An OS-Oriented Performance Monitoring Tool for Multicore Systems

Juan Carlos Saez(✉), Jorge Casas, Abel Serrano, Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias

Facultad de Informática, Complutense University of Madrid, Madrid, Spain
{jcsaezal,jorcasas,abeserra,rrodriguezr,fcastror,dani02,mpmatias}@ucm.es

Abstract. Hardware performance monitoring counters (PMCs) have proven effective in characterizing application performance. Because PMCs can be only accessed directly at the OS privilege level, kernel-level tools must be developed to enable the end user and userspace programs to access PMCs. A large body of work has demonstrated that the OS scheduler can perform effective runtime optimizations in multicore systems by leveraging per-thread performance-counter data. Notably, while existing tools greatly simplify collecting PMC application data from user space, they do not provide a simple mechanism making it possible for the thread scheduler to use performance counters for its own purpose.

To address this shortcoming we present *PMCTrack*, a novel tool for the Linux kernel that provides a simple architecture-independent mechanism making it possible for the OS scheduler to access per-thread PMC data. Despite being an OS-oriented tool, PMCTrack still allows gathering PMC values from user space, enabling kernel developers to carry out the necessary offline analysis and debugging to assist them during the scheduler design process. In addition, the tool provides both the scheduler and the userspace PMCTrack components with other insightful metrics available in modern processors that are not directly exposed as PMCs, such as cache occupancy or energy consumption. In this paper, we analyze different case studies that demonstrate the potential benefits of PMCTrack.

Keywords: Performance monitoring counters · PMCTrack · OS scheduler · Linux kernel · Asymmetric multicore · Cache monitoring · Intel CMT

1 Introduction

Most modern complex computing systems are equipped with hardware Performance Monitoring Counters (PMCs) that enable users to collect application's performance metrics, such as the number of instructions per cycle (IPC) or the Last-Level Cache (LLC) miss rate. These PMC-related metrics aid in identifying possible performance bottlenecks, thus providing valuable hints to programmers

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 697–709, 2015.
DOI: 10.1007/978-3-319-27308-2_56

and computer architects. Notably, direct access to PMCs is typically restricted to code running at the OS privilege level. As such, a kernel-level tool, implemented in the OS itself or as a driver, is usually in charge of providing userspace tools with a high-level interface enabling to access performance counters [4,7,16].

Previous work has demonstrated that the OS can also benefit from PMC data making it possible to perform sophisticated and effective runtime optimizations on multicore systems [8,9,13,19,20,25]. While public-domain tools to access PMCs make it possible to monitor application performance from user space, they do not provide an architecture-independent API empowering the OS itself to leverage PMC information in different subsystems, such as the thread scheduler. As a result, many researchers turned to architecture-specific *ad-hoc* code to access performance counters when implementing different scheduling schemes [8,9,13]. This approach, however, leads the scheduler implementation to be tied to certain processor models, and at the same time, forces developers to deal with (or even write themselves) the associated low-level routines to access PMCs on each architecture targeted by the scheduler.

To overcome these limitations, we propose *PMCTrack*, an OS-oriented PMC tool for the Linux kernel. PMCTrack's novelty lies in the *monitoring module* abstraction, an architecture-specific extension responsible for providing any OS scheduling algorithm that leverages PMC data with the performance metrics it requires to function. This abstraction makes it possible to implement architecture-independent OS scheduling algorithms. Notably, in doing so, the developer does not have to deal with the platform-specific low-level code to access PMCs on a given architecture, which greatly simplifies the implementation.

PMCTrack is also equipped with a set of command-line tools and user space components. These userland tools assist OS-scheduler designers during the entire development life cycle by complementing the existing kernel-level debugging tools with PMC-related offline analysis and tracing support. Moreover, due to the flexibility of PMCTrack's monitoring modules, any kind of metric provided by modern hardware but not exposed directly via performance counters, such as power consumption or an application's cache footprint, can also be exposed to the OS scheduler or to user applications as PMCTrack's *virtual counters*.

To demonstrate the effectiveness and flexibility of our proposal, we analyze three case studies on real multicore hardware. The first case study showcases the ability of PMCTrack's monitoring modules to aid in the implementation of state-of-the art thread schedulers for asymmetric single-ISA multicore systems [9,11,19,20,24] in the Linux kernel. The second one focuses on cache-usage monitoring via PMCTrack's support for Intel Cache Monitoring Technology [14]. Leveraging this technology, we propose a technique to build applications' Miss-Rate Curves (MRCs) on a real system. The third case study illustrates PMCTrack's ability to carry out energy and power consumption measurements on different processor models. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 outlines the design of PMCTrack. Section 4 focuses on the case studies to evaluate our design and finally Sect. 5 concludes.

2 Related Work

Hardware performance monitoring counters are usually exposed to the software as a set of privileged registers. For example, in x86 processors, PMCs can be accessed from the system software via Model-Specific Registers (MSRs) [6]. Other processor architectures, such as ARM, give more freedom to the processor implementer on how these counters are ultimately exposed to the OS [1].

Several tools [4,7,15,16,23] have been created for the Linux kernel over the last few years, enabling to hide the diversity of the various hardware interfaces and providing users with convenient access to PMCs from user space. Overall, these tools can be divided into two broad categories. The first group encompasses tools such as OProfile [4], perfmon2 [7] or perf [16], which expose PMCs to the user via a reduced set of command-line tools. These tools do not require to modify the source code of the application being monitored; instead they act as external processes with the ability to receive PMC data of another application. The second group of tools provides the user with libraries to access counters from an application's source code, thus constituting a fine-grained interface to PMCs. The libpfm [7] and PAPI [15] libraries follow this approach.

The perf [16] tool, which relies on the Linux kernel's *Perf Events* subsystem, is possibly the most comprehensive tool in the first category currently available. Not only does perf support a wide range of processor architectures, but also empowers users with striking software tracing capabilities enabling them to keep track of a process' system calls or scheduler-related activity, or various network/file-related operations executed on behalf of an application. Despite the potential of perf and the other aforementioned tools, neither of them implement a kernel-level architecture-agnostic mechanism enabling the OS scheduler to leverage PMC data for its internal decisions. PMCTrack has been specifically designed to fill this gap. As PMCTrack, perf has also the capability to expose non-PMC hardware-related data exposed by modern hardware to the user, such as the LLC occupancy. However, we found that the complexity of Linux Perf Events subsystem, on which perf is based, makes it difficult to add the necessary support. We elaborate on this issue in Sect. 4.2. Instead, PMCTrack's monitoring modules constitute a more straightforward vehicle to expose this kind of metrics to users and to the OS scheduler.

3 Design

Figure 1 depicts PMCTrack internal architecture. The tool consists of a set of user and kernel space components. At a high level, end users and applications interact with PMCTrack using the available command-line tools or via *libpmctrack*. These components communicate with PMCTrack's kernel module by means of a set of Linux /proc entries exported by the module.

The kernel module implements the vast majority of PMCTrack's functionality. To gather per-thread performance counter data, the module needs to be fully-aware of thread scheduling events (e.g., context switches, thread creation).

In addition to exposing application's performance counter data to the userland tools, the module implements a simple API to feed with per-thread monitoring data to any scheduling policy (class) that requires performance-counter information to function. Because both the core Linux Scheduler and scheduling classes are implemented entirely in the kernel, making PMCTrack's kernel module aware of these events and requests requires some minor modifications to the Linux kernel itself. (Augmenting a recent version of the Linux kernel – 2.6.38 and above – to support PMCTrack entails including two new source files to the kernel tree, and adding less than 20 extra lines of code.) These modifications, referred to as PMCTrack kernel API in Fig. 1, comprise a set of notifications issued from the core scheduler to the module. To receive key notifications PMCTrack's kernel module implements the `pmc_ops_t` interface shown in Listing 1. Most of these notifications get engaged only when PMCTrack's kernel module is loaded and the user (or the scheduler itself) is using the tool to monitor the performance of specific applications.

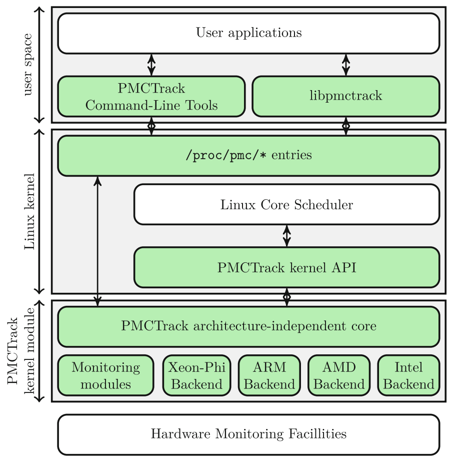


Fig. 1. PMCTrack architecture

```
typedef struct pmc_ops{
/* invoked when a new thread is created */
  void* (*pmcs_alloc_per_thread_data)
      (unsigned long,struct task_struct*);
/* invoked when thread leaves the CPU */
  void (*pmcs_save_callback)(void*, int);
/* invoked when thread enters the CPU */
  void (*pmcs_restore_callback)(void*, int);
/* invoked every tick on a per-thread basis */
  void (*pmcs_tbs_tick)(void*, int);
 /* invoked when a process invokes exec() */
  void (*pmcs_exec_thread)
      (struct task_struct*);
/* invoked when thread exists the system */
  void (*pmcs_exit_thread)
    (struct task_struct*);
/* invoked when thread descriptor is freed up */
  void(*pmcs_free_per_thread_data)
  (struct task_struct*);
/* invoked when the scheduler requests
  per-thread monitoring information */
  int  (*pmcs_get_current_metric_value)
    (struct task_struct*, int, uint64_t*);
} pmc_ops_t;
```

Listing 1. `pmc_ops_t` interface

PMCTrack's kernel module consists of various components. The architecture-independent core layer implements `pmc_ops_t` interface and interacts with PMCTrack userland components. This layer relies on a Performance Monitoring Unit (PMU) backend to carry out low-level access to performance counters, as well as for translating user-provided counter configuration strings into internal data structures for the platform in question. At the time of this writing, PMCTrack includes four backends providing the necessary support for most modern Intel and AMD processors, for some ARM Cortex processor models and for the Intel Xeon Phi Coprocessor. PMCTrack's kernel module also includes a set of platform-specific *monitoring modules*. The primary purpose of a monitoring module is to provide the end user or a scheduling algorithm implemented in the kernel with high-level performance metrics or other insightful runtime information potentially exposed by the hardware (via PMCs or by other means), such as power consumption or a process's last-level cache occupancy.

3.1 Usage Models

PMCTrack can be used to gather performance counter data from the OS scheduler (using an in-kernel interface) and from user space.

(A) Accessing PMC data from the OS scheduler. This feature enables any scheduling algorithm in the kernel (i.e., scheduling class) to collect per-thread monitoring data, thus making it possible to drive scheduling decisions based on tasks' memory behavior or other microarchitectural properties. Turning on this feature for any thread from the scheduler's code boils down to activating a flag in the thread's descriptor. A scheduling algorithm relying on PMCTrack typically enables in-kernel monitoring for all threads belonging to its scheduling class.

To ensure that the implementation of the scheduling algorithm that leverages this feature remains architecture independent, the scheduler does not configure nor deals with performance counters directly. Instead, one of PMCTrack's *monitoring modules* is in charge of feeding the scheduling policy with the necessary high-level performance monitoring metrics, such as a task's instruction per cycle ratio or its last-level cache miss rate.

PMCTrack may include several monitoring modules compatible with a given platform. However, only one can be enabled at a time: the one that provides the scheduler with the PMC-related information it requires to function. In the event several compatible monitoring modules are available, the system administrator may tell the system which one to use by writing in a special /proc file. The scheduler can communicate with the *active* monitoring module to obtain per-thread data via the following function from PMCTrack's kernel API:

```
int pmcs_get_current_metric_value(struct task_struct* task,
        int metric_id, uint64_t* value);
```

For simplicity, each metric is assigned a numerical ID, known by the scheduler and the monitoring module. To obtain up-to-date metrics, the aforementioned function may be invoked from the tick processing function in the scheduler.

Monitoring modules make it possible for a scheduling policy relying on performance counters to be seamlessly extended to new architectures or processor models as long as the hardware enables to collect necessary performance data. All that needs to be done is to build a monitoring module or adapt an existing one to the platform in question. From the programmer's standpoint, creating a monitoring module entails implementing an interface very similar to `pmc_ops_t`. Specifically, it consists of several callback functions enabling to notify the module on activations/deactivations requested by the system administrator, on threads' context switches, every time a thread enters/exits the system, whenever the scheduler requests the value of a per-thread PMC-related metric, etc. The programmer typically implements only the subset of callbacks required to carry out the necessary internal processing. Notably, in doing so, the developer does not have to deal with performance-counter registers directly. Specifically, the programmer indicates the desired counter configuration (encoded in a string) to the PMCTrack architecture-independent core. Whenever new PMC samples are collected for a thread, a callback function of the monitoring module gets invoked, passing the samples as a parameter. Due to this feature, a monitoring module will only access low-level registers to provide the scheduler or the end user with other hardware monitoring information not modeled as PMCs, such as energy consumption.[1] This information is exposed to the user via *virtual counters.*

(B) Using PMCTrack from userspace. In addition to the in-kernel API presented above, PMCTrack also enables to gather PMC data from user space by using the `pmctrack` command line tool or `libpmctrack`.

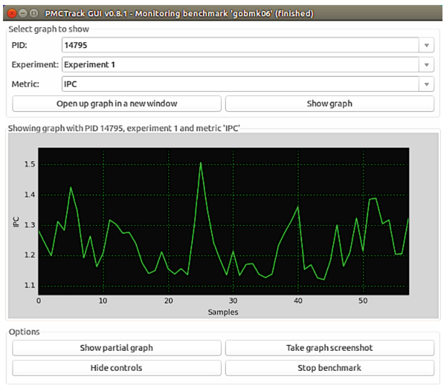
The `pmctrack` command allows the user to gather an applications's performance data at regular time intervals (a.k.a., time-based sampling - TBS) or by using the interrupt-on-counter-overflow feature available on most modern performance monitoring units (a.k.a., event-based sampling - EBS). This command enables to specify counter and event configurations using mnemonics in much the same way as existing user-oriented tools [4,7,16]. In addition, the program supports monitoring both multithreaded and single-threaded applications and has event-multiplexing capabilities (several event sets can be monitored in a round-robin fashion). To complement this command-line tool with real-time visualization of high-level performance metrics (such as the IPC or the LLC miss rate) we also created PMCTrack-GUI, a Python front-end for `pmctrack`. Figure 2 shows a screenshot of PMCTrack-GUI. This application extends the capabilities of the PMCTrack stack with other relevant features, such as an SSH-based remote monitoring mode or the ability to plot user-defined performance metrics.

`Libpmctrack` enables to characterize performance of code fragments via PMCs in sequential and multithreaded programs. Libpmctrack's API makes it possible to indicate the desired PMC configuration to the PMCTrack's kernel module at any point in the application's code or within a runtime system. The programmer may then retrieve the associated event counts for any code snippet (via TBS or EBS) simply by enclosing the code between invocations to the `pmctrack_start_count()` and `pmctrack_stop_count()` functions.

Not only do libpmctrack and the pmctrack command enable the user to gather PMC application data but also have the ability to provide users with any other relevant information exported by the active monitoring module as a *virtual counter.* In Sects. 4.2 and 4.3 we leverage this capability. To feed libpmctrack and the pmctrack program with PMC and virtual-counter data, the PMCTrack's kernel module stores this information in kernel-space ring buffers.

Despite the fact that some features of the `pmctrack` command and libpmctrack are also available in existing tools [15,16], the design of PMCTrack coupled with the virtual counter abstraction makes it simpler to expose any new insightful information provided by the hardware to userland tools. Specifically, in tools

[1] Most modern Intel processors and many ARM development boards provide energy consumption readings via separate registers or sensors (see Sect. 4.3).


PMCTrack GUI v0.8.1 - Monitoring benchmark 'gobmk06' (finished)

Select graph to show

PID: 14795

Experiment: Experiment 1

Metric: IPC

Showing graph with PID 14795, experiment 1 and metric 'IPC'

Fig. 2. PMCTrack-GUI

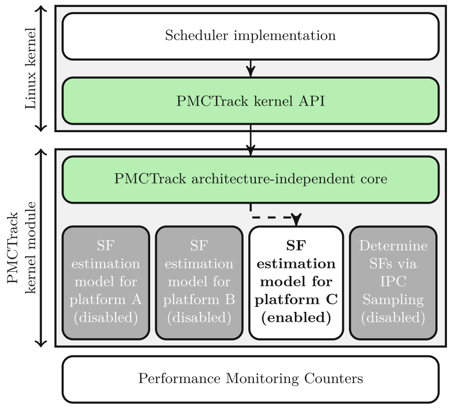

Linux kernel
Scheduler implementation
PMCTrack kernel API
PMCTrack kernel module
PMCTrack architecture-independent core
SF estimation model for platform A (disabled)
SF estimation model for platform B (disabled)
SF estimation model for platform C (enabled)
Determine SFs via IPC Sampling (disabled)
Performance Monitoring Counters

Fig. 3. PMCTrack monitoring modules

such as perf [16] or PAPI-C [15], the Linux kernel and the associated userspace components must be typically modified to benefit from new hardware monitoring facilities. In PMCTrack, by contrast, this extra support can be provided by creating a new monitoring module (modifying PMCTrack's kernel loadable module), which can be done even without rebooting the system.

4 Case Studies

4.1 Scheduling on Asymmetric Single-ISA Multicore Systems

Previous research has highlighted that asymmetric single-ISA multicore (AMP) processors, which couple same-ISA complex high-performance *big* cores with power-efficient *small* cores on the same chip, have been shown to significantly improve energy and power efficiency over their symmetric counterparts [10]. The ARM big.LITTLE processor [2] and the Intel Quick-IA prototype [3] demonstrate that AMP designs have drawn the attention of major hardware players.

Despite their benefits, AMPs pose significant challenges to the OS scheduler. One of the main challenges is how to effectively distribute big-core cycles among the various applications running on the system. Previous work has proposed thread scheduling algorithms to optimize throughput and fairness on AMPs [9,11,19,20,24]. Notably, the key to optimizing both metrics is to factor in the big-to-small speedup (aka, *speedup factor*) of the various threads when making thread-to-core mappings. A thread's speedup factor is defined as $\frac{IPS_{big}}{IPS_{small}}$, where IPS_{big} and IPS_{small} are the thread's instructions per second ratios achieved on big and small cores respectively.

Three different schemes have been explored to determine per-thread speedup factors (SFs) online. The first approach boils down to measure SFs directly [10,24], which entails running each thread on big and small cores to track the IPC (instructions per cycle) on both core types. Previous work has demonstrated that this approach, known as *IPC sampling*, is subject to inaccuracies in SF estimation associated with program-phase changes [21]. The second approach relies on *estimating a thread's SF* using its runtime properties collected on any core type at runtime using PMCs [9,19]. Unfortunately, estimating SFs via PMCs requires to derive predictions models specifically tailored to the platform in question [9,19]. The third technique is PIE [24], a hardware-aided mechanism enabling accurate SF estimation from any core type. Notably, the required hardware support for PIE has not yet been adopted in commercial systems.

To leverage the first two approaches, which can be used in off-the-shelf AMPs, the OS scheduler needs to access PMCs in the platform in question. By using the approach depicted in Fig. 3, PMCTrack monitoring modules make it possible to aid in creating an architecture-independent implementation of the scheduler. Overall, the various schemes to obtain threads' SFs can be implemented as separate PMCTrack monitoring modules: one that leverages IPC sampling and several others that provide SF-estimation on the various platforms supported by the scheduler. Apart from the architecture-agnostic scheduler implementation, this design approach provides three additional benefits. First, the scheduler implementation is completely decoupled from the underlying scheme to determine the SF; the kernel developer or the system administrator can decide which scheme to use by activating the corresponding monitoring module. Second, existing SF-enabled modules can be modified and new ones can be created for a particular scheduler even without rebooting the system. Third, since the SF can be seamless exposed as a virtual counter to PMCTrack's command-line tools, per-thread SF and PMC traces can be gathered from user space for debugging purposes. Notably, in previous work [20] we leveraged this potential of PMCTrack monitoring modules to implement state-of-the-art asymmetry-aware scheduling algorithms [9,11,19,24] in the Linux kernel.

4.2 Cache Monitoring

Intel's Cache Monitoring Technology (CMT) [6] is a new feature, introduced in the Intel Xeon E5 2600 v3 product family, that allows an operating system or a Hypervisor/Virtual Machine Monitor (VMM) to determine the current last-level cache (LLC) usage of the various applications running on the platform. At a high level, CMT works as follows. The OS or VMM assigns a certain ID, referred to as the Resource Monitoring ID (RMID)[2], to each application/VM. Cache occupancy is monitored by CMT-enabled hardware on a per-RMID basis, so the OS or the VMM can read LLC occupancy for a given application/VM at any time. To make this possible, the processor needs to be aware of the RMID of every thread (or virtual CPU) currently running on the system. To this end, the hardware exposes a per-core privileged register that stores the RMID associated with the thread currently running on it. The OS is in charge of updating per-core RMID registers when threads' context-switches take place.

[2] Because the amount of RMIDs available is limited by the hardware implementation, the OS must be equipped with a carefully crafted RMID allocation policy.

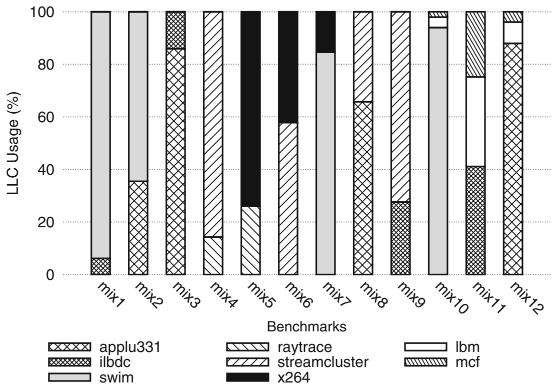


Fig. 4. Breakdown of LLC occupancy for different mixes.

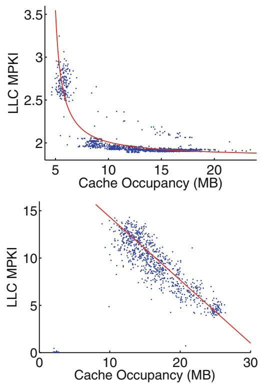


Fig. 5. MRCs for the *lbm* (top) and *omnetpp* (bottom) applications.

A patch [5] has been recently created to augment perf [16] with Intel-CMT capabilities. This support is expected to be included in future Linux kernel releases. We found that PMCTrack's implementation is simpler and exhibits some important advantages over perf's implementation. First, our implementation is done in a loadable kernel module rather than on the kernel itself, which greatly simplified the development. Second, it uses less than 500 lines of code against the 1500 lines used by the perf patch for the Linux kernel [5]. Moreover, perf's implementation entails making changes to 7 different source files from the Linux kernel, whereas our implementation requires adding a new file to PMCTrack's sources, and changing another source file to register the new monitoring module. Third, because Intel-CMT support is encapsulated in a monitoring module, any shared-resource contention-aware scheduling policy implemented in the Linux kernel could easily retrieve an application's LLC usage (via PMCTrack's API) and perform effective thread-to-core mappings [14]. At the same time, the monitoring module exposes the LLC usage to the userspace tools as a virtual counter.

(A) LLC occupancy analysis. We first performed an analysis on the LLC occupancy of 12 multiprogram workloads consisting of sequential programs (`lbm` and `mcf` from the SPEC CPU 2006 suite) and parallel applications (from SPEC OMP 2012 and PARSEC). All multithreaded applications run with 4 threads. To carry out the experiment, we employed a 14-core "Haswell-EP" Xeon E5-2695 v3 processor operating at 2.3 GHz, which features a 35 MB last-level (L3) cache.

Figure 4 shows the average per-application LLC occupancy for the entire execution of the various workloads. Our experiment reveals that some applications, like the *swim* SPEC OMP parallel program, use a great portion of the LLC (65–95 %) when running concurrently with sequential applications or with other multithreaded SPEC OMP or PARSEC programs. Conversely, other parallel programs, like *ilbdc* from the SPEC OMP suite, typically occupy a much more reduced portion of the L3 cache regardless of the co-runner application. As expected, other applications account for significantly different portion of LLC depending on the co-runner.

(B) MRCs online generation. As a more sophisticated application of PMCTrack and CMT, we now introduce a technique to generate Miss Rate Curves (MRCs) online. The MRC reports an application's cache occupancy on a given cache level (usually the LLC) vs. a certain related performance metric, like the number of Misses Per Kilo Instructions (MPKI). MRCs can be employed for different purposes, such as to efficiently distribute a shared cache among threads [18,22]. Several mechanisms have been proposed [18,22] for building these curves, but they all pose different limitations, such as requiring hardware support or relying on code instrumentation.

We now describe an online technique that leverages Intel's CMT to generate the MRCs of co-running applications. Overall, the proposed technique works as follows. Using PMCTrack we gather the MPKI and the LLC occupancy of the co-running applications periodically, thus obtaining different discrete MRC points. Then, when enough points have been collected, we apply regression analysis to obtain the whole MRCs for the applications. Note, however, that when several applications share a cache, they usually reach an equilibrium state in the distribution of the cache. To obtain points in the whole range of cache sizes, we slow down co-runner applications by applying duty-cycle modulation techniques to the cores where they run. This allows other applications to increase their occupancy, which in turn, makes it possible for us to explore different MPKI values for the whole cache size range. Figure 5 illustrates two examples of curves obtained with this technique. The MRC for the *lbm* application shows a steep MPKI fall for small cache occupancy values and then saturates from a certain cache size point on. Conversely, the MRC for the *omnetpp* program, shows a linear MPKI drop for the whole range of cache sizes.

4.3 Measuring Power and Energy Consumption

PMCTrack also has the ability to interact with power and energy measurement facilities of modern high-performance Intel processors [6] and low-power ARM big.LITTLE systems [2]. The necessary support is provided by a set of PMCTrack monitoring modules. As such, energy-related information is readily available at runtime to both the OS scheduler and to the end users via virtual counters.

On Intel systems we turned to the Running Average Power Limit (RAPL) feature [6], which employs a software power model based on hardware monitoring to approximate energy usage. Specifically, energy consumption can be gathered independently for different power domains (core-level, processor package/uncore and DRAM). Figure 6(a) and (b) show power consumption measurements reported by PMCTrack on the Intel Xeon E5 v3 processor described in the previous section. Specifically, the data illustrate both package-level and DRAM power

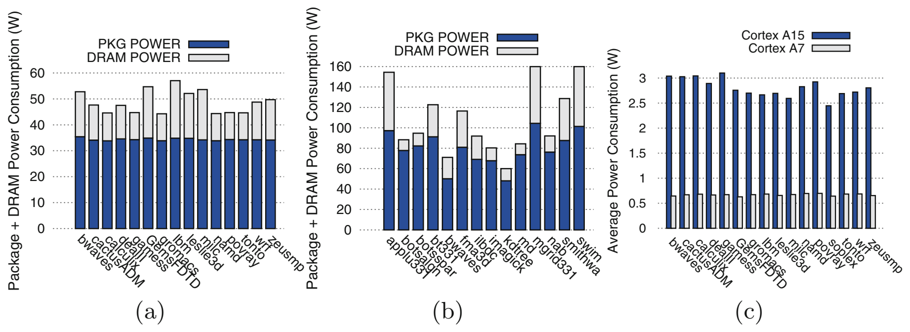


Fig. 6. Average power consumption gathered via Intel RAPL for (a) FP SPEC CPU2006 (b) and SPEC OMP 2012 benchmarks. (c) Average per-cluster power consumption for FP SPEC CPU 2006 benchmarks on an ARM big.LITTLE processor.

measurements for several floating-point sequential programs (SPEC CPU2006) and parallel applications (SPEC OMP 2012). As evident, sequential programs exhibit almost constant package consumption across the board, since the application is using only one core in the processor package.

PMCTrack also provides support for 32-bit ARM big.LITTLE processors featuring one cluster of Cortex A15 *big* cores and another cluster of Cortex A7 *small* cores. In our evaluation, we used the ARM CoreTile Express development platform, which is equipped with a set of sensors enabling to measure power and energy consumption on a per-cluster basis. Figure 6(c) shows the observed average cluster power consumption of floating-point SPEC CPU 2006 benchmarks when running alone on the various core types. Information retrieved via the aforementioned sensors reveals that big A15 cores yield up to 4.9x the power consumption of small A7 cores. Still, both ARM cores exhibit significantly lower package-level power consumption for the same benchmarks compared to that of the high-performance Intel processor we used (Fig. 6(a)).

To validate the results shown in Fig. 6, we measured power consumption using perf [16] and lm-sensors [12] on the Intel and the ARM systems respectively. (Unlike PMCTrack, neither of these tools enables to monitor energy consumption on both platforms.) The results, omitted due to space constraints, reveal similar measurements to those reported by PMCTrack (deviations no greater than 1 %).

5 Conclusions and Future Work

In this paper we have proposed PMCTrack, a tool enabling the OS scheduler to leverage performance monitoring counter (PMC) information in decision making. By using PMCTrack's monitoring module abstraction, the implementation of any scheduling policy that relies on per-thread PMC data to function remains fully platform independent. Not only do monitoring modules provide the OS scheduler with PMC-related metrics but also have the ability to feed it with virtually any insightful information exposed by modern hardware and not necessarily provided via PMCs, such as the LLC occupancy or the energy/power consumption. Despite being an OS-oriented tool, PMCTrack is also equipped with a set of userland tools that allow to gather hardware-performance monitoring information from user space in various ways.

In an earlier work [20], we leveraged the potential of PMCTrack's monitoring modules in assisting scheduling algorithms for asymmetric single-ISA multicore systems implemented in the Linux kernel. By using PMCTrack's in-kernel API and libpmctrack, we plan to evaluate OS and runtime-level scheduling schemes that leverage information on cache occupancy and power consumption.

PMCTrack's source code has been released[3] under the GPLv2. Additional information on PMCTrack will be available at PMCTrack's official website [17].

Acknowledgements. This work has been supported by the Spanish government through the research contract TIN2012-32180 and the HIPEAC3 (see footnote 3) European Network of Excellence.

References

1. ARM: Arm Architecture Reference Manual. http://infocenter.arm.com/
2. ARM: Benefits of the big.LITTLE Architecture. http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf
3. Chitlur, N., et al.: QuickIA: exploring heterogeneous architectures on real prototypes. In: Proceedings of HPCA 2012, pp. 1–8 (2012)
4. Cohen, W.: Tuning programs with oprofile. Wide Open Mag. **1**, 53–62 (2004)
5. Flemming, M.: perf: Intel cache QoS monitoring support (2014). https://lkml.org/lkml/2015/1/23/590
6. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 3A and 3B. http://www.intel.com/products/processor/manuals
7. Jarp, S., Jurga, R., Nowak, A.: Perfmon2: a leap forward in performance monitoring. J. Phys. Conf. Ser. **119**, 042017 (2008)
8. Knauerhase, R., et al.: Using OS observations to improve performance in multicore systems. IEEE Micro **28**(3), 54–66 (2008)
9. Koufaty, D., Reddy, D., Hahn, S.: Bias scheduling in heterogeneous multi-core architectures. In: Proceedings of Eurosys 2010, pp. 125–138 (2010)
10. Kumar, R., et al.: Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of ISCA 2004, pp. 64–75 (2004)
11. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multicore architectures. In: Proceedings of HPCA 2010, pp. 1–12 (2010)
12. Lm-sensors: HW monitoring by lm-sensors (2015). http://www.lm-sensors.org/
13. Merkel, A., et al.: Resource-conscious scheduling for energy efficiency on multicore processors. In: Proceedings of EuroSys, pp. 153–166 (2010)
14. Nguyen, K.: Benefits of intel(r) cache monitoring technology in the intel(r) xeon(tm) processor e5 v3 family (2014). https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring
15. Papi: Overview. http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview
16. Perf: Wiki tutorial on perf (2015). https://perf.wiki.kernel.org/index.php
17. PMCTrack: project official website. http://pmctrack.dacya.ucm.es/

[3] https://github.com/jcsaezal/pmctrack.

18. Qureshi, M., et al.: Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: MICRO (2006)
19. Saez, J.C., et al.: Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. ACM Trans. Comput. Syst. **30**(2), 38 (2012). Article 6
20. Saez, J.C., et al.: ACFS: a completely fair scheduler for asymmetric single-ISA multicore systems. In: Proceedings of ACM SAC 2015 (2015)
21. Shelepov, D., et al.: HASS: a scheduler for heterogeneous multicore systems. ACM OSR **43**(2), 66–75 (2009)
22. Tam, D.K., et al.: RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In: Proceedings of ASPLOS 2009, pp. 121–132 (2009)
23. Taniça, L., Ilic, A., Tomás, P., Sousa, L.: SchedMon: a performance and energy monitoring tool for modern multi-cores. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part II. LNCS, vol. 8806, pp. 230–241. Springer, Heidelberg (2014)
24. Van Craeynest, K., et al.: Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of PACT 2013, pp. 177–187 (2013)
25. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing cache contention in multicore processors via scheduling. In: Proceedings of ASPLOS 2010, pp. 129–142 (2010)