perfmon2: a performance monitoring interface for Linux

Stéphane Eranian
HP Labs
January 2005
CERN, Geneva, Switzerland

Agenda

- What is performance monitoring?
- What is the PMU?
- Overview of the perfmon2 interface
- Current implementations
- porting to Xen/ia64
- Examples of performance tools for Linux/ia64

What is performance monitoring?

- The action of collecting information related to how an application/system performs when executing.
- Information obtained by instrumenting the code
  - Extract program-level information
  - Statically: by compilers (-pg option)
  - Dynamically: e.g., HP Caliper, Intel PIN tool
  - example: count basic-block execution
- Information obtained from processor/chipsets
  - Extract micro-architectural level information
  - Uses hardware performance counters
  - Example: count TLB misses

What is the PMU?

- Piece of CPU HW collecting micro-architectural events:
  - From pipeline, system bus, caches, ...
- All modern processors have a PMU
  - May even be part of the architecture, e.g., Itanium®
- PMU has existed for a long time (think debug)
  - Not always made public or documented properly
- PMU is highly specific to processor implementation
  - Large differences even inside same processor family
- New trend is to expose PMU to users
  - Foster developments of good performance tools
- Many new PMUs go beyond just collecting counts

Performance monitoring and IPF

- IPF performance is based mostly on code quality
  - EPIC: parallelism of the machine is exposed to users
- Optimization decisions made at compile time
  - Must extract as much parallelism as possible from source
- Performance feedback needed by compilers
  - Profile Guided Optimization (PBO) to tweak optimizations
  - Static optimization
- Performance feedback needed by Managed Runtimes (MRE)
  - Needed to tweak embedded JIT compiler
  - Dynamic optimization
- Must have very good monitoring infrastructure
  - Need access to low-level performance informatio

The Itanium® PMU

- IPF architecture specifies PMU interface (framework):
  - Up to 256 control (PMC) and 256 data (PMD) registers
  - Minimal config: 4 counters, 2 events, overflow intr. capability
- Lots of room for extensions:
  - Itanium®: 14 PMC, 18 PMD
    - 4 counters (32bits), ≈230 events
    - Opcode match, range restrictions, D-EAR, I-EAR, BTB
  - Itanium® 2: 16 PMC, 18 PMD
    - 4 counters (47bits), ≈475 events
    - Opcode match, range restrictions, D-EAR, I-EAR, A-EAR, BTB
  - Montecito(2005): expect more exciting features

Accessing the PMU

- Some operations require priviledged access
  - e.g.: processing of PMU interrupts, setup of PMU registers
- Some PMUs allow certain operations at user level:
  - Itanium®: read PMD, start and stop with simple instructions
- OS support required: device driver or system call?
  - System call: makes it a builtin feature
  - Device driver: makes it more modular and optional
  - System call: HPUX, Linux,MacOS (per-thread and syswide)
  - Device driver: Windows (syswide)

The perfmon challenge

- No standard kernel interface exist on Linux
  - Various patches exist for IA-32, PowerPC, X86_64
  - Most interesting is perfctr
  - Other OS may have proprietary interfaces
- Slows down developments of modern tools
  - Unexploited hardware resources to help boost performance
- PMU is specific to each processor implementation
- Huge variations make it difficult to abstract hardware
- Challenge:

  How to design a generic, yet powerful and extensible, kernel interface to access the PMU of modern processors which could support a variety of performance tools?

The perfmon2 interface

- Provides a generic interface to access PMU
  - Not dedicated to one app, avoid fragmentation
- Must be portable across all PMU models:
  - Almost all PMU-specific knowledge in user level libraries
- 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 modification to applications or system
- Builtin, efficient, robust, secure, simple,documented

Perfmon2 interface

- Uses a system call
  - More fexibility, ties with ctxsw, exit, fork
  - Kernel compile-time option on Linux
- Perfmon2 context enscapsulates all PMU state
  - Each context uniquely identified by file descriptor

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

PFM_CREATE_CONTEXT	PFM_READ_PMDS	PFM_START
PFM_WRITE_PMCS	PFM_LOAD_CONTEXT	PFM_STOP
PFM_WRITE_PMDS	PFM_UNLOAD_CONTEXT	PFM_RESTART
PFM_CREATE_EVTSET	PFM_DELETE_EVTSET	PFM_GETINFO_EVTSET
PFM_GETINFO_PMCS	PFM_GETINFO_PMDS	PFM_GET_CONFIG
PFM_SET_CONFIG		

Perfmon2 PMU registers

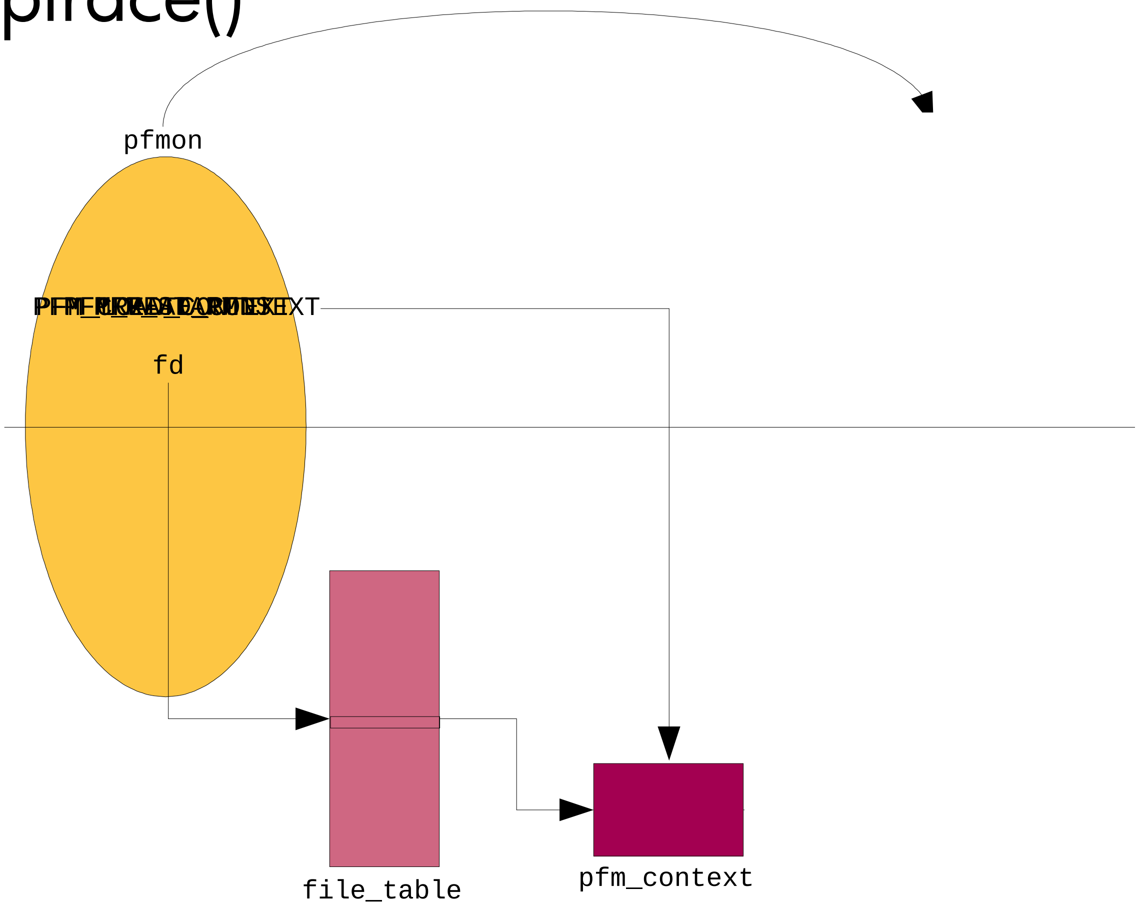
- Logical PMU registers exposed by interface:
  - PMC: configuration registers
  - PMD: data registers (counters, buffers, ...)
- Counters are always exported as 64-bit wide
- Mapping to actual registers depends on PMU
- Mapping returned by PFM_GETINFO_PM[CD]S
  - Calls return actual register name and index or address
  - Example: PMC4 = MSR @ 0x300
- Possibility to have virtual PMD registers
  - Can map to OS or processor resource
  - Example: PMD356 = amount of free physical memory

Typical self-monitoring session

```
pfarg_ctx_t ctx;
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 = pc[0].reg_num = outp.pfp_pc[0].reg_num;
perfmonctl(0,PFM_CREATE_CONTEXT, &ctx,1);
perfmonctl(ctx.ctx_fd, PFM_WRITE_PMCS, pc, 1);
perfmonctl(ctx.ctx_fd, PFM_WRITE_PMDS, pd, 1);
load.load_pid = getpid();
perfmonctl(ctx.ctx_fd, PFM_LOAD_CONTEXT, &load, 1);
perfmonctl(ctx.ctx_fd, PFM_START, NULL, 0);
/* run code to measure */
perfmonctl(ctx.ctx_fd, PFM_STOP, NULL, 0);
perfmonctl(ctx.ctx_fd, PFM_READ_PMDS, pd, 1);
printf("total cycles %"PRIu64"\n", pd[0].reg_value);
close(fd);
```

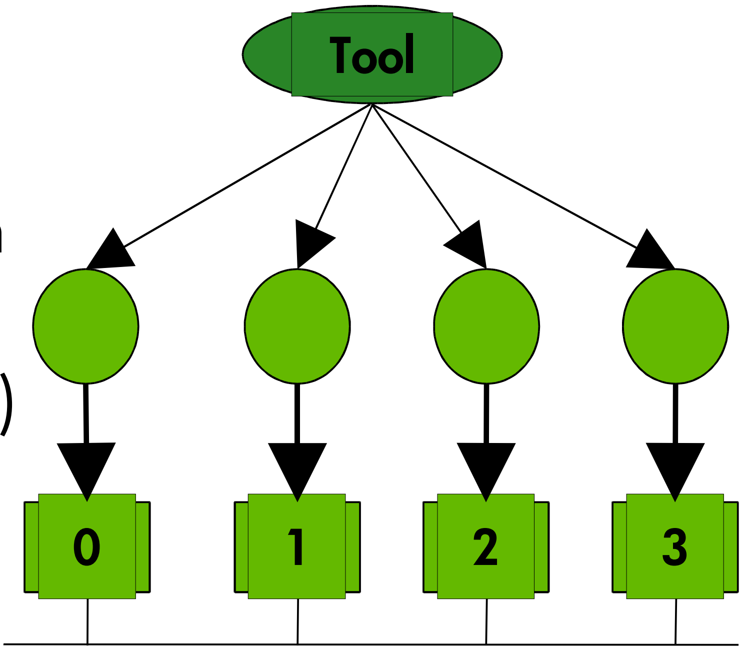
Monitoring an unmodified binary

- Can fork/exec binary or attach to a running thread
- Ability to follow across fork/pthread_create using ptrace()



System wide monitoring

- Monitor across processes
- Built as union of cpu-wide sessions
  - Simplicity of kernel implementation
  - Better scalability
  - Better atune to hardware (P4 PEBS)
  - Use sched_setaffinity() for pinning
- Ability to exclude idle task
- Cannot run concurrently with per-thread session



Perfmon2 event notification

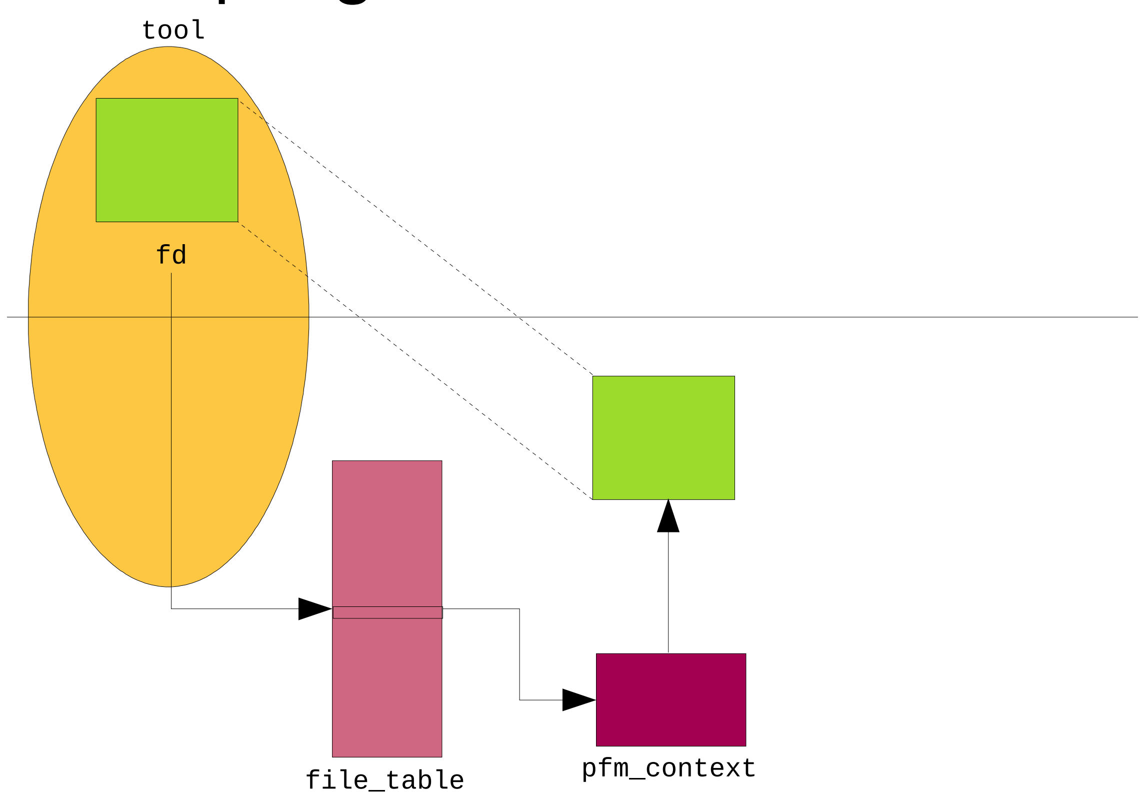
- Can receive a message on:
  - A counter overflow: when it wraps from 2^{64} to 0
  - a thread termination
- Message channel is a simple queue
- Exploit existing file infrastructure:
  - Extraction via read()
  - Support for select/poll to poll on multiple descriptors
  - Asynchronous notification via signal (SIGIO)
- Tuneable behavior on overflow notification
  - Monitoring is stopped, resumed with PFM_RESTART
  - Possibility to block monitored thread to limit blind spots

Support for sampling

- Support time-based sampling from user level
- Support for Event-Based Sampling (EBS) in kernel
  - Sampling period p expressed as 2^{64}-p occurrences of event
- As many sampling periods as there are counters
  - Allows overlapping sampling measurements
- Support for randomized sampling period
  - Very important to avoid avoid biased samples
  - setup is per counter
- Suport optional kernel level sampling buffer
  - amortize cost of overflow notification
  - Samples stored in kernel buffer, notification when buffer full

Kernel level sampling buffer

- Buffer remapped into user level address space
  - Avoid large data copies
  - Remapped read-only via an mmap() call
- support custom sampling formats via kernel modules

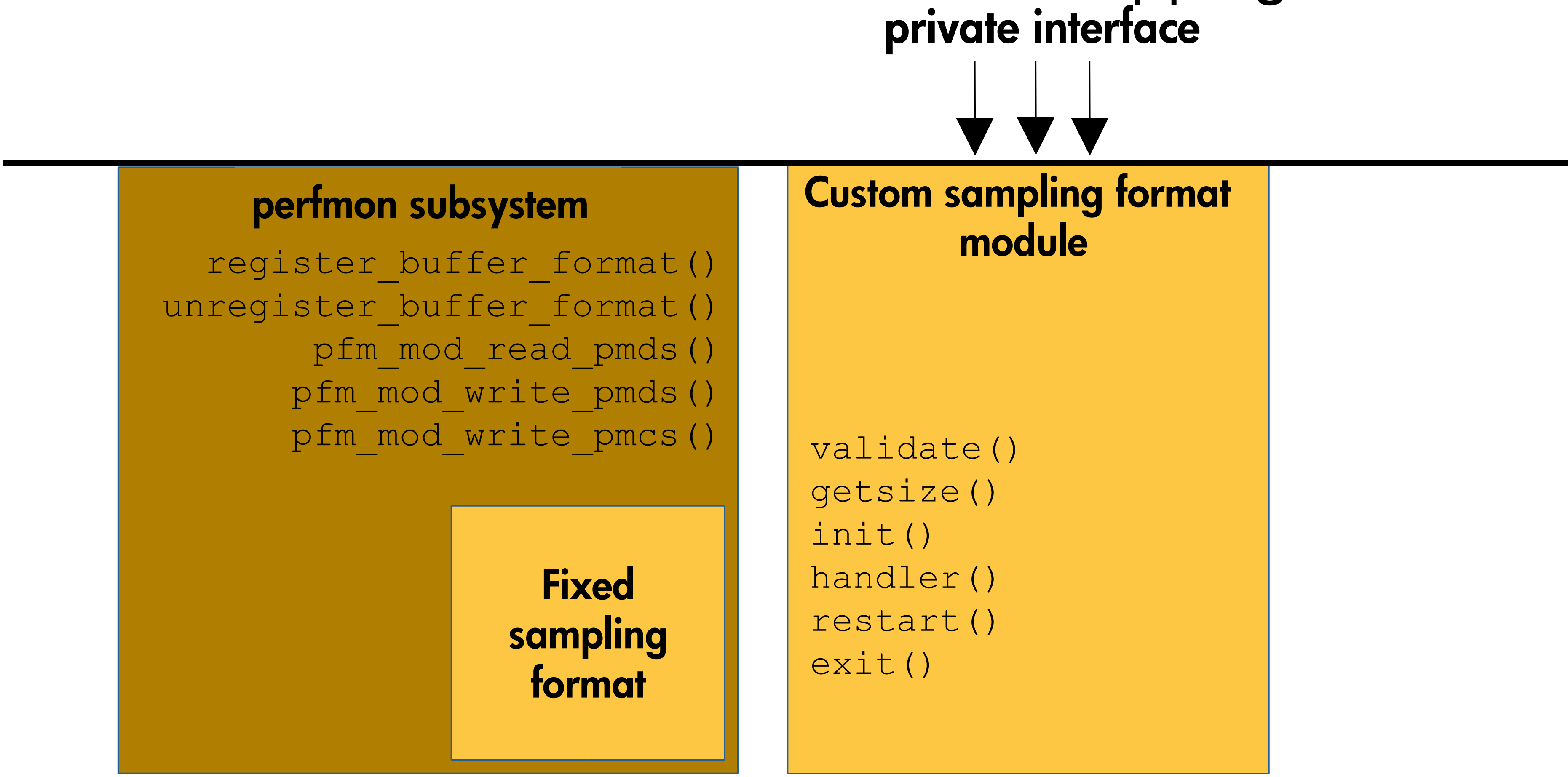


Custom sampling buffer formats

- No single format can satisfy all needs
  - Keep complexity very low
- Provides interface for plug-in formats:
  - Easier to port existing tools, e.g., Oprofile or VTUNE
  - Exploit kernel infrastructure: kernel modules
- Each format provides:
  - A 128-bit UUID for identification
  - A handler function called on each counter overflow
- Each format controls:
  - Where and how samples are stored
  - What gets recorded, how the samples are exported
  - When a “buffer full” condition is declared

Custom sampling format infrastructure

- Modules may have private interface to export data
- Modules do not have to use buffer remapping service



Existing sampling formats

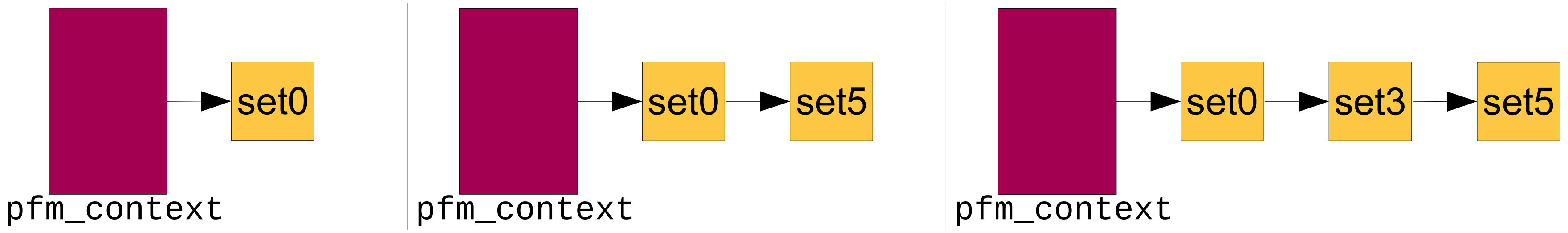
- Default format (builtin):
  - Simple linear buffer
  - Very generic samples: fixed header + PMD in body
  - Samples stored sequentially
- Oprofile format:
  - 10 lines of codes, reuse 100% of existing code
- n-way sampling format (released separately):
  - Implements split buffer (up to 8-way)
  - Process one part while storing in others: minimize blind spots
- Kernel call stack format (experimental):
  - Combines PMU sampling with kernel stack unwinder
  - Record kernel call stacks on counter overflow

Event sets and multiplexing

- What is the problem?
  - Number of counters is always limited (4 for Itanium®2)
  - Some events cannot be measured at the same time
  - Some measurements require a lot of events:
    - Example: cycle breakdown on Itanium®2 requires at least 15 events
- Solution:
  - Create sets of up to m events when PMU has m counters
  - Time share PMU between sets

Event sets

- Each set encapsulates the full PMU state
  - All PMC and PMD registers
- Each set is identified by user-specified unique number
  - Up to 65k sets are supported
  - set0 created by default (cannot be removed)
- Only one set can be active at a time
- Sets can dynamically be added, modifed, removed
- Sets are ordered based on their unique number
  - order determines the switching order



Event sets (cont'd)

- Runtime information about a set:
  - Use PFM_GETINFO_SETS
  - Infos: number of activations, aggregated duration of activation
- System-wide per-set modes:
  - Exclude idle task execution
  - Exclude interrupt-triggered execution (Itanium® only)
  - Exclude all but interrupt-triggered execution (Itanium® only)

Set multiplexing

- List of sets managed in round-robin fashion
- Two modes of switching: timeout or overflow
  - Selected per set, can mix and match
- Timeout-based switching:
  - Timeout specified per set
  - granularity depends on OS timer (Linux/ia64 = 1ms)
- Overflow-based switching:
  - after n overflows of a “trigger” counter
  - Multiple simultaneous triggers are supported
- Possibility to build cascading counters
  - Activate a set of counters after a certain threshold is reached

Linux/ia64 perfmon implementations

- In Linux/ia64 since 2.4.0
- In all 2.4-based kernels: perfmon1
  - First generation interface
  - Included in SLES-8, RHAS-2.1, RHEL-3.0 (but broken)
  - Several limitations : no monitoring across fork()
- In all 2.6-based kernels: perfmon2
  - Second generation interface
  - Included in SLES9 and RHAS4
  - Not backward compatible with perfmon-1
  - Currently includes: sampling formats
  - Event set support not yet public

Porting perfmon2 to Xen/ia64

- Two possibilities:
  - port to guest OS (XenoLinux/ia64)
  - port to hypervisor with Domain0 as controller
- Port to XenoLinux/ia64
  - monitor each domain separately
  - easier because familiar environment
  - ring0 vs ring1 issues
- Port to hypervisor
  - allow cross-domain monitoring
  - non Linux-environment
  - issues: memory allocation, interrupt, file descriptor intf., memory remapping

Porting perfmon2 to XenoLinux

- Ring 1 vs. ring 0 issues:
  - mov to/from pmd[]/pmc[]
  - toggling of psr.pp and psr.up
  - toggling of dcr.pp
- PMU interrupt:
  - managed as asynchronous external device interrupt
  - reuse Xen I/O descriptor ring (Xen -> XenoLinux only)
- PMU state must be saved & restored on domain switch

Linux/ia64 monitoring tools

- Caliper(HP):
  - Per-thread monitoring, binary product, free download
  - Source level profiles
- VTUNE(Intel) for Linux/ia64
  - PMU-based, system-wide flat profile, Windows-side GUI
- OProfile for Linux/ia64
  - PMU-based, system-wide flat profile
- PAPI toolkit (U. of Tenessee) for Linux/ia64
  - PMU-based, counting, sampling, uses libpfm
- pfmon/libpfm (HPLabs) for Linux/ia64
- q-tools, qprof (HPLabs) for Linux/ia64

Monitoring complicated workloads

- Implemented with pfmon-3.0 for perfmon-2:
  - Can follow across fork/vfork and pthread_create
  - Works for counting and sampling
  - Supports regular expression to filter binaries of interest
- Example: elasped cycles of a compilation

```
$ pfmon --us-c -u -k --follow-all -ecpu_cycles,ia64_inst_retired \
  -- cc e.c -o e

 1,164,772   CPU_CYCLES        /usr/lib/gcc-lib/ia64-linux/2.96/cpp0
 1,295,480   IA64_INST_RETIRED /usr/lib/gcc-lib/ia64-linux/2.96/cpp0
13,758,346   CPU_CYCLES        /usr/lib/gcc-lib/ia64-linux/2.96/cc1
21,863,635   IA64_INST_RETIRED /usr/lib/gcc-lib/ia64-linux/2.96/cc1
 5,708,731   CPU_CYCLES        as
 7,165,599   IA64_INST_RETIRED as
27,046,535   CPU_CYCLES        /usr/bin/ld
35,247,760   IA64_INST_RETIRED /usr/bin/ld
 1,381,134   CPU_CYCLES        /usr/lib/gcc-lib/ia64-linux/2.96/collect2
 1,508,977   IA64_INST_RETIRED /usr/lib/gcc-lib/ia64-linux/2.96/collect2
 1,913,253   CPU_CYCLES        cc
 1,976,590   IA64_INST_RETIRED cc
```

Detailed cycle breakdown

- Can use current pfmon with wrapper script
  - i2prof.pl written by Per Ekman
- Using the experimental version of pfmon:

```
$ pfmon -m itanium2-stalls -ku –system-wide –print-interval – mcf inp.in

# ---------------------------------------------------------------------------------
# %itlb %icache %bra   %unstall %BE    %score %RSE   --------------- D-access ------
#                      exec     flush  board         %d1tlb %d2tlb %cache -loaduse-
#                                                                  res    %gr    %fr
# ---------------------------------------------------------------------------------
   0.00 0.02    2.81   32.08    10.06 1.19    0.00   0.57   5.28   4.19   43.80 0.00
   0.00 0.02    2.81   32.12    10.06 1.19    0.00   0.57   5.28   4.19   43.77 0.00
   0.00 0.02    2.81   32.09    10.06 1.19    0.00   0.57   5.28   4.19   43.78 0.00
   0.00 0.00    0.08   59.29     0.22 0.05    0.00   0.03   0.01   1.75   38.57 0.01
   0.00 0.00    0.06   54.49     0.16 1.16    0.00   0.46   3.16   3.74   36.76 0.00
   0.00 0.05    2.83   42.14    10.08 1.06    0.02   0.68   4.77   5.69   32.69 0.00
   0.00 0.05    2.79   42.27     9.97 1.07    0.02   0.69   4.88   5.67   32.59 0.00
   0.00 0.03    2.44   41.42     8.74 1.11    0.00   0.55   4.30   4.32   37.09 0.00
   0.00 0.02    2.82   32.07    10.07 1.16    0.00   0.62   5.69   4.46   43.08 0.00
```

Opcode matching with pfmon

- Constrains monitoring to instructions or patterns
  - Based on opcode, e.g., st8.*
  - Based on functional unit, e.g., M,F,I,B
  - Pattern uses a match+mask fields
  - Not all instructions can be uniquely identified
  - Two opcode matching registers on Itanium® 1 & 2
- Ex.: counting the number of br.cloop instructions:

```
$ pfmon -us-c --opc-match8=0x1400028003fff1fa \
   -e IA64_TAGGED_INST_RETIRED_IBRP0_PMC8 -- foo
          4,999,950,164 IA64_TAGGED_INST_RETIRED_IBRP0_PMC8
```

Range restrictions

- Constrains monitoring to range of data or code
  - Implemented via debug registers (not used as breakpoints)
  - Can specify a range inside the kernel (Linux/ia64)
  - Works for both per-process and system-wide
  - Not all events support range restrictions
- Range must be aligned on size for exact measurements
  - gcc -falign-functions= option can be useful
- Ex.: how many L2 misses while executing init_tab()

```
$ pfmon –us-c –el2_misses -- foo
          1,245,516 L2_MISSES (misses for the entire execution)
$ pfmon –us-c –irange=init_tab –el2_misses -- foo
             14,456 L2_MISSES (misses for init_tab() only)
```

Sampling cache and TLB misses (EARS)

- Very useful to find where cache/TLB load misses occur
  - Cannot be done with naïve IP-based sampling
- Pinpoint the source of a miss, not the consequence
  - Careful because not all misses lead to stalls
- Ex.: sample every 1000 cache misses with latency > 4 cycles

```
$ pfmon --long-smpl-periods=1000 -edata_ear_cache_lat4 - foo

  entry 2000 PID:608 CPU:0 STAMP:0xfe3e1212e5 IIP:0x4000000000000990
         accessed data: 0x2000000000357000
         miss latency : 16 cycles
         inst address : 0x4000000000000981

  4000000000000980:         [MMI]         ld8 r15=[r16]
  4000000000000981:                       ld8 r14=[r17]← miss source
  4000000000000982:                       nop.i 0x0;;
  4000000000000990:         [MMI]         cmp.ltu p7,p6=r14,r15;; ← stall
```

Data load cache misses profiles

- Obtained using the Data EARS
- Provides two views:
  - Instruction view: which loads trigger misses?
  - Data view: on which data do misses occur?
- Example: mcf instruction and data views

```
#count %self    %cum    %L2    %L3   %RAM instruction addr
 6358  11.11% 11.11%  3.05%  5.17% 91.77% price_out_impl+0x820<mcf>
 6238  10.90% 22.01% 26.74% 69.93%  3.33% price_out_impl+0x850<mcf>
 5404   9.44% 31.45% 74.43% 24.94%  0.63% bea_compute_red_cost+0x50<mcf>
 5016   8.77% 40.22% 46.69% 33.77% 19.54% bea_compute_red_cost+0xa1<mcf>
 4968   8.68% 48.90% 42.43%  9.98% 47.58% primal_bea_mpp+0x7b1<mcf>
 4878   8.52% 57.42% 36.67% 51.87% 11.46% bea_compute_red_cost+0x90<mcf>
```

```
#count  %self    %cum    %L2    %L3   %RAM data addr
   37   0.06%   0.06% 62.16% 32.43%  5.41% 0x200000000017ebd0
   32   0.06%   0.12% 75.00% 18.75%  6.25% 0x20000000000d07b0
   29   0.05%   0.17% 68.97% 24.14%  6.90% 0x20000000000e2438
   28   0.05%   0.22% 96.43%  3.57%  0.00% 0x20000000000d3708
   26   0.05%   0.27% 88.46% 11.54%  0.00% 0x20000000000d8c58
```

Sampling branches (BTB)

- Capture up to the last 4 branches:
  - Each entry contains source/target addr., prediction outcome
  - Possible to filter branches: taken/not taken, mispredicted
  - Can be combined with EAR to build a path to a cache/tlb miss
- Ex.: sample every 1000 taken branch, record last 4

```
$ pfmon --smpl-periods-random=5:0xff --btb-tm-tk \
   --long-smpl-periods=1000 -ebranch_event -- foo

  entry 231 PID:673 CPU:0 STAMP:0x12957325ac49 IIP:0x40000000000004d0
     last reset : 1004
     branch source address: 0x40000000000004f2
     branch target address: 0x40000000000004c0
     branch taken : yes, prediction: success, pipe flush: no
     ...
  40000000000004f0:[MFB]         nop.m 0x0
  40000000000004f1:              nop.f 0x0
  40000000000004f2:              br.cloop.sptk.few 40000000000004c0
```

Current and future work

- Full interface specification document
  - To be released as HPLabs tech report in February 2005
- Engage in discussion with Linux community to standardize performance monitoring interface
- Ensure SLES9/RHEL4 have decent perfmon2 support
  - Important for HP and Intel and entire user community
- Open-source event set multiplexing support
- Update pfmon/libpfm for Montecito support
- Develop new kinds of perf. tools exploiting the interface

Kernel level call stack sampling

- Combines kernel stack unwinder with perfmon2:
  - On counter overflow, record the call stack
  - Uses a custom sampling buffer format
- Example using the modified version of pfmon:

```
$ pfmon -el3_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

bh_lru_install,__find_get_block,__getblk,ext3_get_inode_loc,ext3_reserve_inode_write,
ext3_mark_inode_dirty,ext3_dirty_inode,__mark_inode_dirty,update_atime,link_path_walk,open_namei,
filp_open,sys_open,ia64_ret_from_syscall

end_bio_bh_io_sync,bio_endio,__end_that_request_first,scsi_end_request,scsi_io_completion,
sd_rw_intr,scsi_finish_command,scsi_softirq,do_softirq,ia64_handle_irq,ia64_leave_kernel

filemap_nopage,do_no_page,handle_mm_fault,ia64_do_page_fault,ia64_leave_kernel

scsi_finish_command,scsi_softirq,do_softirq,ia64_handle_irq,ia64_leave_kernel

end_page_writeback,end_buffer_async_write,end_bio_bh_io_sync,bio_endio,__end_that_request_first,
scsi_end_request,scsi_io_completion,sd_rw_intr,scsi_finish_command,scsi_softirq,do_softirq,
ia64_handle_irq,ia64_leave_kernel
```

Conclusions

- Monitoring is key to achieving world-class performance
- Having a standardized perfmon interface is important
- Perfmon2 is the most advanced monitoring interface of all Linux implementations
- The Itanium® 2 PMU is very powerful
- Linux/ia64 already has a variety of performance tools
- Need to develop better, smarter tools for non-experts

PMU resources

PMU resources

- pfmon/libpfm, q-tools, q-prof (HPLABS)
  - http://www.hpl.hp.com/research/linux
- Caliper(HP):
  - http://www.hp.com/go/caliper
- VTUNE(Intel):
  http://ww.intel.com/software/products/vtune
- PAPI
  http://icl.cs.utk.edu/projects/papi
- OProfile
  http://oprofile.sf.net
- Prospect:
  http://prospect.sf.net

Linux/ia64 perfmon resources

- i2prof.pl:
  http://www.pdc.kth.se/~pek/i2prof.pl
- IPF PMU architecture:
  http://developer.intel.com/design/itanium/
- Itanium® 2 PMU specification:
  http://developer.intel.com/design/itanium/manuals.htm
- N-way sampling buffer format:
  ftp://ftp.hpl.hp.com/pub/linux-ia64/nway_smpl-0.1.tar.gz

Backup slides

i n v e n t