be sent to user
Existing sampling formats

• Default format (builtin):
  – linear buffer, fixed header followed by optional PMDs values

• OProfile format (IA-64, X86)
  – 10 lines of C, reuse all generic code, small user level changes

• N-way sampling format (released separately):
  – implements split buffer (up to 8-way)
  – parsing in one part while storing in another: fewer blind spots

• Kernel call stack format (experimental, IA-64):
  – records kernel call stacks (unwinder) on counter overflow

• Precise Event Based Sampling (P4, Intel Core 2 Duo)
  – 100 lines of C, first interface to provide access to feature!
Event sets and multiplexing (1)

• What is the problem?
  – number of counters is often limited (4 on Itanium®2 PMU)
  – some events cannot be measured together

• Solution:
  – create sets of up to \( m \) events when PMU has \( m \) counters
  – multiplex sets on actual PMU HW
  – global counts approximated by simple scaling calculation
  – higher switch rate \( \Rightarrow \) smaller blind spots \( \Rightarrow \) higher overhead

• Kernel support needed to minimize overhead
  – switching always occur in context of the monitored thread
Event sets and multiplexing (2)

• Each set encapsulates the full PMU state
  – unique identifier: 0-65535
  – sets placed in ordered list
• Switching mode determined per set
• Timeout-based switching
  – granularity depends on kernel timer tick (HZ)
  – actual vs. requested timeout is reported to user
• Overflow-based switching
  – after threshold of $n$ overflows of a counter
  – threshold specified per counter and per set
• Works with counting and sampling
PMU description module

- Logical $\Rightarrow$ actual PMU register mappings
- PMC and PMD mapping description tables
  - type, logical name, default value, reserved bit fields
- Implemented by kernel module:
  - auto-loading on first context creation
  - easier for: support of new HW, maintenance

$ cd /sys/kernel/perfmon/pmu_desc/pmc0; ls; cat *
addr  dfl_val  name  rsvd_msk
0x186 0x100000 PERFEVTSEL0
0x100000
0xffffffff00300000
Security

• Cannot assume tools/users are well-behaved
• Vector arguments, sampling buffers have max. size
  – tuneable via /sys

• Per-thread and system-wide contexts
  – can only attach to thread owned by caller
  – each type can be limited to a users group (via /sys)

• Reading of PMU registers
  – direct access (some arch): limited to self-monitoring
  – interface access: can only read registers declared used

• PMU interrupt flooding
  – need to add interrupt throttling mechanism
Perfmon2 architecture summary

PMU Hardware

- user level
  - sysfs
  - syscalls
  - perfmon arch-specific

PMU description

- kernel level
  - file
    - default
    - n-way
    - kernel-call-stack
    - OProfile
    - PEBS
  - perfmon core
    - res
    - sets
    - ctxsw
    - intr
  - pebs
  - pmu

Default
n-way
kernel-call-stack
OPerf
PEBS
Supported Processors

- Intel Itanium: all processors (HP)
- Intel X86:
  - PIII, Pentium M, Core Duo/Solo, Core 2 Duo (HP)
  - Pentium 4, Xeon (incl. HT) (Intel)
- AMD:
  - family 0x0f (HP)
  - family 0x10 (AMD), incl. Instruction-Based-Sampling (IBS)
- IBM:
  - Power 5 (IBM),
  - Cell (IBM, Sony, Toshiba)
- MIPS: various models (Phil Mucci, Broadcom)
- Cray: BlackWidow (Cray)
Kernel integration status

• Won support from top Linux kernel people
  – with help from performance monitoring community

• Code reviewed 2006 & now by top-level maintainers
  – about 700KB reviewed line by line
  – dozens of changes, improvements

• Why is it taking so long?
  – kernel is a moving target
  – update/fix general kernel infrastructure (ctxsw, NMI, Oprofile)
  – new hardware support, bug fixing, X86 Oprofile co-existence

• target: 26.24 in -mm

• once in mainline, will appear in distros
Current Challenges
Challenges for perfmon2

• Sharing the PMU resource
  – between different subsystems: watchdog, Oprofile, perfmon2
  – between conflicting users: per-thread and system-wide
  – mutual-exclusion is too restrictive, especially on large systems
  – workaround via affinity restriction is invalid

• PMU access in virtualized environments
  – PMU usage is never for correctness but for performance
  – usage model evolving: from development to always on
  – used by monitoring, tools, managed runtimes, OS kernels
PMU sharing: what?

- PMU state to share:
  - data/config registers (dependencies)
  - interrupt vector (unique)
  - possibly start/stop controls

- Sharing consequences:
  - symmetrical register functionalities
  - independent start/stop, freeze
  - tools must be prepared to use partial PMU
PMU sharing: example

- perfmon2, CPUs=0,1
- NMI, CPUs=all
- perfmon2, CPUs=all

Kernel:
- perfmon2
- NMI watchdog
- E1, all CPUs

User:
- sys
  - E1, CPUs=0,1
  - E1, all CPUs
- thread
  - E2, my thread

Config:
- 0
- 1
- 2
- 3

Data:
- 0
- 1
- 2
- 3

sys
all CPUs
my thread
E1
Usage models in virtual environments

• Ensure continuity of service: PMU virtualization
  – OS, applications using PMU must continue to work
  – Performance must be maintained: JVM with DPGO
  – must provide PMU access to guest
  – no visibility into VMM execution

• Assessment global performance: system-wide
  – measure across hypervisor (VMM) and guest environments

• Must deal with multiple virtual machines
  – work with VT-*/AMD-V and para-virtualization
  – Xen (para): XenOprofile
  – KVM, lguest
Perfmon & petaflops computing

• How do you know effective FLOPS?
  – guess by looking at the code?
  – instrumentation does now work: must use HW counters

• PMU Metrics for scientific code:
  – Flops
  – Cache behavior
  – Bus bandwidth utilization
  – profiles to identify key loops

• Some metrics unavailable or unreliable
  – e.g.: no FLOPS on AMD64

• Need to identify key metrics to influence future HW
Summary

• Monitoring key to achieve world-class performance
  – current HW trend makes this critical
• Perfmon2 is a very advanced monitoring interface
  – supports all major processor architecture
• Perfmon2 to become the Linux monitoring interface
  – strong community of users/developers
• Need to solve sharing/virtualization challenges
• Call to action: try it out!
  – start porting/developing performance tools
  – visit http://perfmon2.sf.net
Basic self-monitoring per-thread session

```c
pfarg_ctx_t ctx; int fd;
pfarg_load_t load;
pfarg_pmd_t pd[1]; pfarg_pmc_t pc[1];
pfmlib_input_param_t inp;
pfmlib_output_param_t outp;...
pfm_find_event("CPU_CYCLES", &inp.pfp_events[0]);
inp.pfp_plm = PFM_PLM3; inp.pfp_count = 1;
pfm_dispatch_events(&inp, NULL, &outp);
pd[0].reg_num = out.pfp_pd[0].reg_num;
pc[0].reg_num = outp.pfp_pc[0].reg_num;
fd = pfm_create_context(&ctx, NULL, 0, 0);
pfm_write_pmcs(fd, pc, 1);
pfm_write_pmds(fd, pd, 1);
load.load_pid = getpid();
pfm_load_context(fd, &load);
pfm_start(fd, NULL);
/* run code to measure */
pfm_stop(fd);
pfm_read_pmds(fd, pd, 1);
printf("total cycles %"PRIu64"
", pd[0].reg_value);
close(fd);
```