An OS-oriented performance monitoring tool for multicore systems

J.C. Sáez, J. Casas, A. Serrano, R. Rodríguez-Rodríguez, F. Castro, D. Chaver, M. Prieto-Matias

Department of Computer Architecture
Complutense University of Madrid

3rd Workshop on Runtime and Operating Systems for the Many-core Era (ROME 2015)

August 25, 2015

Performance monitoring counters

- Most modern complex computing systems are equipped with hardware Performance Monitoring Counters (PMCs)
- High-level performance metrics collected via PMCs provide valuable hints to programmers and computer architects
  - IPC, Last-Level Cache (LLC) miss rate, ...
- Direct access to PMCs is typically restricted to code running at the OS privilege level
  - Kernel-level tools enable users to access PMCs
  - Low-level access to PMCs is tedious

PMCs and the OS scheduler (I)

- *The OS scheduler can leverage PMCs to perform effective optimizations in modern CMPs*
  - **Symmetric CMPs**: [Tam et al., Eurosys'07], [Knauerhase et al, IEEE Micro (2008)], [Saez et al., ICPP'08], [Zhuravlev et al., ASPLOS'10], [Merkel et. al, Eurosys'10], [Zhuravlev et al., PACT'11] ...
  - **Asymmetric CMPs**: [Koufaty et. al, Eurosys'10], [Saez et. al, Eurosys'10], [Petrucci et al., ACM TECS, (2015)], [Saez et. al, ACM SAC'2015],...

Overall Idea:

1. OS characterizes application behavior online using PMCs
2. Perform thread-to-core mappings to optimize a certain metric

PMCs and the OS scheduler (II)

Unfortunately...

- Current public-domain tools do not feature a specific in-kernel API to aid in implementing such OS scheduling schemes
  - Fully user-space oriented
  - Designed that way from the ground up
- Researchers' workarounds (not suitable for production use)
  1. Simplistic user-space scheduling prototypes
  2. Write platform-specific low-level code to deal with PMCs within the scheduler

The PMCTrack performance monitoring tool

PMCTrack

- Project started in 2007
  - It provided access to PMCs from the scheduler code only
  - Versions for the Linux kernel and Solaris (proprietary)
- Today, it is an open-source tool for the Linux kernel (GPL v2)
  - Performance monitoring information can be gathered from user space and from the OS scheduler's code
  - Other monitoring information beyond HW PMC events:
    - Energy/Power consumption readings (Intel/ARM)
    - Last-level cache usage (Intel Cache Monitoring Technology)

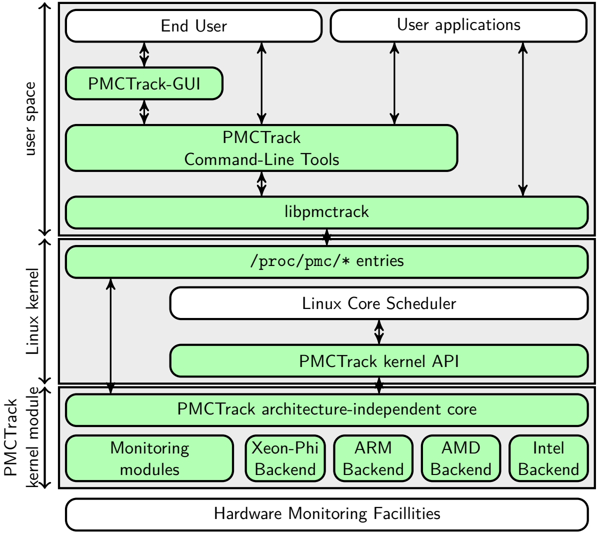
Outline

1. **Introduction**
2. **PMCTrack architecture and usage modes**
3. **Case studies**
   - OS Scheduling for AMPs
   - Cache usage monitoring: Intel CMT
   - Monitoring power/energy consumption
4. **Conclusions**

Outline

1. Introduction
2. **PMCTrack architecture and usage modes**
3. Case studies
   - OS Scheduling for AMPs
   - Cache usage monitoring: Intel CMT
   - Monitoring power/energy consumption
4. Conclusions

PMCTrack architecture



PMCTrack's kernel-level components

Kernel-level components

1. PMCTrack kernel module
   - Implements almost all the entire kernel-level functionality
     - Low-level access to HW monitoring facilities
     - /proc-based interface with user-space components
   - Modular design (*monitoring modules*)
2. PMCTrack kernel API (kernel patch)
   - Code issues notifications to PMCTrack kernel module
     - context-switches, thread creation/termination, ...
   - Changes can be easily applied to different kernel versions
     - 2 new source files
     - ~20 extra lines of code in existing files (x86)

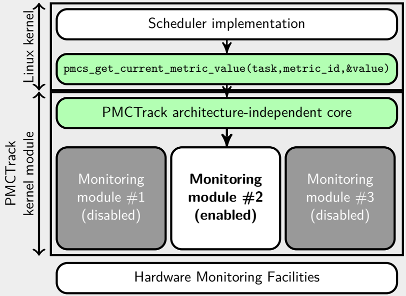
PMCTrack monitoring modules (I)

- A monitoring module (MM) is a *"plug-in"* whose code lives in PMCTrack's loadable kernel module
  - Each MM implements a set of callback functions (notifications)
- Only one MM can be active at a time
  - Administration of MMs via /proc/pmc/mm_manager
- A MM may take full control of PMCs and configure them using an architecture-independent mechanism
  - MM code accesses performance counters indirectly via API calls

PMCTrack monitoring modules (II)

Any monitoring module may...

1. Provide the OS scheduler with per-thread performance metrics



2. Expose any kind of monitoring information as *virtual counters* to user space components or even to the OS scheduler
   - Example: measured or predicted power consumption

Using PMCTrack from user space

Usage modes

1. **Time-Based Sampling (TBS)**
   - An application's PMC and virtual counter values are collected at regular time intervals
2. **Time-Based system-wide monitoring mode**
   - TBS for each CPU in the system
3. **Event-Based Sampling (EBS)**
   - An application's PMC and virtual counter values are collected when a given HW event counter reaches a certain count
4. **Self-monitoring mode** (instrumentation with *libpmctrack*)
   - Retrieve PMC and virtual counter values for specific code fragments

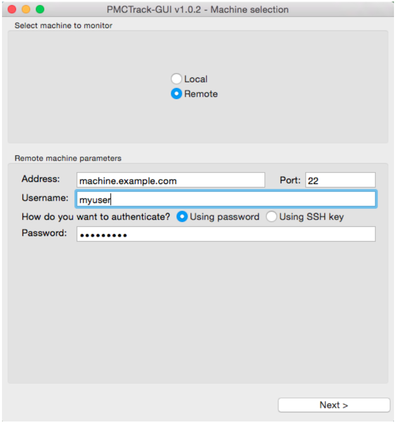
The pmctrack command-line tool (TBS)

TBS with pmctrack

```
$ pmctrack  -T 1  -c instr,cycles,llc_misses -V energy_core ./mcf06
[Event-to-counter mappings]
pmc0=instr
pmc1=cycles
pmc3=llc_misses
virt0=energy_core
[Event counts]
nsample    pid      event          pmc0          pmc1          pmc3         virt0
      1   5051       tick    2031752774    3278225927      28076472       6816345
      2   5051       tick    1220553549    3581105680      24851799       7674194
      3   5051       tick    1200669666    3579946939      24758056       7533020
      4   5051       tick    1439111649    3377276912      22649829       8372497
      5   5051       tick    1741678429    2910646974       7125557       9042236
      6   5051       tick    2288054908    3591634920      19342428       6588195
      7   5051       tick    2427548635    3593134689      18843632       6766113
      8   5051       tick    1387333303    3592647444      22690759       6272949
      9   5051       tick    1451673704    3593864932      22313698       6244079
     10   5051       tick    1331258605    3593793009      22677829       6211608
     11   5051       tick    1323855919    3593486094      22600530       6065124
     12   5051       tick    1352018668    3592025587      22390828       6119812
     13   5051       tick    1327291415    3552079221      21806709       6572875
     14   5051       tick    1292799158    3584908606      21796200       6203979
...
```

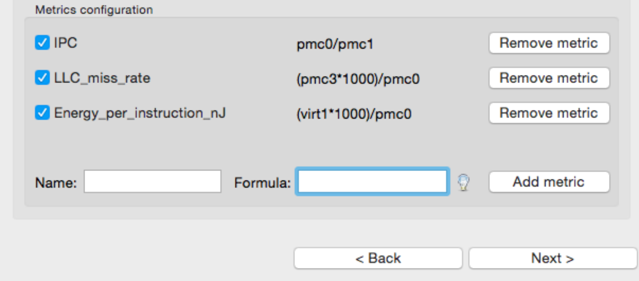
PMCTrack-GUI: a Python graphical frontend

(1) Select the machine (local/remote)



(2) Select HW events and metrics

PMCTrack-GUI: a Python graphical frontend

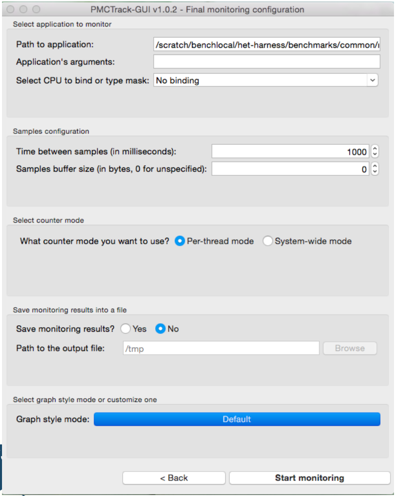


Custom high-level performance metrics can be defined using simple arithmetic expressions

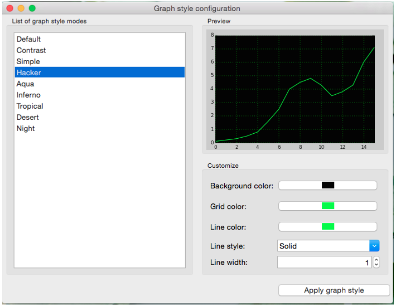
ArTeCS

PMCTrack-GUI: a Python graphical frontend

(3) Specify application/global options

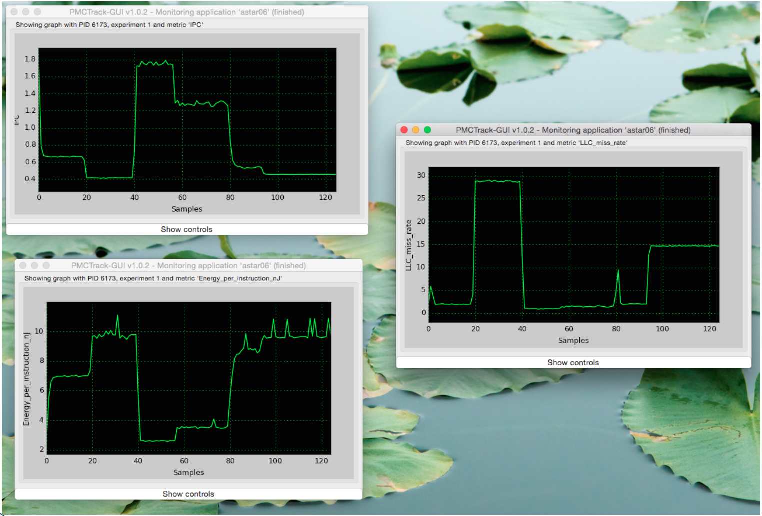


(4) Customize graphs (optional)



PMCTrack-GUI: a Python graphical frontend

(5) Visualize metric graphs (updated in real time)



Outline

1. Introduction
2. PMCTrack architecture and usage modes
3. **Case studies**
   - OS Scheduling for AMPs
   - Cache usage monitoring: Intel CMT
   - Monitoring power/energy consumption
4. Conclusions

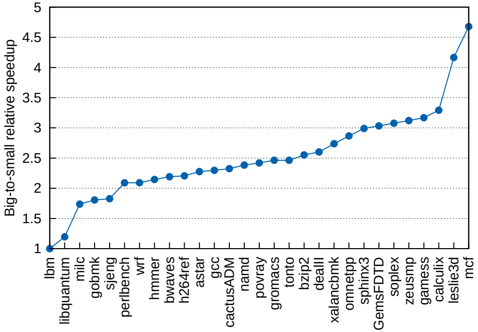
OS Scheduling on AMPs

Asymmetric Multicore Processors (AMPs)

- High-performance big cores + low-power small cores
- Same Instruction Set Architecture (ISA) but different features
- Actual AMPs:
  - ARM big.LITTLE
  - Intel Quick-IA prototype system (Xeon E5450 + Atom N330)

ArTeC

OS Scheduling on AMPs: challenges



SPEC CPU 2006 on the Intel QuickIA

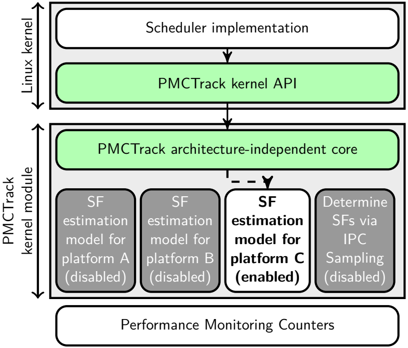
- Applications may derive different benefit (speedup factor - SF) from the big cores relative to small ones
- The speedup may vary over time
- Linux default scheduler (CFS) does not factor in this issue when making scheduling decisions

- An effective asymmetry-aware scheduler should be equipped with a mechanism to determine thread's big-to-small speedups (SFs) online

ArTeCS

Determining the speedup factor (SF)

- Mechanisms to obtain the SF on a real system using PMCs[1]:
  1. Direct measurement on both core types (aka *IPC sampling*)
  2. Estimation via platform-specific model on the *current* core type

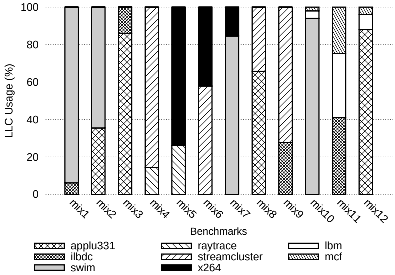


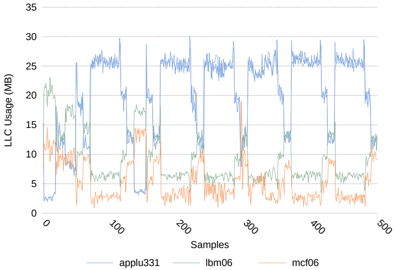
- Every method/model to determine SFs can be implemented as a separate monitoring module
- In previous work[1], we leveraged this approach to evaluate real-world implementations in the Linux kernel of state-of-the-art asymmetry-aware schedulers

[1] Saez et al, *ACFS: a completely fair scheduler for asymmetric single-ISA multicore systems*, In Proc. of ACM SAC'15

Analyzing cache contention: Intel CMT

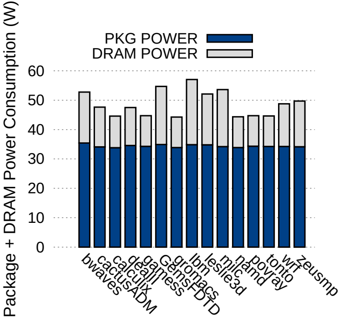
- Recent Intel Xeon processors support monitoring LLC usage on a per-application basis
- A PMCTrack monitoring module provides the associated support
  - Tested on “Haswell-EP” and “Broadwell” Xeon processors



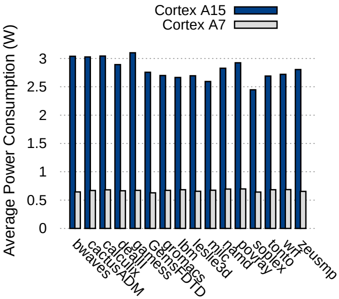


Measuring power/energy consumption

- Power/Energy consumption monitoring support in PMCTrack
  - Intel processors with RAPL capabilities
  - ARM Development boards featuring the big.LITTLE processor



Intel Xeon E5 v3 @ 2.3Ghz



ARM Cortex A15 vs. ARM Cortex A7

Outline

Conclusions

- Public-domain PMC tools do not feature an in-kernel API enabling the OS scheduler to access PMCs in a convenient way
- The PMCTrack tool fills this gap, and also ..
  1. Enables to decouple the low-level PMC code from the scheduler kernel code (platform-independent implementation)
     - The kernel developer does not access PMCs directly
  2. Enables researchers to easily add support to monitor other HW-aided information not exposed as PMCs
     - Monitoring modules $\rightarrow$ faster adoption of HW monitoring facilities
  3. Monitoring information can be accessed from within the OS scheduler, the runtime system code (*libpmctrack*) or via user space tools

PMCTrack open-source project

- PMCTrack's source code has been released under the GPL v2
  - https://github.com/jcsaezal/pmctrack
- More information will be available soon at the official website
  - http://pmctrack.dacya.ucm.es

Questions