Advance Access publication on 8 September 2016
doi:10.1093/comjnl/bxw065

PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler

J. C. Saez[1], A. Pousa[2], R. Rodríguez-Rodríguez[1], F. Castro[1]* and M. Prieto-Matias[1]

[1]*ArTeCS Group, Facultad de Informática, Complutense University of Madrid, Madrid, Spain*
[2]*III-LIDI, Facultad de Informática, National University of La Plata, La Plata, Argentina*
[*]*Corresponding author: fcastror@ucm.es*

Hardware performance monitoring counters (PMCs) have proven effective in characterizing application performance. Because PMCs can only be 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 can perform effective runtime optimizations in multicore systems by leveraging performance-counter data. Special attention has been paid to optimizations in the OS scheduler. While existing performance monitoring tools greatly simplify the collection of PMC application data from userspace, they do not provide an architecture-agnostic kernel-level mechanism that is capable of exposing high-level PMC metrics to OS components, such as the scheduler. As a result, the implementation of PMC-based OS scheduling schemes is typically tied to specific processor models. To address this shortcoming we present *PMCTrack*, a novel tool for the Linux kernel that provides a simple architecture-independent mechanism that makes it possible for the OS scheduler to access per-thread PMC data. Despite being an OS-oriented tool, PMCTrack still allows the gathering of monitoring data from userspace, 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 OS and the user-space PMCTrack components with other insightful metrics available in modern processors and which are not directly exposed as PMCs, such as cache occupancy or energy consumption. This information is also of great value when it comes to analyzing the potential benefits of novel scheduling policies on real systems. In this paper, we analyze different case studies that demonstrate the flexibility, simplicity and powerful features of PMCTrack.

Keywords: performance monitoring counters; PMCTrack; OS scheduling; Linux kernel; asymmetric multicore; energy efficiency; cache monitoring; Intel CMT

Received 5 December 2015; revised 4 July 2016
Handling editor: Javier Barria

1. INTRODUCTION

Most modern complex computing systems are equipped with hardware Performance Monitoring Counters (PMCs) that enable users to collect an 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 clues to programmers and computer architects. It should be noted that direct access to PMCs is typically restricted to code running at the OS privilege level. Thus, 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 that enables access to performance counters [1–3].

Previous work has demonstrated that the OS can also benefit from PMC data by making it possible to perform sophisticated and effective runtime optimizations on multicore systems [4–12]. Special attention has been paid to optimizations in the OS scheduler. Notably, many of the

proposed PMC-based OS scheduling schemes rely on per-thread high-level metrics that are estimated by means of platform-specific prediction models [9–14]. In this scenario, determining the necessary per-thread high-level metrics (e.g. energy efficiency or performance ratios across cores) at run time entails monitoring a specific set of hardware PMC events that may differ substantially across processor models and architectures [9, 10]. Unfortunately, public-domain PMC monitoring tools, which are largely userspace oriented, do not provide an architecture-independent mechanism that enables feeding PMC-based OS scheduling schemes with the necessary high-level monitoring information they require to function. Due to the limited support for in-kernel monitoring in public-domain PMC tools, some researchers have employed architecture-specific *ad-hoc* code to access performance counters in the scheduler implementation [4, 9, 10]. However, this approach still leads the scheduler to be tied to certain processor models. Other researchers have resorted to evaluating their proposals by means of simplistic userspace scheduling prototypes [6, 8, 11, 14] that rely on existing userspace-oriented PMC tools.

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, a platform-specific component that is responsible for collecting the necessary high-level metrics that a given OS scheduling algorithm requires to function. This abstraction makes it possible to create architecture-independent implementations of OS scheduling algorithms that leverage PMC data. Figure 1 illustrates the interaction between the scheduler and PMCTrack's monitoring modules. Essentially, the scheduler does not access or deals with performance counters or hardware events directly, but instead uses the PMCTrack kernel API to retrieve the necessary per-thread or per-application performance metrics from the underlying monitoring module. In this way, ensuring that a

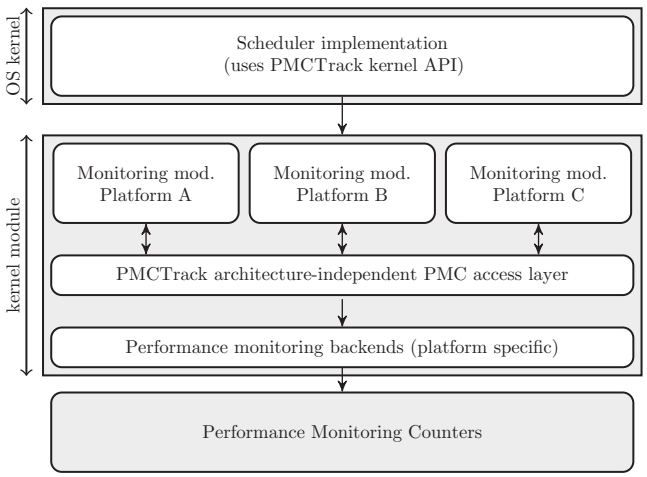

FIGURE 1. Interaction between the OS scheduler and PMCTrack's monitoring modules.

PMC-based thread scheduler works on a new processor model or architecture comes down to developing the associated platform-specific monitoring module in a loadable kernel module. More importantly, the monitoring module developer does not have to deal with the low-level code to access PMCs directly on a given architecture, since PMCTrack offers an architecture-independent interface to easily configure events and gather PMC data, which greatly simplifies the implementation. Furthermore, due to the flexibility of PMCTrack's monitoring modules, any kind of insightful monitoring information provided by modern hardware but not modeled directly via performance counters, such as power consumption or an application's cache footprint, can also be exposed to the OS via the PMCTrack kernel API and to user applications via PMCTrack's *virtual counters*.

Despite being an OS-oriented tool, PMCTrack is also equipped with a set of command-line tools and userspace components to assist OS-scheduler designers during the entire development process. These userspace tools complement existing kernel-level debugging tools with PMC-related off-line analysis and tracing support. As shown in this paper, these tools can be of great value to researchers when it comes to assessing the potential benefits of novel OS scheduling policies. Although the main focus of this paper is on illustrating how PMCTrack can aid the OS scheduler, the tool could potentially be used to perform PMC-based optimizations in other OS components as well (e.g. memory management).

To demonstrate the effectiveness and flexibility of PMCTrack we analyze three case studies on real multicore hardware. In doing so, we make the following contributions:

- We perform an experimental analysis of the throughput and fairness of state-of-the-art thread schedulers for asymmetric single-ISA multicore systems [9,12, 15–17] implemented in a real operating system. Most of these algorithms require the collection of different sets of hardware events across platforms to determine the high-level metrics necessary to drive scheduling decisions. PMCTrack enabled us to create platform-independent implementations of these schemes. Notably, some of the schemes studied were evaluated before using emulated asymmetric hardware [16] or simulators [17]. Instead, we performed an extensive evaluation on real asymmetric hardware, which enabled us to detect important benefits and drawbacks of the various schemes.
- We also showcase the ability of PMCTrack to sample performance counters and energy-consumption registers/sensors in a fully coordinated fashion on different architectures. Notably, this functionality is missing in standard Linux monitoring tools [18, 19] on some platforms, as discussed in Section 2. PMCTrack fills this gap, thus making it possible to measure insightful high-level metrics that factor in

information on performance counters and energy consumption. By using this feature, we explore the potential reduction in the energy-delay product (EDP) that can be achieved by using an energy-aware scheduling scheme on a system featuring an ARM big.LITTLE processor.
- Finally, we propose a technique for building the Miss-Rate Curve (MRC) of an application on a real system. This technique relies on PMCTrack's support for cache-usage monitoring on systems equipped with Intel's Cache Monitoring Technology (CMT) [20].

The rest of the paper is organized as follows. Section 2 discusses the background and related work. Section 3 outlines the design of PMCTrack. Section 4 presents the case studies to evaluate our design and, finally, Section 5 concludes.

2. BACKGROUND AND RELATED WORK

Hardware PMCs 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) [21]. Other processor architectures, such as ARM, give more freedom to the processor implementer on how these counters are exposed to the OS [22].

Several userspace-oriented tools have been created for the Linux kernel in the last few years [2, 3, 18, 23–25]. These tools hide the diversity of the various hardware interfaces to end users and provide them with convenient access to PMCs. Overall, they can be divided into two broad categories. The first group encompasses tools such as OProfile [3], perfmon2 [2] or perf [18], which expose performance counters to the user via a reduced set of command-line tools. These tools do not require modification of the source code of the application being monitored; instead, they act as external processes with the ability to receive the 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 with PMCs. The libpfm [2] and PAPI [23] libraries follow this approach.

The perf [18] tool, which relies on the Linux kernel's *Perf Events* [1] subsystem, is possibly the most comprehensive tool available in the first category at the time of writing. Not only does perf support a wide range of processor architectures, but it also empowers users with striking software tracing capabilities, enabling them to keep track of a process's system calls or scheduler-related activity, or various network/file-related operations executed on behalf of an application.

Despite the potential of perf events and the other aforementioned tools, none of them implement a kernel-level mechanism that is specifically tailored to create architecture-independent implementations of scheduling schemes that leverage PMC data for its internal decisions. In particular, the perf events subsystem includes a kernel API which has been designed primarily to build userspace-oriented monitoring tools on top of it. Currently, the API is being used to provide the support necessary for the perf command-line tool as well as for OProfile [3]. Because none of the scheduling algorithms implemented in the mainstream Linux kernel requires performance counters to function, this API is not being used from the scheduler. Although a reduced set of functions in perf event's API enable access to performance counters within the kernel, the mechanism required to indicate the set of hardware events to monitor is largely platform specific (i.e. hardware events are assigned different IDs in different processor models). Therefore, if that API were used from the scheduler implementation directly, different code paths would be necessary to deal with hardware PMC events on every single architecture/processor family supported by the hypothetical scheduler implementation. The complexity of the implementation with such an API would increase further for PMC-based scheduling schemes that rely on platform-specific estimation models [9–14], since a different set of hardware events (possibly unrelated across platforms) must be monitored in different processor models. PMCTrack makes it possible to solve this problem, by feeding the scheduler with the PMC-related high-level metrics it requires to function.

Notably, our proposed tool does not rely on the perf events subsystem to access hardware counters. This makes it possible for PMCTrack's monitoring modules to have a finer-grained control with regards to in-kernel event multiplexing. Specifically, when the number of hardware events to be monitored exceeds the number of performance counters available in a processor core, time multiplexing must be used; in other words, different subsets of events must be monitored in a round-robin (RR) fashion (different sampling intervals). In this scenario, perf event's kernel API manages multiplexing implicitly, and so, the kernel programmer is not notified at the end of every sampling interval associated with different event sets. Instead, this API provides a (scaled) estimate of each event count to approximate the value of the event as if no time multiplexing were used. This behavior is suitable for user-space monitoring; however, it prevents kernel developers from implementing mechanisms that rely on managing event multiplexing explicitly, such as those required by the phase-aware estimation models presented in Sections 4.1.2 and 4.2.2. PMCTrack makes it possible to overcome this limitation, and also enables the monitoring of separate sets of hardware events in different core types of the system.

PMCTrack is also equipped with key features missing in the perf events subsystem. In particular, perf events lacks support for accessing special registers and sensors for measuring energy and power consumption on many embedded boards featuring ARM processors that are extensively used for research today, such as the ARM CoreTile Express TC2 board [26] and the ARM Juno Development Board [27]. These registers and sensors are not exposed as regular performance counters to the system software. Notably, the support necessary to access them on Linux is being implemented[1] as part of the *hwmon* kernel subsystem, on which the lm-sensors userspace tool [19] relies. Because perf events and hwmon are two separate kernel subsystems, and employ very different mechanisms to expose hardware monitoring facilities to the user, gathering insightful high-level metrics over time that combine information on performance counters and energy consumption (e.g. the energy consumption per instruction over time) becomes very challenging and unreliable in these systems, as separate tools must be used to obtain them. By contrast, PMCTrack makes it possible to obtain these high-level metrics in a seamless way from both user space and kernel space, as performance counters and energy-consumption registers (exposed to the user as virtual counters) are sampled in a fully coordinated fashion. The analysis we carry out in Section 4.2.2 showcases the potential of this feature.

We should also highlight that, unlike PMCTrack, the perf events subsystem is implemented entirely inside the Linux kernel rather than as a loadable kernel module. Therefore, significant extensions or bug fixes for this subsystem require going through the typical development cycle in the kernel: build the kernel, install it and restart the machine. Although PMCTrack requires minimal changes to the Linux kernel, the vast majority of its functionality is encapsulated in a loadable kernel module, as described in the next section. Thus, providing additional support to gather PMC or non-PMC related data (such as LLC occupancy) can be performed without rebooting the system. This fact greatly simplified the development and maintenance of PMCTrack. Moreover, in some cases, adding the support necessary to access new hardware monitoring facilities leads to a simpler implementation to that of the perf event subsystem. We elaborate on this aspect in Section 4.3.

3. DESIGN

This section outlines PMCTrack's internal architecture as well as the various supported usage modes.

3.1. Architecture

Figure 2 depicts PMCTrack internal architecture. The tool consists of a set of user and kernel space components. At a high level, the end user interacts with PMCTrack using the available command-line tools or PMCTrack-GUI (a graphical frontend). Alternatively, applications may access PMCTrack's functionality directly via the *libpmctrack* user space library (see Section 3.2.2). These components communicate with PMCTrack's kernel module by means of a set of Linux/proc entries exported by the module.

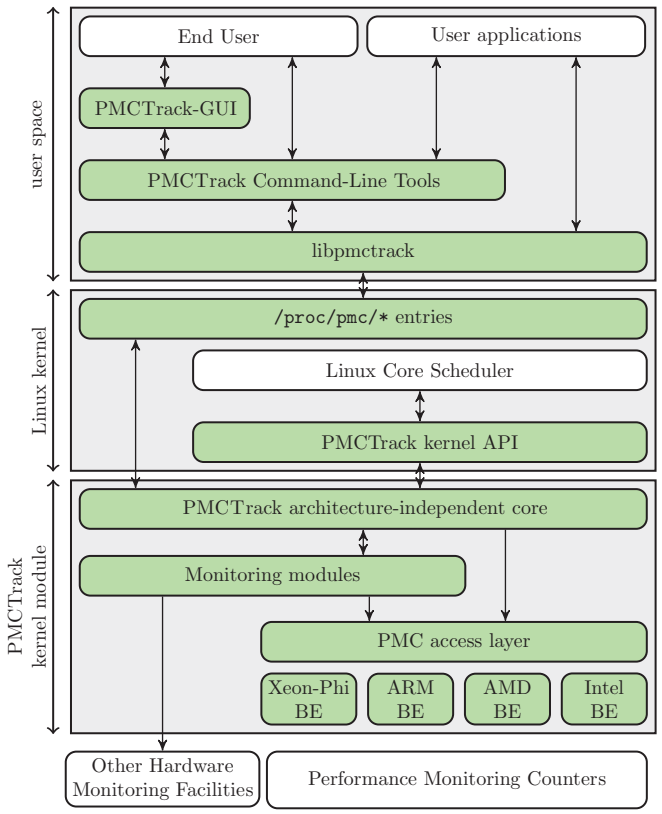

FIGURE 2. PMCTrack architecture.

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/termination). In addition to exposing an application's performance counter data to the userland tools, the module implements a simple mechanism to feed with per-thread monitoring data to any scheduling policy (class) that requires performance-counter information to function; such a mechanism is described in Section 3.2.1. Because both the core Linux Scheduler and scheduling classes are implemented entirely in the kernel, making PMCTrack's kernel module aware of thread-related events and requests from the OS scheduler requires some minor modifications to the Linux kernel itself. These modifications, referred to as PMCTrack kernel API in Fig. 2, comprise a set of notifications issued from the core scheduler to the module. To receive key

[1] https://community.arm.com/docs/DOC-9321

notifications, PMCTrack's kernel module implements the following interface:

```
typedef struct pmc_ops{
  /* invoked when a new thread is created */
  void* (*pmcs_alloc_per_thread_data)(unsigned long,
       struct task_struct*);
  /* invoked when a thread leaves the CPU */
  void (*pmcs_save_callback)(void*, int);
  /* invoked when a thread enters the CPU */
  void (*pmcs_restore_callback)(void*, int);
  /* invoked every clock 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 a thread exits the system */
  void (*pmcs_exit_thread)(struct task_struct*);
  /* invoked when a thread's 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* task, int key, uint64_t* value);
} pmc_ops_t;
```

Most of these notifications are 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 a specific application.

As shown in Fig. 2, PMCTrack's kernel module consists of various components. The architecture-independent core, implements the `pmc_ops_t` interface and interacts with PMCTrack userspace components via the Linux *proc* file system. PMCTrack's kernel module also provides an API to build *monitoring modules*. As stated above, the primary purpose of a monitoring module is to provide a scheduling algorithm that is implemented in the kernel with high-level performance metrics or other insightful runtime information that is potentially exposed by the hardware (via PMCs or by other means), such as power/energy consumption or a process's LLC occupancy. In addition, a monitoring module may expose this information to PMCTrack's userspace components by means of *virtual counters*. Notably, both the kernel module's upper layer and the monitoring modules rely on the architecture-agnostic PMC access layer to perform low-level access to performance counters, as well as to translate PMC configuration strings into internal data structures for the platform in question. In turn, the platform-specific support is encapsulated in a set of Performance Monitoring Unit Backends (PMU BEs). At the time of writing, PMCTrack provides four BEs with the necessary support for most modern Intel and AMD processors, for some ARM Cortex processor models and for the Intel Xeon Phi Coprocessor.

Augmenting the Linux kernel to support PMCTrack entails adding two new source files to the kernel tree

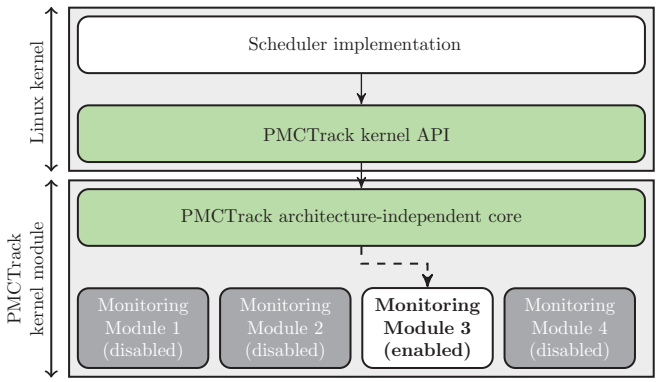

FIGURE 3. PMCTrack monitoring modules.

(implementation of the kernel API), and adding <20 lines[2] of code to the core scheduler sources. Thus, the changes required by PMCTrack can be easily applied to different kernel versions.

3.2. Usage modes

PMCTrack can be used to gather performance counter data from the OS scheduler using an in-kernel interface, (scheduler mode) and from userspace in various ways.

3.2.1. Scheduler mode

This mode 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 runtime properties. Turning on this mode for a particular thread from the scheduler's code simply involves 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 benefits from this feature remains architecture independent, the scheduler itself (implemented in the kernel) does not deal with performance counters or hardware events directly, but instead requests the necessary per-thread high-level performance monitoring metrics from a platform-specific *monitoring module*. As shown in Fig. 3, PMCTrack may include several monitoring modules that are 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 that several compatible monitoring modules are available, the system administrator may tell the system which one to use by writing in the `/proc/pmc/mmon_manager` file. (In

[2]The line count corresponds to the necessary changes in the Linux kernel v2.6.38 and above. In this paper, we have experimented with different Linux versions: 3.2, 3.10 and 4.1.

Section 4.1, we consider the scenario of having different monitoring modules for various scheduling algorithms.) 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, which is 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 the collection of 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 the `monitoring_module_t` interface, which features very similar notifications to those found in `pmc_ops_t`. Specifically, it consists of several callback functions that make it possible 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. Nevertheless, the programmer typically implements 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, when a thread enters the system the monitoring module can impose the set(s) of performance events to be monitored by using the `configure_performance_counters_set()` function, which accepts an argument that encodes the desired counter configuration in a string. (The associated format is discussed in Section 3.2.2.) Whenever new PMC samples are collected for a thread (i.e. the sampling period expires), a callback function of the monitoring module is invoked, passing the samples as a parameter. Due to this feature, a monitoring module will only access monitoring-related registers (or sensors) in the event that it is responsible for providing the OS or the end user with other hardware monitoring information not modeled as PMCs, such as energy consumption readings.

3.2.2. *Using PMCTrack from user space*

In addition to the in-kernel mechanism presented above, PMCTrack also enables the gathering of PMC data from user space by using the `pmctrack` command-line tool, the PMCTrack-GUI application, and `libpmctrack`.

To support user-oriented monitoring, PMCTrack's kernel module stores performance and virtual counter values using ring buffers. Userspace tools retrieve samples from ring buffers by reading from a/proc file that blocks the monitor process till new samples are generated or the monitored application terminates.

A) The pmctrack command-line tool. This tool allows the user to gather an application's performance data at regular time intervals (a.k.a., time-based sampling — TBS) or when a certain event count reaches a specified threshold (a.k.a., event-based sampling — EBS). Notably, both modes support monitoring multithreaded and single-threaded applications and provide information on performance counters as well as on any monitoring information exposed by the active monitoring module as a virtual counter, such as energy consumption readings.

More recently, we augmented the `pmctrack` tool with a system-wide TBS mode enabling the gathering of monitoring information on a per-CPU basis rather than on a per-application basis. A full description of the various command-line options supported by the `pmctrack` tool, as well as several usage examples, can be found on PMCTrack's official website [28].

To illustrate how the `pmctrack` tool works, let us consider the following sample command for the TBS (default) mode:

```
$ pmctrack -c instr,llc_misses ./mcf06
[Event-to-counter mappings]
pmc0=instr
pmc3=llc_misses
[Event counts]
nsample      pid      event          pmc0         pmc3
      1      7008       tick    1961001132       110634
      2      7008       tick    1247853112         8323
      3      7008       tick    1230836405         3859
      4      7008       tick    1358134323       409386
      5      7008       tick    1280630906      1199270
      6      7008       tick    1231578609     15488307
...
```

This command provides the user with the number of instructions retired and LLC misses every second (default setting for the configurable sampling period) on a system featuring a quad-core Intel Xeon Haswell processor. To indicate the sets of hardware events to monitor, the -c option must be used. As is evident, the command-line tool makes it possible to specify counter and event configurations using mnemonics in much the same way as other userspace-oriented tools [2,3,18]. The beginning of the command output shows the event-to-counter mapping for the various hardware events. The 'Event counts' section in the output displays a table with the raw counts for the various events; each sample (one per second) is represented by a different row. At the end of the line, we specify the command to run the associated application we wish to monitor (e.g. ./mcf06).

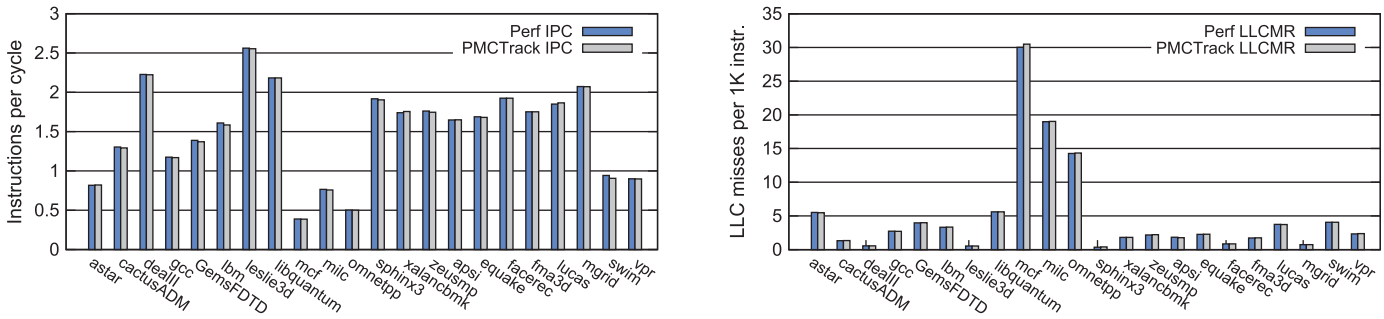

FIGURE 4. Average IPC (left) and LLCMR (right) for a subset of SPEC benchmarks collected with perf and PMCTrack.

Clearly, using mnemonics to indicate the sets of hardware events to be monitored, as in the example above, may prove suitable for inexperienced users, as the low-level event codes and the event-to-physical-counter mappings do not have to be specified. Despite the simplicity of this format, using mnemonics is not well suited to in-kernel event monitoring as translating mnemonics into the actual hex values written in PMC registers may involve traversing rather long event tables.[3] To avoid the associated overhead, monitoring module developers must specify event configurations using a lower-level string format, referred to as the *raw* format. (For example, the raw string `pmc0,pmc3=0x2e,umask3=0x41` would make it possible to gather the same hardware events as in the example above on a modern Intel processor.) Nevertheless, monitoring module developers may turn to PMCTrack's `pmc-events` helper command to obtain raw configurations strings from mnemonic-based representations and to retrieve low-level PMU parameters and event listings.

To assess the accuracy of the performance metrics gathered via the `pmctrack` command-line tool, we collected the IPC and LLC miss rate (LLCMR) for the entire execution of diverse memory-intensive programs from the SPEC CPU2006 and CPU2000 suites using both PMCTrack and perf [18]. For the experiment we employed TBS as perf is also equipped with this capability. Figure 4 shows the values for the different metrics obtained on an Intel 'Haswell' Xeon E3–1225 v3 processor by means of PMCTrack and perf. As is evident, the results illustrate that both tools report very similar values for the metrics. We also measured the overhead from TBS using sampling periods ranging from 100 ms to 1 s and found that both PMCTrack and perf yield similar and almost negligible overheads in our setting (up to 1% when using the smallest value).

Note that in the scenarios considered in Sections 4.1.2 and 4.2.2 we make extensive use of the EBS feature of

[3]In some userspace-oriented tools such as perf [18], event tables for every supported processor model are hard-coded in the kernel. To avoid the need for rebuilding the entire OS kernel every time a new event is added to a table, PMCTrack conveniently stores the tables in files manipulated in userspace.

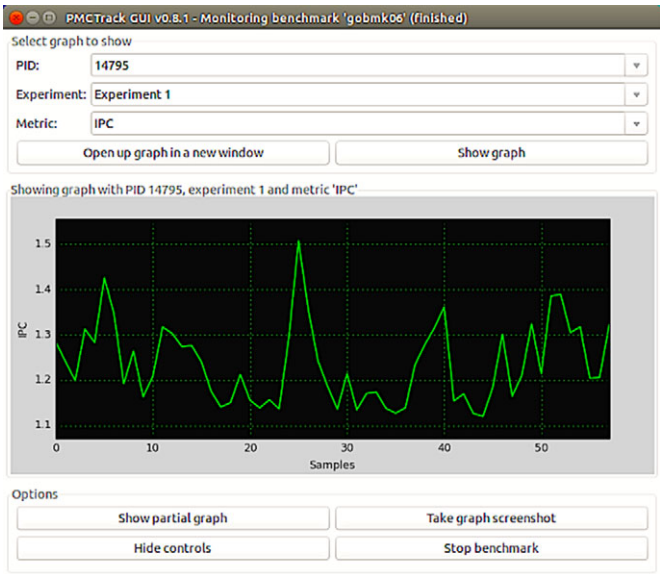

FIGURE 5. PMCTrack-GUI.

PMCTrack, which makes it possible to collect application monitoring information for individual instruction windows. This information makes it possible to compare the performance and energy efficiency of individual program phases in an application when it runs on different core types of an asymmetric single-ISA multicore system.

B) PMCTrack-GUI. To complement the `pmctrack` command-line tool with real-time visualization of high-level performance metrics (such as the IPC or the LLCMR) we also created PMCTrack-GUI, a Python front-end for `pmctrack`. Figure 5 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.

C) Libpmctrack. This userspace library enables the characterization of the performance of code fragments via PMCs in sequential and multithreaded programs. To this end, libpmctrack's API offers a set of calls to indicate the desired PMC

configuration to the kernel module at any point in the application's code. The programmer may then retrieve the associated event counts for any code snippet (either via TBS or EBS) simply by enclosing the code between invocations to the `pmctrack_start_counters()` and `pmctrack_stop_counters()` functions. Overall, *libpmctrack* provides a similar functionality to that of PAPI-C [29]. The main advantage over PAPI-C is that *libpmctrack*'s API also makes it possible to retrieve virtual counter values associated with code fragments. More importantly, due to the simple mechanism employed by *libpmctrack*'s API to access virtual counters, libpmctrack requires no modifications to benefit from new hardware monitoring support. Specifically, a kernel-level monitoring module is in charge of implementing the associated support and exposing the new features to the library.

In Section 4.1 we demonstrate the benefits of PMCTrack's scheduler mode in the context of single-ISA asymmetric multicore systems. In this scenario, the OS scheduler can effectively use per-thread PMC-related information to drive scheduling decisions. For multithreaded applications, optimizations at the OS level can also be complemented with scheduling decisions at the runtime level, such as asymmetry-aware load balancing among worker threads [30]. Because libpmctrack can be used at the runtime level (user space) to access per-thread PMC and virtual-counter data, PMC-related online optimizations could be performed in the context of runtime systems of parallel programming models such as Cilk or OpenMP. Exploring these optimizations constitutes an interesting avenue for future work.

4. CASE STUDIES

We now demonstrate the potential of PMCTrack in three different scenarios: OS scheduling for asymmetric single-ISA multicore systems, energy/power consumption monitoring and cache-usage monitoring.

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 upon the energy and power efficiency of their symmetric counterparts [31]. The ARM big.LITTLE processor [32] and the Intel Quick-IA prototype [33] 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. Most existing scheduling schemes have focused on maximizing the system throughput for multi-application workloads [9, 10, 15, 34, 35, 31]. To this end, the scheduler must follow the *HSP* (High-SPeedup) approach, meaning it must preferentially use big cores for those applications that derive a greater benefit (*speedup*) from running on big cores. Note that for a single-threaded (ST) program, the speedup matches the *speedup factor* (SF) of its single runnable thread, defined as $\frac{\text{IPS}_{\text{big}}}{\text{IPS}_{\text{small}}}$, where IPS_{big} and $\text{IPS}_{\text{small}}$ are the thread's instructions per second ratios achieved on big and small cores, respectively. The SF, however, does not approximate the overall speedup that a multithreaded application as a whole derives from using the big cores in an AMP [15, 36, 37]. Previous research [10, 38] has derived analytical formulas to approximate the speedup for several types of multithreaded applications based on the runnable thread count (a proxy for the amount of thread-level parallelism (TLP) in the application), the SF of the application threads, and the number of big cores in the AMP.

Schedulers that aim to maximize throughput alone, however, are known to be subject to QoS-related issues [17, 38]. For example, equal-priority applications may not experience the same performance penalty (slowdown) when running together relative to their performance when running alone on the AMP. This one and other related issues can be addressed via fairness-aware scheduling [12, 16, 17, 38]. Notably, fairness-aware schedulers also need to take per-thread SFs into account to ensure acceptable system throughput [12, 17]. This fact underscores that the ability to gather accurate SFs online plays an important role in the effectiveness of the asymmetry-aware scheduler regardless of its target objective.

Three different schemes have been explored to determine per-thread SFs online. The first approach basically consists in measuring SFs directly [17, 35, 31], which entails running each thread on big and small cores to track the IPC on both core types. Previous work has demonstrated that this approach, known as *IPC sampling*, is subject to significant inaccuracies in SF estimation associated with program-phase changes [34]. The second approach relies on *estimating a thread's SF* using its runtime properties collected on any core type at run time using performance counters [9, 10, 15]. Because this technique enables prediction of the SF from the *current* core type on which the thread is running, the mechanism does not suffer from the same program-phase related issues as IPC sampling. Unfortunately, estimating SFs via hardware counters requires derivation of an estimation model that is specifically tailored to the platform in question [9, 10]. The third technique is PIE (Performance Impact Estimation) [39], a hardware-aided mechanism enabling accurate SF estimation from any core type. It should be noted that, the required hardware support for PIE has not yet been adopted in commercial systems, and so scheduler implementations on existing asymmetric hardware, such as the ones we

considered in this work, cannot benefit from this approach. In addition, recent research has highlighted that PIE poses several problems that make it difficult to deploy on actual hardware [40].

To evaluate the effectiveness of different scheduling algorithms for AMPs, we built different PMCTrack monitoring modules capable of feeding the scheduler with per-thread SFs at run time. The modules rely on either IPC-sampling or employ a platform-specific estimation model to determine SFs as threads go through different program phases. Relying on monitoring modules to feed the scheduler implementation with per-thread SFs provides three important benefits. First, the scheduler implementation remains fully architecture independent. Second, because the scheduler obtains a thread's SF from the monitoring module in question via PMCTrack's kernel API, monitoring modules constitute fully replaceable components. This makes it possible to explore the effectiveness of various SF-enabled monitoring modules for the same scheduler implementation without rebooting the system. Third, since the SF can be seamlessly exposed as a virtual counter to PMCTrack's command-line tools, per-thread SF and PMC traces can be gathered from userspace for debugging purposes.

4.1.1. *Scheduling algorithms*

For our study, we considered several state-of-the-art schedulers for AMPs: HSP [9, 10], RR [41, 35], A-DWRR [16], EQual-Progress (EQP) [17] and ACFS[4] [12]. We implemented all the scheduling algorithms as a separate scheduling class in the Linux kernel 3.2.

As stated above, the **HSP** scheduler optimizes throughput by dedicating big cores to the execution of those application threads in the workload that obtain the greatest benefits from using big cores in the AMP. The main difference between the available variants of the HSP scheduler [9, 10, 15, 34, 35, 31] lies primarily in the mechanism employed to obtain the threads' SFs online. In our analysis, we found that using the monitoring module that estimates SFs via a platform-specific performance model provides better system throughput than using the one based on IPC sampling. Therefore, we opted to use SF estimation for HSP. To approximate the overall speedup for different types of multithreaded applications in HSP's implementation we used the formulas derived in previous work [15, 38], which factor in the amount of TLP of the application as well as the SF of its threads to approximate the overall speedup. By using such speedup approximations, HSP may map high-SF sequential parts of parallel applications to big cores and relegates high-TLP phases to small cores, which makes it possible to improve throughput [36, 42].

The first approach to fairness-aware scheduling on AMPs was an **asymmetry-aware Round-Robin (RR)** scheduler that simply fair-shares big cores among applications by performing periodic thread migrations [35]. Fair-sharing big cores has been shown to provide better performance on AMPs than default schedulers in general purpose OSes, which are largely asymmetry agnostic, and also provides more repeatable completion times across runs [16]. For this reason, RR has been widely used as a baseline for comparison [35, 41, 43]. In our experiments, we observed that RR does ensure repeatable completion time across runs for CPU-bound workloads (such as the ones we used). This is not the case of the default scheduler of the Linux mainstream kernel (Completely Fair Scheduler – CFS), which can map the same application to different core types in different runs of the same CPU-bound multi-program workload. These random thread-to-core mappings lead to unrepeatable fairness and throughput results across a series of runs for the same workload. As unrepeatable results may lead to misleading conclusions, we opted not to display the results associated with the default Linux scheduler in our experimental analysis (Section 4.1.3).

A-DWRR [16] aims to deliver fairness on AMPs by factoring in the computational power of the various cores when performing per-thread CPU accounting. To that end, it relies on an extended concept of CPU time for AMPs: *scaled* CPU time. Using scaled CPU time, the CPU cycles consumed on a big core *are worth more* than on a small one. To ensure fairness, A-DWRR evens out the scaled CPU time consumed across threads in accordance to their priorities. Like RR, A-DWRR does not take into account the fact that applications usually derive different (and possibly time-varying) speedups when using big cores on the platform. As our experimental results reveal, this leads both RR and A-DWRR to degrading both fairness and throughput.

Finally, the **EQP** [17] and **ACFS** [12] schedulers seek to optimize fairness on AMPs. Both schedulers leverage per-thread SF values to continuously track the slowdown that each thread in the workload experiences at run time and try to enforce fairness by evening out observed slowdowns. EQP and ACFS exhibit important differences. First, when determining a thread's slowdown, EQP does not factor in the past speedup phases that the thread underwent. Instead, the slowdown is approximated by taking into account the total number of cycles that the thread has consumed on each core type thus far, and the current SF. ACFS, on the contrary, maintains a per-thread counter that accumulates the thread's total progress based on the current and the past speedup application phases. Second, EQP relies on either IPC sampling or PIE [39] to estimate SFs, whereas ACFS relies on SF estimation using platform-specific models. Since PIE is not available on existing asymmetric hardware, in this work we evaluated the IPC-sampling variant of EQP. Third, EQP was designed to achieve equal slowdown across threads, and so it only takes into account the SF of individual threads when computing slowdowns. ACFS, by contrast, takes into account the application-wide speedup to guarantee *equal slowdowns among applications*. This feature makes it possible for ACFS to provide a better support when multithreaded applications are included in the workload.

[4] ACFS stands for Asymmetry-aware Completely Fair Scheduler.

Fourth, the ACFS scheduler supports user-defined priorities, whereas the EQP scheduler does not.

4.1.2. *Experimental setup and determining SFs online*

For the evaluation we used the Intel QuickIA prototype system [33]. This platform consists of a dual socket UMA system featuring a quad-core Intel Xeon E5450 processor and a dual-core Intel Atom N330 processor. To reduce shared resource contention effects for the experiments, we disabled one core on each die in the Xeon processor. This setting gives us a pairing of two high-performance *big* cores (E5450) with two low power *small* ones (N330). We will refer to this asymmetric configuration as 2B-2S.

To run workloads that include multithreaded applications we opted to use an AMP configuration with a greater core count than that of 2B-2S. To this end, we also experimented with a NUMA multicore server that integrates two AMD Opteron 2425 hex-core processors. On this platform we emulated an AMP system consisting of 2 big cores and 10 small ones (2B-10S) by reducing the processor frequency on some cores; specifically, 'big' cores on 2B-10S operate at 2.1 GHz whereas 'small' cores run at 800 MHz.

To feed the various schedulers with per-thread SFs we implemented three PMCTrack monitoring modules, which sample performance counters every 200 ms.[5] The first monitoring module provides the EQP scheduler with SFs approximated via the IPC sampling approach on both multicore servers. The other two monitoring modules, used by both HSP and ACFS, rely on platform-specific models to estimate SFs on the AMD and the Intel system, respectively.

To aid in the construction of such platform-specific SF estimation models in this work, we used a variant of the technique detailed in [10]. This technique requires certain offline processing, which can be summarized as follows. We first pick a representative set of single-threaded benchmarks (e.g. a subset of SPEC CPU benchmarks) and a comprehensive set of performance metrics that allow characterization of the microarchitectural and memory behavior of the various applications. We then run these benchmarks on big and small cores to monitor the aforementioned metrics via PMCs using fixed instruction windows; we used PMCTrack's EBS feature to do so. Using the information collected with PMCs, we identify coarse-grained program SF phases by matching the various PMC samples collected on both core types for contiguous instruction windows of the same application. (The SF of a certain application's instruction window can be obtained from the IPC values collected on both core types for that instruction window.) Finally, we use the per-application SF-phase data obtained in the previous step as input to the additive-regression estimation engine provided by the WEKA machine-learning package [44]. This engine enables us to generate two estimation models: one enabling the estimation of SFs from big-core metrics, and the other allowing approximation of SF values from small-core metrics. Notably, the additive-regression prediction engine automatically identifies irrelevant performance metrics for the model (low regression coefficients) and so these metrics can be discarded from the final estimation models. This significantly reduces the number of metrics collected online by PMCTrack's monitoring modules.

On the AMD platform, the methodology described above makes it possible to derive accurate estimation models that rely on a reduced set of performance metrics. In addition, these metrics can be monitored simultaneously using the four per-core general-purpose performance counters available on the platform, which greatly simplifies the implementation of the associated monitoring module in PMCTrack. The simplicity of the models obtained stems from the fact that cores in this AMP setting differ only in processor frequency; in this scenario cache and memory-related metrics such as the LLC miss or the LLC access rates are known to show a negative correlation with the SF [9, 15], as Fig. 6(a) reveals. Models for the AMD system achieve correlation coefficients of 0.97 and 0.96 when predicting the speedup for the SPEC CPU benchmarks on the faster and the slower core, respectively.

On the QuickIA prototype, the cores exhibit more profound differences (microarchitecture, cache sizes and processor frequency, among others) which lead to complex SF estimation models. Figure 6(b) shows that the LLCMR alone does not exhibit any correlation with the SF, as opposed to what can be observed when cores differ in processor frequency only. On the QuickIA platform, estimation models with acceptable accuracy that can be obtained via the described methodology require the monitoring of a higher number of hardware events than that of AMD models. Table 1 displays the set of performance metrics and associated hardware events that must be monitored on each core type by the PMCTrack monitoring module implementing the most accurate SF estimation modules we could obtain for this platform. These models achieve associated correlation coefficients of 0.95 (big core) and 0.94 (small core), respectively.

The higher complexity of the estimation models for the QuickIA platform coupled with the reduced set of per-core general-purpose performance counters on this system make it harder to implement the associated PMCTrack monitoring module. Specifically, on the Intel QuickIA, only three insightful performance metrics can be monitored simultaneously on any core type: the IPC, using the available fixed-function PMCs, and two other high-level metrics using general-purpose PMCs. Since the estimation models depend on more than three performance metrics, we need to use event multiplexing in the kernel to obtain SFs online. Thus, the associated PMCTrack monitoring module continuously monitors different sets of hardware events in a RR fashion. Specifically, to gather the necessary metrics on the QuickIA prototype, two PMC sampling periods are required for the big-core model and three for the small-core model. Once all the required performance metrics have been gathered on a

[5] We observed that the overhead associated with PMC sampling and determining the SF becomes negligible at this rate.

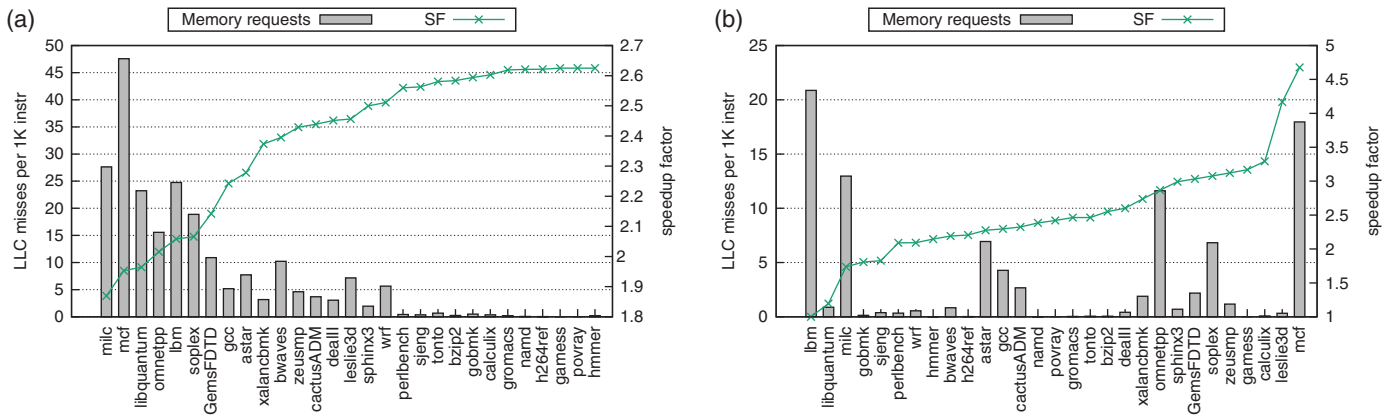

FIGURE 6. SF vs. big-core LLCMR for SPEC CPU 2006 benchmarks observed on the (a) AMD system and (b) on the Intel platform.

TABLE 1. Performance metrics and associated hardware events used to predict the SF on the Intel QuickIA prototype.

Core	Hardware Events	Performance metrics
Big	Instructions retired, processor cycles, L2 cache accesses, L2 (last-level) cache misses, Branch instructions retired	Instructions per cycle, L1 data cache misses per 1K instr., L1 data TLB misses per 1M instr., Mispredicted branches per 1K instr., L2 (last-level) cache access per 1K instr., L2 (last-level) cache misses per 1K instr., Branch instructions retired per 1K instr.
Small	Instructions retired, processor cycles, L2 cache accesses, L2 (last-level) cache misses, Branch instructions retired Mispredicted branches, ITLB misses, DTLB misses	Instructions per cycle, L1 data cache misses per 1K instr., L1 data TLB misses per 1M instr., Mispredicted branches per 1K instr., L2 (last-level) cache access per 1K instr., L2 (last-level) cache misses per 1K instr., Branch instructions retired per 1K instr. Mispredicted branches per 1K instr. ITLB misses per 1M instr. DTLB misses per 1M instr

certain core type using the necessary sampling periods, the metric values are used to predict the thread's SF. This process is repeated continuously throughout the thread's execution to feed the scheduler with up-to-date SF estimates.

We found that an important issue becomes apparent when implementing such a scheme. Performance metrics monitored in subsequent PMC sampling periods on a given core may not belong to the same program phase. Notably, using event metric values from different phases leads to inaccurate SF predictions. To overcome this issue, we augmented the prediction scheme with a heuristic to detect transitions between program phases. This heuristic is similar to that proposed in [35], which was evaluated in a simulation environment. At a high level, the heuristic works as follows. For each thread, we maintain a running average of the IPC. To make this possible, we always monitor the number of IPC together with the other metrics in a particular event set. If a sudden variation of the IPC is detected in the last sample (with respect to the running average of the IPC), we assume that the sample belongs to a new program phase. If a sample for a different program phase is detected, previously collected samples from the same sampling round are discarded when estimating the SF, and a new sampling round is started. By contrast, if an entire sampling round is completed without detecting phase transitions, then collected samples are used to generate an up-to-date SF value with the associated estimation model. For the sake of clarity, Fig. 7 depicts how this mechanism works in a hypothetical scenario where three different performance metric

sets (three sampling periods) are required to obtain the samples necessary to estimate the SF. We observed that this heuristic proves effective in detecting phase transitions, and, in turn, leads to more accurate SF estimates over time. Note that implementing this scheme requires to control event multiplexing in an explicit way from the PMCTrack monitoring module. To make that happen the monitoring module developer has to employ a function of PMCTrack's API to specify the set of events to be monitored for a complete sampling round for each thread on each core type (the event sets may differ). When a sampling period completes, PMCTrack invokes a callback of the monitoring module, passing the samples obtained as a parameter. As stated in Section 2, the perf events Linux subsystem is not equipped with similar explicit in-kernel event-multiplexing support, thus making it impossible to implement this kind of scheme.

To conclude the discussion, it is worth noting that the offline analysis required to derive the SF estimation models has to be performed just once on a given asymmetric platform. Thus, to deploy the mechanism transparently in a production system, the associated experiments could be automatically launched by the OS when it boots for the first time. Alternatively, estimation models can be automatically rebuilt periodically by leveraging idle system periods to collect new performance samples for additional applications. In our experiments, we carried out a full execution of SPEC CPU benchmarks on the different core types of the system, in order to perform a comprehensive characterization of the performance of these benchmarks on each evaluated AMP configuration. However, running the entire SPEC CPU benchmark suite can be a time-consuming process on some *slow* cores (Intel Atom), so it may not constitute a good approach to be performed upon the first OS boot. Nevertheless, our analysis reveals that a complete execution of the SPEC CPU benchmarks is not really necessary to build the SF estimation models. Specifically, the vast majority of the representative program phases for most long-running benchmarks become apparent at the beginning of the execution (such as for `lbm`). Conversely, information from multiple sample phases to build the models can be collected by running a few short phased benchmarks, such as `soplex` or `astar`. Hence, collecting information for just a small number of instruction windows for a few selected benchmarks would suffice to obtain similar models, thus constituting a more convenient option. Finally, it is worth highlighting that in the experimental evaluation shown in the next section we used additional applications (from SPEC CPU and other benchmarks suites) that are different from those used to generate the estimation models.

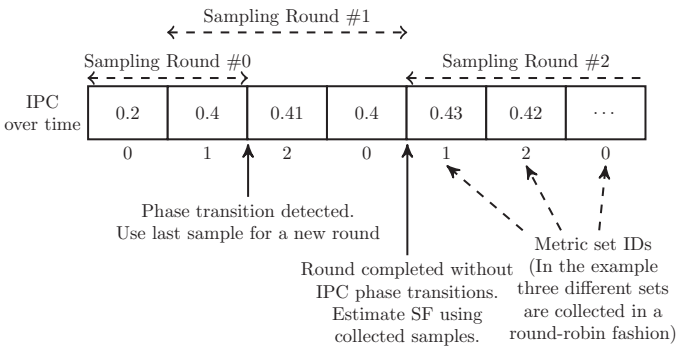

FIGURE 7. Dealing with program phase changes when using event multiplexing.

4.1.3. *Experimental evaluation*

Our evaluation targets multi-application workloads consisting of benchmarks from diverse suites (SPEC CPU2006, SPEC OMP, PARSEC and Minebench). We also experimented with `BLAST`—a bioinformatics benchmark, and `FFTW3D`—a program performing the FFT. In all the experiments, the total thread count in the workload was set to match the number of cores on the platform, since this is how runtime systems typically set the number of threads for CPU-bound workloads [45], such as the ones we used. In multi-application experiments, we ensure that all applications are started simultaneously, and when an application terminates it is restarted repeatedly until the longest application in the set completes three times. We then obtain the aggregate speedup and unfairness for the scheduler in question, by using the geometric mean of the completion times for each program.

To quantify throughput under the various schedulers we used the *Aggregate SPeedup* (ASP) metric, employed in previous work [12, 38]:

$$\mathrm{ASP} = \sum_{i=1}^{n} \left(\frac{\mathrm{CT}_{\mathrm{slow},i}}{\mathrm{CT}_{\mathrm{sched},i}} - 1 \right) \tag{1}$$

where n is the number of applications in the workload, $\mathrm{CT}_{\mathrm{slow},i}$ is the completion time of application i when it runs alone in the AMP and uses small cores only, and $\mathrm{CT}_{\mathrm{sched},i}$ is the completion time of application i under a given scheduler. The ASP metric captures the overall efficiency that a workload derives from the various cores in an AMP under a particular scheduler.

Regarding *fairness*, we employ the notion, widely used in the context of CMPs [46–48] and more recently in that of AMPs [17,38], that a scheme is fair if equal priority applications suffer the same slowdown due to sharing the system with respect to the situation in which the whole system is available to each application. To cope with this notion of fairness, we used the lower-is-better *unfairness* metric [48]:

$$\mathrm{Unfairness} = \frac{\mathrm{MAX}(\mathrm{Slowdown}_1, \ldots, \mathrm{Slowdown}_n)}{\mathrm{MIN}(\mathrm{Slowdown}_1, \ldots, \mathrm{Slowdown}_n)} \tag{2}$$

where $\mathrm{Slowdown}_i = \mathrm{CT}_{\mathrm{sched},i}/\mathrm{CT}_{\mathrm{fast},i}$, and $\mathrm{CT}_{\mathrm{fast},i}$ is the completion time of application i when running alone in the AMP

(with all the big cores available). As opposed to other metrics, the unfairness metric can be easily extended to factor in application priorities or weights by replacing an application slowdown with its weighted counterpart [12, 48].

In creating the multiprogram workloads for our study, we categorized applications into three groups relating to their parallelism: highly parallel (HP), partially sequential (PS)—parallel programs with a serial component of over 25% of the total execution time—and ST. We further divided these three application groups into three subclasses based on their SFs—high (H), medium (M) and low (L). The 16 selected program mixes, shown in Tables 2 and 3, mimic scenarios with different SF ranges and varying degrees of competition for the scarce big cores in the AMP. Note that the workload name encodes the category of each application. For example, in 2PSH-1HPM, `BLAST` and `semphy` are PSH applications and `wupwise_m` is an HPM program. Note that the application categories specified in the table match those observed on the AMP system on which we ran the workload. Workloads in Table 2 are evaluated on the 2B-2S system; workloads in Table 3 were run on 2B-10S. For multithreaded applications, the number in parentheses by each program's name is the number of threads it runs with.

We begin by analyzing the effectiveness of the various schedulers for workloads consisting of multiple ST applications (Table 2) running on 2B-2S (Intel QuickIA). The results are shown in Fig. 8. In creating these workloads we opted to explore diverse program mixes with a different combination of application SF classes (H, M and L). Workloads are displayed in descending order according to the number of HSP applications. Note that workloads in the middle of the graph exhibit a higher diversity of speedup factors; these workloads constitute favorable scenarios for throughput-optimized scheduling (HSP), as pointed out in [34], as such fairness-aware schedulers may be subject to higher throughput degradation in these scenarios.

Overall, the scheduler that optimizes throughput (HSP) effectively obtains the best ASP across the board, but that comes at the expense of delivering the worst unfairness numbers (the higher, the worse). Previous work has analytically demonstrated that fairness and system throughput constitute conflicting objectives on AMPs [38], and our experimental results clearly exhibit this trend.

The results also exhibit very different throughput and fairness figures for fairness-aware schedulers. For example, both RR and A-DWRR fair-share big cores among threads in this scenario without taking into account their SFs, so these schemes perform similarly in most cases. Fair-sharing big cores makes it possible to reduce unfairness while achieving acceptable throughput for some SF-homogeneous workloads (e.g. 4STH, 3STH-1STM and 2STH-2STM). However, for workloads exhibiting a wide range of big-small speedups (e.g. 3STH-1STL and 2STH-2STL) A-DWRR and RR are subject to throughput and fairness degradation. This fact underscores the importance of factoring in the SF when making scheduling decisions on AMPs.

ACFS and EQP aim to optimize fairness by evening out the progress made by the various threads in this scenario. To make this happen, both schedulers take the threads's SF into consideration. Note that the results demonstrate that ACFS achieves the best unfairness figures across the board while yielding better system throughput than EQP for most workloads. We found that these throughput and fairness differences stem from: (i) the different mechanism employed by the two schedulers to keep track of the threads' slowdowns; and (ii) the fact that they rely on different schemes to determine the

TABLE 2. Multi-application workloads consisting of single-threaded applications.

Workload	Benchmarks
4STH	`calculix, gamess, GemsFDTD, bzip2`
3STH-1STM	`calculix, GemsFDTD, bzip2, h264ref`
3STH-1STL	`gamess, GemsFDTD, bzip2, sjeng`
3STH-1STL$_B$	`calculix, gamess, sphinx3, sjeng`
2STH-2STM	`gamess, soplex, povray, h264ref`
2STH-2STL	`mcf, calculix, sjeng, gobmk`
1STH-1STM-2STL	`mcf, h264ref, sjeng, gobmk`
2STM-2STL	`namd, h264ref, gobmk, libquantum`

TABLE 3. Workloads consisting of single-threaded and multithreaded applications.

Workload	Benchmarks
2STH-1HPH-1HPM	`gamess, hmmer, fma3d_m`(5), `wupwise_m` (5)
3STH-1HPH	`hmmer, gobmk, h264ref, fma3d_m`(9)
3ST{H,M,L}-1PSH	`gamess, astar, soplex, blackscholes`(9)
1STH-1PSH-1PSL	`gobmk, BLAST`(6), `FFTW3D`(5)
2PSH-1PSL	`BLAST`(4), `semphy`(4), `FFTW3D`(4)
1PSH-1PSL	`semphy`(6), `FFTW3D`(6)
2PSH-1HPM	`BLAST`(4), `semphy`(4), `wupwise_m`(4)
1PSH-1HPH	`BLAST`(6), `fma3d_m`(6)

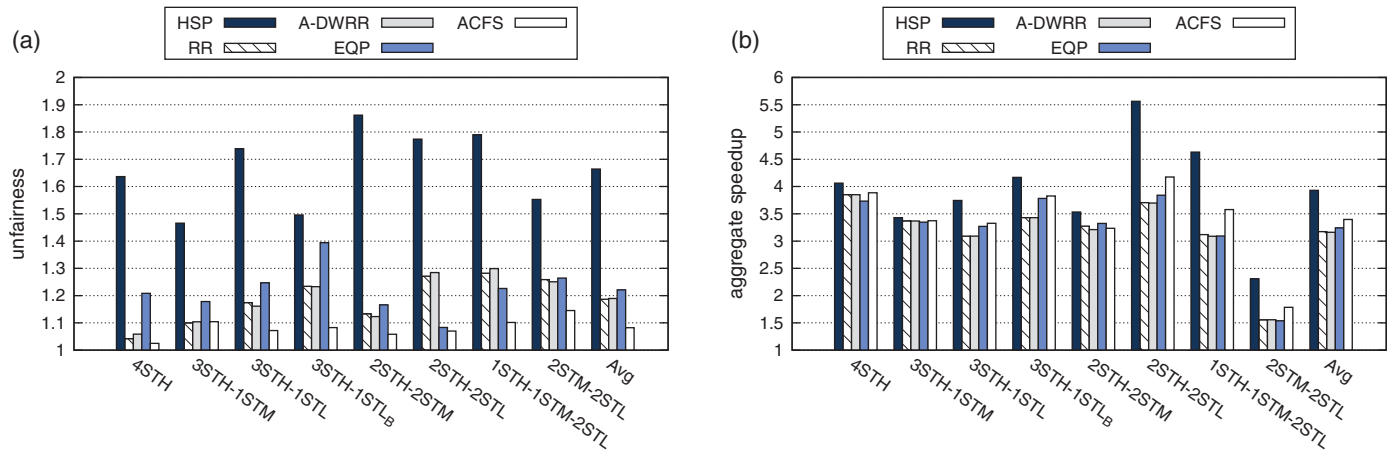

FIGURE 8. Fairness and throughput on 2B-2S (Intel QuickIA).

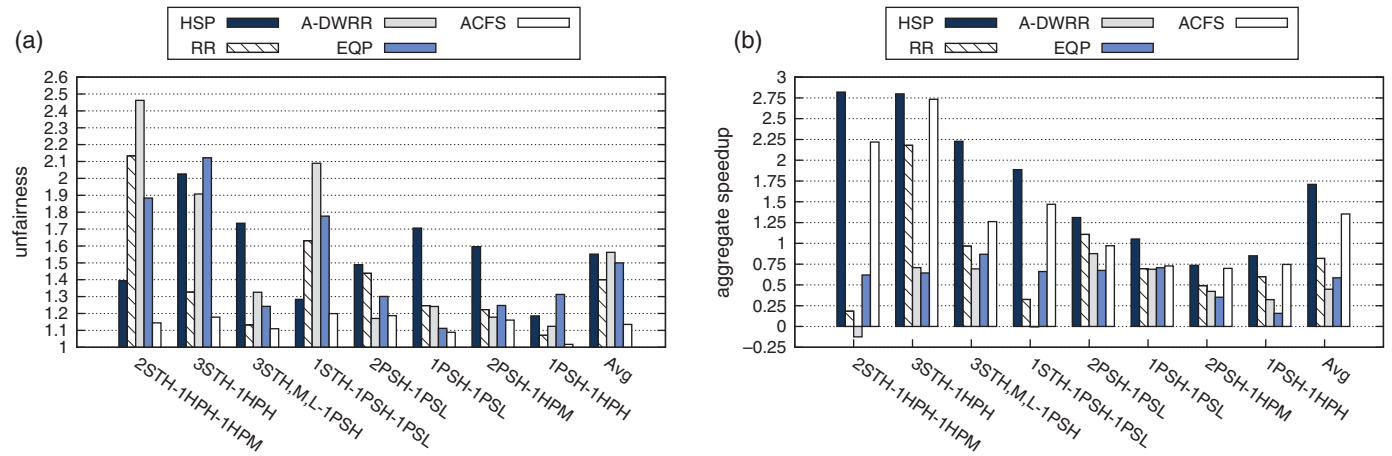

FIGURE 9. Fairness and throughput on 2B-10S (AMD platform).

threads' SFs at run time. Specifically, EQP relies on IPC sampling to determine threads SFs on off-the-shelf AMPs. IPC sampling, however, has been shown to lead to inaccurate SFs, since IPC values collected on both core types may belong to different program phases [34]. As a result, the IPS ratio used to approximate the SF may not always reflect an accurate value. We observed that these inaccuracies become apparent especially in the first five workloads, thus leading to frequent suboptimal thread-to-core assignments. In contrast, the ACFS scheduler is not subject to the aforementioned program-phase issues, since it is fed with predicted per-thread SF values obtained by a PMCTrack monitoring module that relies on estimation models. These models rely on performance metrics collected on the *current core type*. The results reveal that, despite the existing imperfections in the SF model, ACFS is able to obtain the best unfairness figures across the board. On average it reduces unfairness by 10% compared to RR and A-DWRR, and by 13% compared to EQP, while ensuring better system throughput than these schemes.

Now we look at the results for workloads in Table 3 (shown in Fig. 9), which include parallel and sequential applications running on 2B-10S (AMD platform). In selecting the workloads we chose program mixes with different speedup ranges and with varying degrees of competition for the two big cores in this AMP. Note that in this scenario there is a wide big-to-small speedup range across applications since some workloads combine parallel applications that derive almost no speedup from using the scarce big cores (such as HPL or HPM applications) with other applications that experience significant performance gains from these cores (such as ST applications). Therefore, mapping applications with a higher sequential component to a big core and dedicating big cores to running parallel (high-TLP) phases in the application leads to improved throughput [36]. Clearly, the application-wide speedup (as opposed to the per-thread SF) must be taken into consideration when it comes to optimizing throughput in this context. Note also that applications that exibit both parallel phases and non-negligible serial execution

phases (PS applications) play an important role in this scenario, as the speedup of these applications varies significantly over time with changes in the number of active threads [10, 36]. For this reason, we opted to include these applications in the workloads.

For HSP and ACFS, the results in Fig. 9 show similar trends to those of workloads on 2B-2S: ACFS achieves the best fairness figures across the board while HSP obtains better throughput than ACFS at the expense of significant fairness degradation. In this scenario, both schedulers make scheduling decisions by taking into account the speedup that the application as a whole derives from using the big cores available on the AMP. Thus, applications with a higher sequential component usually receive a higher big-core share under these schedulers.

We also observe that HSP does not always yield the worst unfairness numbers. For example, RR yields higher unfairness than HSP for the 2STH-1HPH-1HPM workload. We found that this is primarily due to the enormous difference in big-to-small speedups among applications present in this workload; both HP applications derive almost no benefit from utilizing the two big cores available, whereas the sequential applications exhibit the maximum overall big-to-small speedup attainable on this platform. Similarly, applications in the 1STH-1PSH-1PSL and the 1PSH-1HPH workloads exhibit program phases with such a high diversity in speedups. Under these circumstances, fair-sharing big-cores among applications (RR) leads to produce higher unfairness and lower throughput than that resulting from mapping high-speedup applications to big cores (HSP). Similarly, the A-DWRR and EQP algorithms, which make scheduling decisions on a per-thread basis, obtain higher unfairness than HSP in the aforementioned cases due to failing to allot a higher big-core share to applications in the workload with a higher sequential component.

We now focus on the EQP/ACFS comparison. Unlike ACFS, EQP does not take into account the application-wide speedup. Specifically, EQP aims to enforce equal slowdown across threads by considering the SF of individual threads only. Nevertheless, ensuring that each thread in the system experiences a similar slowdown does not ensure *equal slowdowns among applications* when multithreaded programs are included in the workload. Failing to consider the application-wide speedup leads EQP to higher fairness degradation than ACFS. In this scenario, ACFS achieves a 24% average reduction in unfairness relative to EQP, and yields a higher throughput than this scheme.

Finally, we look at the results of A-DWRR and RR on 2B-10S. A-DWRR ensures that each thread in the workload receives the same AMP-scaled CPU time, regardless of the application it belongs to; therefore, programs with a high thread count receive a high big-core share. RR, by contrast, fair-shares big cores among applications. Because applications with a high thread count usually derive low speedup from the scarce big cores, A-DWRR yields higher throughput and greater fairness degradation than RR. However, the RR scheduler exhibits worse fairness and throughput figures than ACFS in this scenario; ACFS improves fairness by 19% on average with respect to RR.

4.2. Measuring power and energy consumption

In this section we begin by illustrating the ability of PMCTrack to provide power and energy consumption readings on different processor models and architectures. Then, we explore the potential benefits of an energy-aware scheduling scheme for asymmetric multicores by leveraging combined information from performance counters and energy consumption registers gathered with PMCTrack.

4.2.1. Power consumption on different processor models

PMCTrack has the ability to interact with power and energy measurement facilities available on modern high-performance Intel processors [21] and on systems integrating low-power ARM big.LITTLE processors [32]. The necessary support is provided by a set of monitoring modules.

On Intel systems, PMCTrack relies on the Running Average Power Limit (RAPL) support [21]. RAPL employs a software power model based on hardware monitoring to approximate energy usage. This feature enables the system software to obtain energy consumption readings for different power domains (core-level, processor package/uncore and DRAM). Intel processors expose the RAPL facilities to the system software via a set of MSRs [21]. Because the Intel QuickIA system [33] used in the previous case study does not feature RAPL capabilities, we opted to experiment with a 14-core Intel Xeon E5 'Haswell' 2695 v3 processor. This processor supports energy consumption readings for the processor package (or core cluster) level as well as for the DRAM level.

Currently, PMCTrack also provides support for measuring energy consumption on various ARM boards featuring 32-bit and 64-bit ARM big.LITTLE processors, such as the ARM CoreTile Express TC2 board [26] and the ARM Juno Development board [27]. The main features of both systems are summarized in Table 4. The ARM CoreTile Express TC2 system is equipped with a daughter board that provides the system software with access to sensors to measure voltage, current, power and energy consumption for each cluster of cores of the same type (either Cortex A15 *big* cores or Cortex A7 *little* cores). On the Juno board, an FPGA (referred to as IOFPGA) provides the system software with a set of energy registers to retrieve the instantaneous current consumption, instantaneous power consumption and cumulative energy consumption for both clusters of Cortex A57 (big) and Cortex A53 (little) cores. Moreover, additional registers exist to provide similar measurements for the GPU and the rest of the Juno SoC fabric—referred to as SYS—, which includes the DRAM controller, among other components. Note that

TABLE 4. Features of evaluated platforms.

Platform	Processor model(s)	Core count	LLC	Main memory
Superserver SYS-6018R-MTR Supermicro	Intel Xeon E5–2695 v3 @ 2.3 GHz	14	35 MB (L3)	32 GB DDR4 @ 2133 MHz
ARM CoreTile Express TC2 Development board	ARM Cortex A15 @ 1 GHz	2	1 MB (L2)	2 GB DDR2 @ 400 MHz
	ARM Cortex A7 @ 800 MHz	3	512 KB (L2)	
ARM Juno Development board	ARM Cortex A57 @ 1.10 GHz	2	2MB (L2)	8 GB DDR3 @ 800 MHz
	ARM Cortex A53 @ 850 MHz	4	1MB (L2)	

accessing energy measurement facilities on these systems from the OS kernel is completely board-specific as the hardware exposes these facilities to the system software in very different ways.

To demonstrate the capabilities for accessing energy measurement facilities on different platforms, we collected the average power consumption for benchmarks in the SPECCPU 2006 suite running on five different processor models: Intel Xeon E5 'Haswell' 2695 v3, ARM Cortex A15, Cortex A7, Cortex A57 and Cortex A53. Figure 10(a)–(e) display the results. In all cases, we collected the package (or core cluster) average power consumption observed for the entire execution of each benchmark running alone on the system. In addition, for those platforms that make it possible to obtain DRAM power consumption measurements, we also display the associated values together with the cluster power measurements. Note, however, that on the ARM Juno Development board (Cortex A57 and A53) there is no separate energy register to measure the DRAM power consumption alone. So, in this case we approximated the net DRAM energy consumption by substracting the observed idle power consumption (0.81W) of the Juno SoC fabric from the power measurement provided by the SYS energy register.

The results reveal that big ARM Cortex out-of-order A15 cores yield up to 4.9× the power consumption of small ARM Cortex A7 in-order cores. Despite the fact that big cores on both ARM platforms run at a similar frequency in our experiments, Cortex A15 cores consume roughly three times the power of Cortex A57 cores. As for the small in-order cores in both development boards, we observe that Cortex A7 cores consume up to 2.5× the power of Cortex A53 cores. We hypothesized that these big differences in power consumption are due to the different manufacturing technologies (lithography) employed on both development boards. Nevertheless, to the best of our knowledge this information is not publicly available. In any case, 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. 10(a)).

To validate the results shown in Fig. 10, we measured power consumption using perf [18] and lm-sensors [19] on the Intel and the ARM systems, respectively. (Unlike PMCTrack, neither of these tools enables the monitoring of energy consumption on both platforms.) We observed similar measurements to those reported by PMCTrack (deviations no >1%).

4.2.2. *Reducing the EDP on asymmetric multicore systems*

As shown in Section 4.1, throughput on asymmetric single-ISA multicore systems can be maximized by mapping to big cores those applications in the workload with the highest big-to-small speedup (a.k.a., the *HSP* approach). However, Zhang *et al.* [49] have highlighted that this approach does not always result in a good performance-energy consumption tradeoff. To address this problem, they proposed PRIM, a rule-set-guided scheduling algorithm to reduce energy consumption on AMPs. At a high level, the proposed scheme works as follows. Initially, when a thread is created, it is mapped to a random core type in the system in order to preserve load balance. Every so often, the scheduler randomly selects a certain number of thread pairs consisting of a thread running on a big core (T_B) and another thread running on a small core (T_S). For each randomly-selected pair, the scheduler estimates whether swapping T_B with T_S would result in energy savings by means of a set of platform-specific rules; if that is the case, the two threads will be swapped. Evaluating these platform-specific rules at run time requires collecting threads' high-level performance metrics (such as the IPC or the LLCMR) by means of hardware performance counters. Thus, a platform-independent real-world implementation[6] of this scheduling algorithm in the Linux kernel could be created by relying on a PMCTrack monitoring module that encapsulates platform-specific rules to be evaluated at run time.

Nevertheless, PRIM platform-specific rules do not quantify the actual energy savings resulting from a thread swap, but instead indicate whether a specific thread swap would be beneficial or not in terms of energy consumption. Therefore, the PRIM scheduler cannot tell whether there is a better candidate thread running on a small core T_{S2} such that swapping T_B with T_{S2} would result in higher energy savings than those resulting from swapping T_B with T_S. This issue, coupled with the fact that thread pairs are selected randomly by PRIM, may cause this approach to perform suboptimal thread-to-core mappings.

[6] In the original work [49], the PRIM algorithm was simulated.

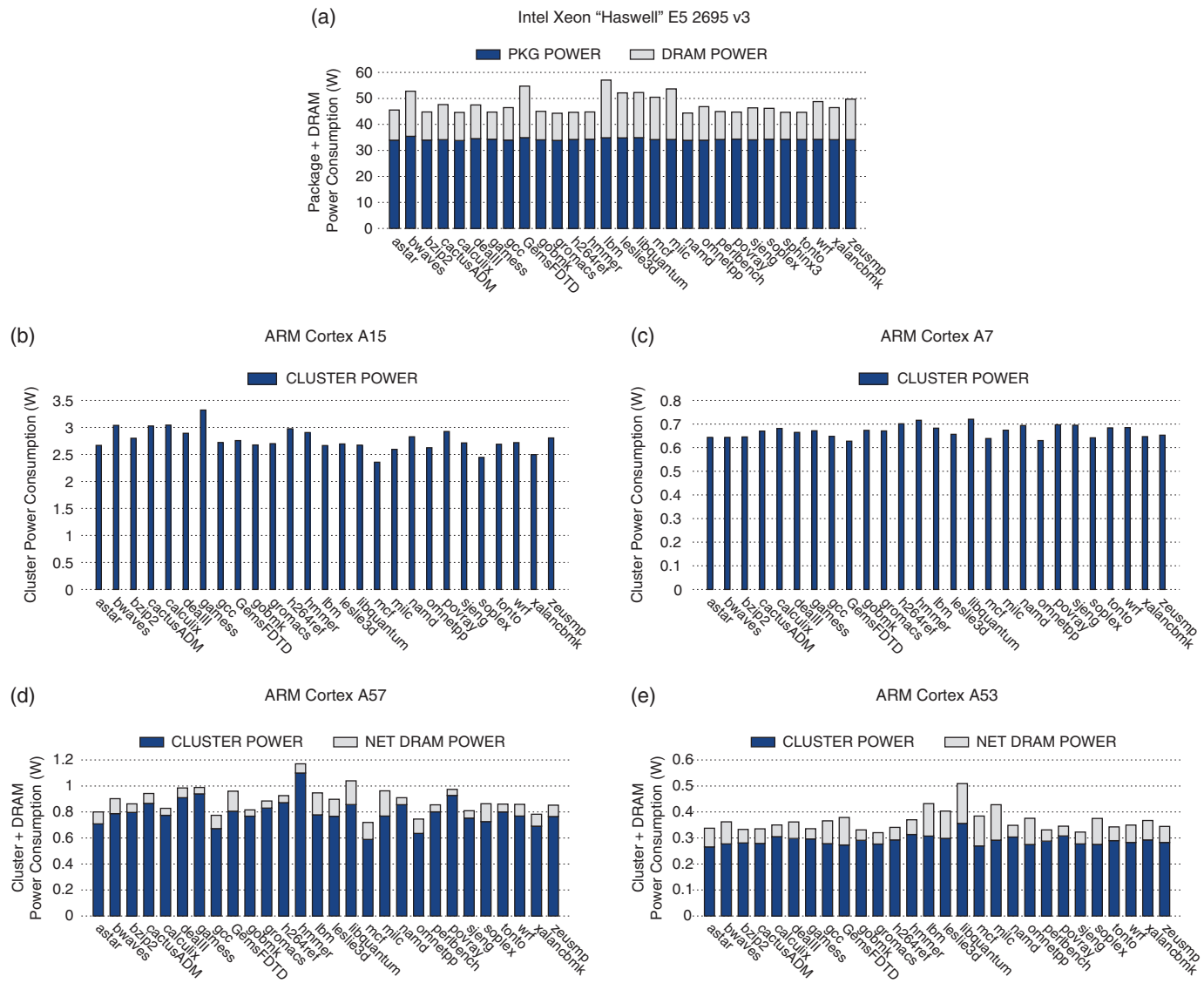

FIGURE 10. Average power consumption gathered for SPEC CPU benchmarks running on different processor models.

To overcome this limitation we propose guiding thread-to-core mappings on AMPs by taking into account an application's *efficiency factor* (EF). We define an application's EF as the ratio of the EDP observed when running this application alone on a small core relative to running it on a big core:

$$\mathrm{EF} = \frac{\mathrm{EDP}_{\mathrm{small}}}{\mathrm{EDP}_{\mathrm{big}}} \tag{3}$$

In turn, the EDP [50, 51] is defined as follows:

$$\mathrm{EDP} = \frac{\mathrm{Energy\_Consumed}}{\mathrm{IPS}} \tag{4}$$

$$= \frac{\mathrm{Energy\_Consumed}^{*}\mathrm{CompletionTime}}{\mathrm{Total\_retired\_instructions}} \tag{5}$$

where the IPS denotes the number of instructions per second.

Because the EDP is a lower-is-better metric, EF values >1 would indicate that mapping the application to a big core would lead to a better (lower) EDP value than mapping the application to a small core. In other words, the higher the EF, the more suitable an application is for being mapped to a big core in terms of energy efficiency. Unlike the rules in the PRIM scheme, the applications' EFs would enable the OS scheduler to quantify the effect on energy savings (EDP reduction) resulting from different threads swaps. Specifically, given a thread T_B mapped to a big core, the best

candidate thread T_S (mapped to a small core) to swap with T_B in terms of energy savings is the thread with the highest EF running on a small core (provided that $EF_{T_S} > EF_{T_B}$). This observation suggests that the scheduler could obtain a significant reduction in the EDP by mapping those applications in the workload with the highest EF to big cores, and relegating the remaining applications to small cores. Henceforth, we will refer to this scheduling scheme as *EF-Driven*.

Note that the EF can also be expressed in terms of the big-to-small speedup of a ST application, also referred to as the SF, and defined as the ratio of instructions per second achieved the application on big and small cores $\left(\frac{IPS_{big}}{IPS_{small}}\right)$. Thus, we can derive the following formula:

$$\begin{aligned} EF &= \frac{EDP_{small}}{EDP_{big}} \\ &= \frac{Energy\_Consumed_{small} * IPS_{big}}{Energy\_Consumed_{big} * IPS_{small}} \\ &= \frac{Energy\_Consumed_{small} * SF}{Energy\_Consumed_{big}} \end{aligned} \quad (6)$$

$$= \frac{SF}{Energy\_Ratio} \quad (7)$$

Equation (7) highlights that the EF metric combines information on both the energy consumption and performance of the application when running on a big core relative to a small core.

To evaluate the effectiveness of using the EF to perform thread-to-core mappings on an AMP system we experimented with the ARM Juno Development board, which features a 64-bit ARM big.LITTLE processor consisting of two big cores and four small cores. Further information about this system can be found in Table 4. Figure 11 shows the average SF and average EF for applications in the SPEC CPU2000 and CPU2006 benchmarks suites. As in [49], we opted to experiment with benchmarks from both suites, but we carried out our experiments on a real system rather than on a simulator. To obtain the average EF, we ran each benchmark on both core types and measured the total energy consumption in each case (core cluster plus DRAM consumption) with PMCTrack.

As is evident, every application experiences performance benefits from running on a big core relative to a small one on this platform (SF > 1 for all benchmarks). In addition, from the EDP standpoint, running an application on the big core yields to energy savings with respect to running it on a small core (as the EF is >1 across the board). We also observe a wide diversity of SFs and EFs across applications. As a result, when running multi-application workloads consisting of SPEC benchmarks on this platform, the thread-to-core assignments performed by an asymmetry-aware scheduler will have an important impact on throughput and energy consumption. Specifically, the HSP scheduler, which aims to optimize throughput (see Section 4.1), would perform very different thread-to-core assignments to those of an EF-driven scheduler. For example, for a hypothetical workload consisting of `art`, `soplex`, `hmmer` and `wupwise` running on the ARM big.LITTLE processor considered, the HSP scheduler would map the `hmmer` and `wupwise` applications to the big cores on the Juno board, and would relegate the other two applications to small cores; an EF-driven scheduler, by contrast, would perform the opposite mapping to reduce the EDP.

To assess the impact on throughput and energy consumption resulting from an EF-Driven scheduling policy, we built 40 multi-program workloads, consisting of six SPEC CPU applications each (one per core on the ARM Juno board) and ran them using two different static thread-to-core assignments: HSP and EF-Driven. In the HSP assignment, threads are mapped to cores in such a way as to maximize the system

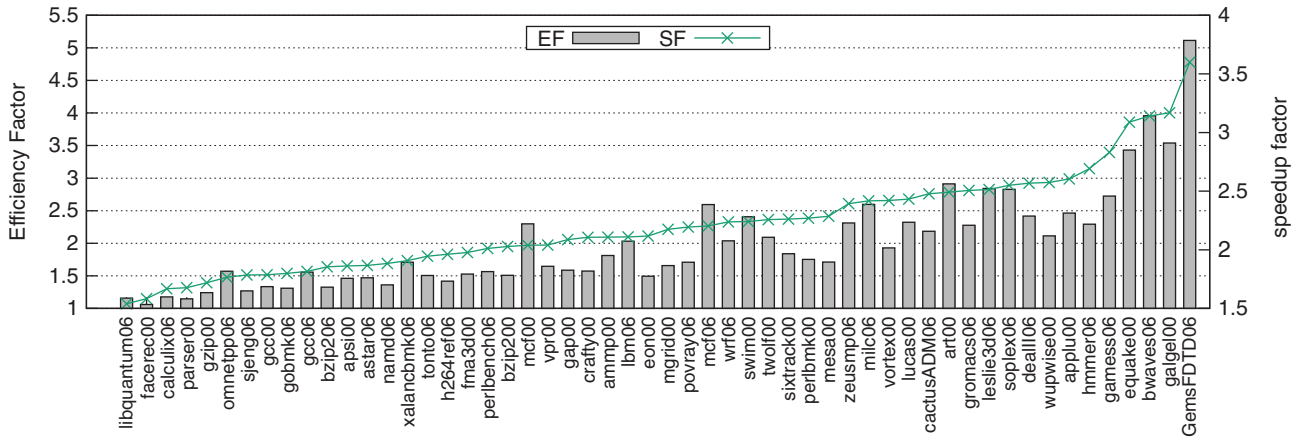

FIGURE 11. Average EF and average SF for SPEC CPU benchmarks observed when running on big and small cores of the ARM Juno development board.

throughput, so threads with the highest overall SFs in the workload are mapped to the big cores and the remaining applications are relegated to small ones. In the EF-Driven assignment, by contrast, big cores are dedicated to running threads with the highest overall EF in the workload. In selecting the workloads, we picked 19 SPEC CPU benchmarks with different EF and SF values. By using these benchmarks we created multiple program mixes consisting of six different applications. In turn, from these program mixes we selected 40 workloads such that the associated HSP and EF-Driven assignments led to different thread-to-core mappings. Since the applications in each workload have a different completion time, we launched the multi-program workloads using the same mechanism as that described in Section 4.1.3.

Figure 12 shows the reduction in the EDP and throughput resulting from the EF-Driven assignment relative to HSP for the multi-program workloads evaluated. To measure the EDP (as defined in Equation (5)) for each workload and thread-to-core mapping, we created a PMCTrack monitoring module that keeps track of the total number of retired instructions and the total energy consumption during the entire execution of each multi-program workload. To make this possible, the monitoring module accesses both the per-core performance counters and system-wide energy registers present in the system. To quantify the system throughput in each case, we use the ASP metric, as defined in Section 4.1.3.

The results reveal that driving thread-to-core assignments based on the EF yields significant energy savings in some cases (EDP reductions ranging from 20% to 60%). Note that in these scenarios throughput degradation over the throughput-optimized static assignment is no $>$7.6%. Despite the fact that the EF-Driven policy always leads to improved EDP over HSP, we observe that the throughput degradation, though small in most cases, largely depends on the diversity in energy ratios (i.e. SF/EF) among applications in the workload. Specifically, we found that workloads experiencing modest EDP reductions under EF-Driven match those including many applications with very similar energy ratios. In these scenarios, the HSP assignment already achieves an acceptable EDP value, thus providing a better throughput-energy tradeoff. This fact suggests that under these circumstances the SF should be taken into consideration by the scheduler when performing thread-to-core mappings. For the sake of the reproducibility of our results, Fig. 13 shows the normalized EDP and throughput associated with 10 selected workloads (listed in Table 5) from the full set. These workloads cover very different points in Fig. 12.

Clearly, static thread-to-core assignments, such as those considered thus far, may lead to suboptimal results in the event that applications go through diverse program-phases with different EF values over time. To examine this aspect, we analyzed the phased behavior of the SPEC CPU benchmarks concerning energy efficiency using PMCTrack. To this end, we obtained an application's EF over time by gathering the IPC and the energy consumption every 400 million instructions retired on both core types for the whole execution. (We observed that using this instruction window makes it possible

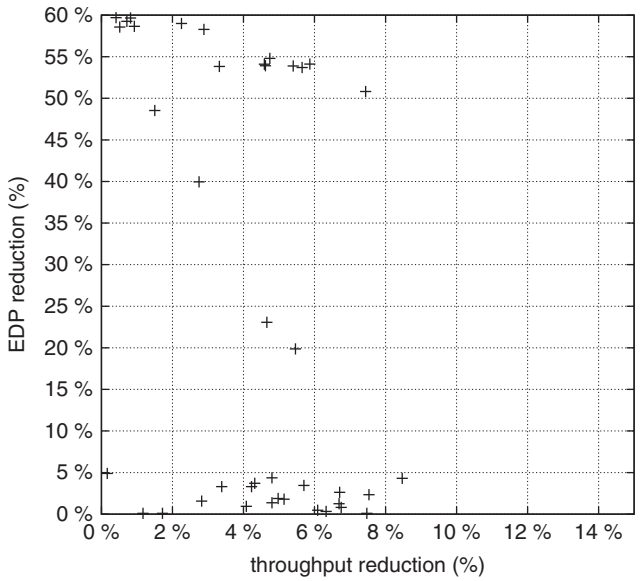

FIGURE 12. EDP and throughput reduction resulting from the EF-Driven assignment relative to HSP.

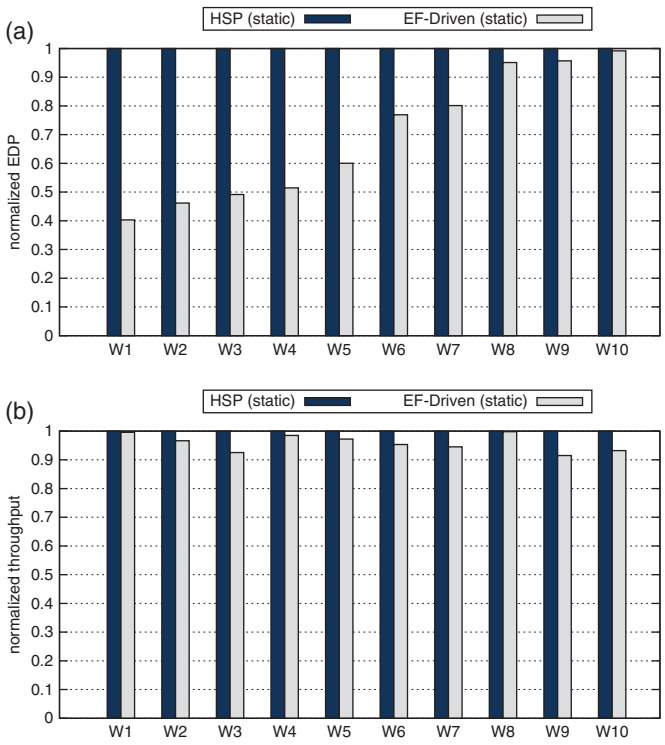

FIGURE 13. EDP (top) and throughput (bottom) normalized with respect to HSP.

TABLE 5. Multi-application workloads.

Workload	Applications
W1	`gamess,wupwise,soplex,bzip2,h264ref,gobmk`
W2	`hmmer,wupwise,soplex,vortex,h264ref,gobmk`
W3	`hmmer,wupwise,soplex,h264ref,gobmk,facerec`
W4	`gamess,wupwise,soplex,sixtrack,h264ref,gobmk`
W5	`swim,sixtrack,perlbmk,h264ref,gobmk,facerec`
W6	`gamess,swim,sixtrack,gap,bzip2,h264ref`
W7	`soplex,gamess,hmmer,vortex,sixtrack,h264ref`
W8	`art,gamess,wupwise,mesa,crafty,h264ref`
W9	`hmmer,mcf,gap,perlbench,bzip2,h264ref`
W10	`gamess,mcf,gap,crafty,perlbench,h264ref`

to effectively capture coarse-grained program phases and filter out many EF spikes.) Note that the EF of a particular instruction window can be obtained from the samples collected on both core types (see Equation (6)) for that instruction window. This analysis is possible thanks to PMCTrack's unique ability to read performance counters and energy registers in a fully coordinated fashion. Specifically, in this context both performance and energy values are read at the PMC overflow interrupt handler by using the EBS mode of PMCTrack.

Our analysis reveals that many SPEC applications actually exhibit distinct EF phases. As an illustrative example, Fig. 14 shows the EF over time for the `gap` and `soplex` benchmarks, which are used in the multi-program workloads considered. Because these benchmarks go through different EF phases, a dynamic scheduling policy with the ability to readjust thread-to-core mappings at run time in response to those changes is likely to obtain further reductions in the EDP compared to a static assignment. Nevertheless, to implement such a scheduling policy, the OS should be equipped with a mechanism to obtain a thread's EF over time. Designing such a mechanism is a challenging task. Note that direct measurement of the EF at run time (collecting EDP measures on both core types) is not possible because energy registers integrated in most platforms do not provide per-thread or per-application measurements but instead system-wide measurements. Thus, the OS scheduler cannot isolate the DRAM and core cluster energy consumption of individual applications in a multi-program workload.

A viable method to determine a thread's EF at run time would be by means of an estimation model relying on high-level performance metrics collected online using performance counters sampled on the current core type. Note that on systems where cores have identical microarchitectures (clearly not the case here), several researchers [5, 13, 14, 52] have used performance counters to estimate energy and power consumption. Because these models are largely processor specific, two separate estimation models should be derived for a big.LITTLE-based asymmetric multicore system: one to predict the EF when a thread runs on the big core and another to predict the EF on a small core. To derive such estimation

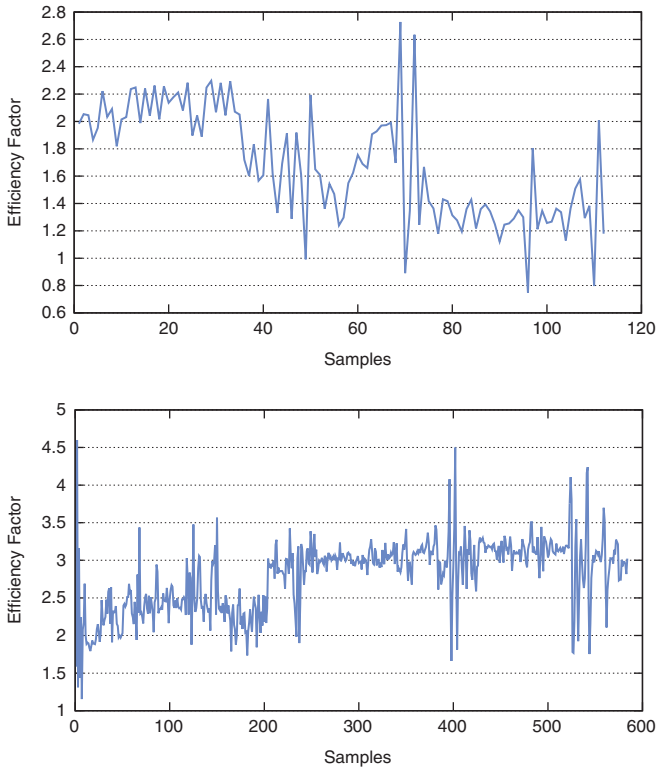

FIGURE 14. EF over time for the `gap` (top) and `soplex` (bottom) applications.

models we used a similar approach to the one used to estimate the SF on the Intel QuickIA platform (described in Section 4.1.2). Specifically, using the EBS mode of PMCTrack we collected a wide range of high-level performance metrics on both core types for all benchmarks in the SPEC CPU2006 and CPU2000 suites. In addition, from the information collected we identified coarse EF phases in the various benchmarks. To obtain an estimation model for each core type, we employed additive regression [53] using the performance and energy data collected.

Figure 15 shows the EFs predicted by the derived estimation models on both core types of the ARM Juno development board. In generating the models, we used performance and energy data from 900 EF-phases from the SPEC CPU benchmarks. Note that to improve the robustness of the models, we employed *cross-validation*; thus, each model was repeatedly generated using part of the program phase data for training and the other part for testing. The correlation coefficients for the estimation on the big and the small cores are 0.96 and 0.95, respectively. Note that the additive-regression prediction engine we used [44] aids in identifying irrelevant performance metrics for the model (low regression coefficients) and so some metrics could be discarded from the final estimation models, thus reducing the complexity. Table 6 enumerates the set of performance metrics and associated hardware events that the models depend upon.

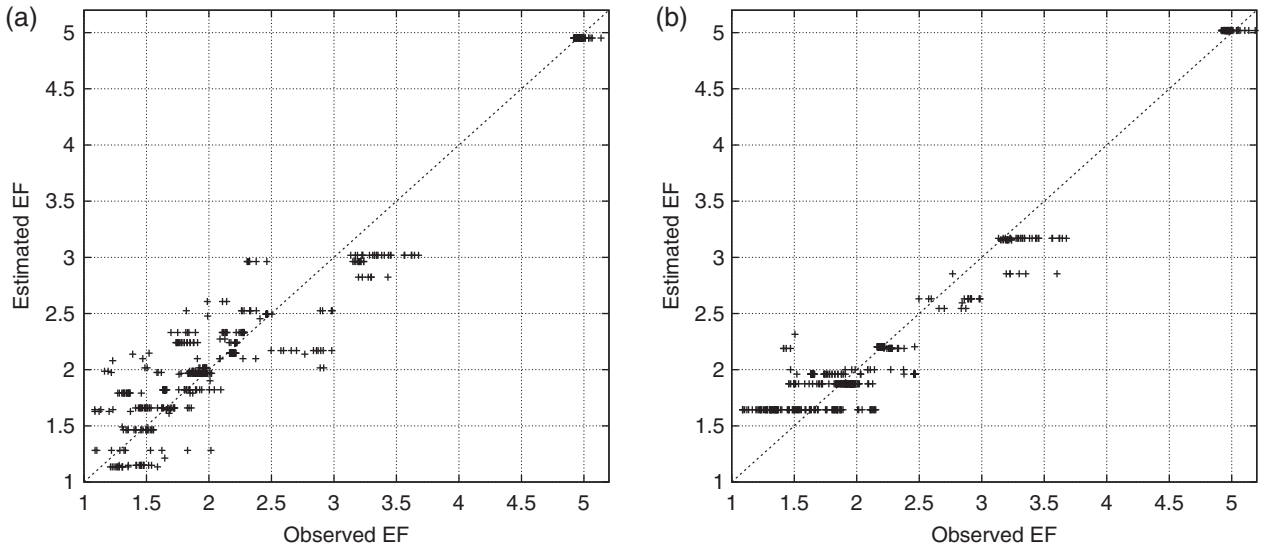

FIGURE 15. EF prediction on the big (left) and the small (right) core on the ARM Juno development board. Perfectly accurate estimations have all points on the diagonal line.

TABLE 6. Performance metrics and associated hardware events used to predict the EF on the ARM Juno development board.

Core	Hardware Events	Performance metrics
Big	Instructions retired, processor cycles, L1 instruction TLB misses Mispredicted branches speculatively executed L2 Data cache accesses, L2 Data cache misses, Data memory access	Instructions per cycle, L1 instruction TLB misses per 1M instr., Mispredicted branches per 1K instr., L2 (last-level) cache accesses per 1K instr., L2 (last-level) cache misses per 1K instr., Data memory accesses per 1K retired instr.
Small	Instructions retired, processor cycles, L1 data cache misses, Mispredicted branches speculatively executed, L2 Data cache misses, L2 Data cache accesses, Conditional branches executed, `STALLS_1`: Data Write operations that stall the pipeline because the store buffer is full, `STALLS_2`: Counts every cycle there is an interlock that is not because of an Advanced SIMD or Floating-point instruction, and not because of a load/store instruction waiting for data to calculate the address in the AGU	Instructions per cycle, L1 data cache misses per 1K instr., Mispredicted branches per 1K instr., L2 (last-level) cache accesses per 1K instr., L2 (last-level) data cache misses per 1K instr. Conditional branches executed per 1K instr. `STALLS_1` per 1K instr., `STALLS_2` per 1K cycles

We have implemented both estimation models by means of a PMCTrack monitoring module. Because the number of hardware events the models depend upon exceeds the number of hardware counters available on big and small cores (six per-core general-purpose PMCs) of this asymmetric multicore system, PMCTrack's implementation needs to use event multiplexing to estimate the EF online. Because the same phase-related issues described in Section 4.1.2 become apparent in this context as well, we follow the same approach as that used when estimating the SF on the QuickIA. Currently, we are porting our kernel-level asymmetry-aware scheduling framework (which consists of 25K lines of code, in Linux v3.2) to a version of the Linux kernel supported by the ARM Juno board (Linux linaro v3.10 and higher). Therefore, we leave for future work the Linux kernel implementation of the dynamic EF-Driven and PRIM algorithms, as well as their evaluation on a real asymmetric system.

4.3. Cache monitoring

Intel's CMT [20, 54] is a new feature introduced in the Intel Xeon E5 2600 v3 product family. This feature allows an

operating system or a Hypervisor/Virtual Machine Monitor (VMM) to determine the current 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 to each application/VM; this ID is referred to as the Resource Monitoring ID (RMID). 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 it possible to track LLC occupancy on a per-application basis, 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 to store the RMID associated with the thread currently running on it. The OS is in charge of updating per-core RMID registers when context switches take place.

Because the number of RMIDs available is limited by the hardware implementation (56 on our experimental platform), the OS must be equipped with a carefully crafted RMID allocation policy. Specifically, the OS must avoid assigning any RMID freed up by a recently terminated application to a new application, since stale cache lines belonging to the former could still remain in the LLC and therefore be accounted to the occupancy of the incoming application. We implemented the necessary OS support for Intel CMT using a PMCTrack monitoring module. In our implementation, we explored three different RMID-allocation policies: FIFO, LIFO and random. Clearly, the FIFO policy constitutes the best choice; the other two (especially LIFO) are subject to the aforementioned issue.

The perf events subsystem has been recently augmented to support Intel-CMT. Notably, the PMCTrack's implementation exhibits some important advantages over perf's implementation. First, it uses <500 lines of code against the 1500 lines used by the perf patch for the Linux kernel [55]. More importantly, perf event's implementation entails making changes to seven 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. Second, our implementation is in a kernel loadable module rather than in the kernel itself, which greatly simplified the development process. 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 kernel API) and perform effective thread-to-core mappings [54]. At the same time, the monitoring module exposes the LLC usage as a virtual counter, so the user can also collect LLC utilization via PMCTrack's userland tools.

To validate PMCTrack's support for Intel-CMT, we collected the LLC occupancy of several multiprogram workloads using both PMCTrack and perf. In order to do so, we used a very recent version of the Linux kernel (v4.1.5) equipped with the Intel-CMT patch for perf.[7] In our validation analysis, we built several multiprogram workloads consisting of parallel and sequential programs from the SPEC CPU2006, SPEC OMP and PARSEC suites. Table 7 shows the 12 multiprogrammed workloads used for the analysis. For multithreaded applications, the number in parentheses by each program's name represents the number of threads it runs with. To carry out the experiments, we employed a 14-core 'Haswell-EP' Xeon E5–2695 v3 processor operating at 2.3 GHz, and featuring a 35 MB last-level (L3) cache.

TABLE 7. Multi-application workloads.

Workload	Benchmarks
mix1	`ilbdc`(4), `swim`(4)
mix2	`applu331`(4), `swim`(4)
mix3	`applu331`(4), `ilbdc`(4)
mix4	`raytrace`(4), `streamcluster`(4)
mix5	`raytrace`(4), `x264`(4)
mix6	`streamcluster`(4), `x264`(4)
mix7	`swim`(4), `x264`(4)
mix8	`applu331`(4), `streamcluster`(4)
mix9	`ilbdc`(4), `streamcluster`(4)
mix10	`swim`(4), `lbm`, `mcf`
mix11	`ilbdc`(4), `lbm`, `mcf`
mix12	`applu331`(4), `lbm`, `mcf`

Figure 16 shows the per-application average LLC occupancy for the various workloads. For each mix we illustrate the results reported using both PMCTrack's implementation and perf's implementation. Clearly, the figure reveals that both tools report almost exactly the same LLC occupancy values across the board. As is evident, some applications, such as the *swim* SPEC OMP parallel program, use a great portion of the LLC (65–95%) when running concurrently with sequential applications or with another multithreaded SPEC OMP or PARSEC programs. Conversely, other parallel programs, such as *ilbdc* from the SPEC OMP suite, typically occupy a much smaller portion of the L3 cache regardless of the co-runner application.

We should also highlight that the TBS feature of PMCTrack makes it possible to monitor the cache occupancy over time for the various applications in the workload. Figure 17 shows the per-application LLC usage for a workload featuring two SPEC CPU2006 (sequential) applications and a multithreaded program from the SPEC OMP suite. As is evident, varying cache distributions can be observed as applications go through different program phases.

4.3.1. *MRCs online generation*

As an interesting application of PMCTrack and CMT, we now introduce a technique to generate MRCs online. The

[7] This support was first introduced in the Linux kernel v4.1.

MRC reports an application's cache occupancy on a given cache level (usually the LLC) vs. a certain related performance metric, such as 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 [56,57] or to adapt the cache size to reduce energy consumption [58]. Several mechanisms have been proposed for building these curves [56,57,59,60], but they all pose different limitations, such as requiring hardware support or relying on code instrumentation.

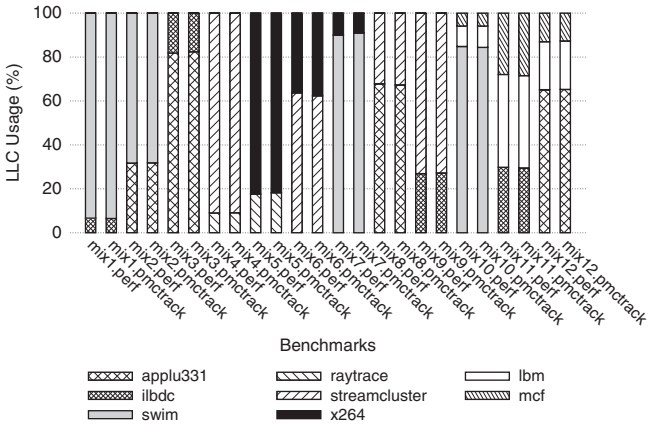

FIGURE 16. Breakdown of LLC occupancy for different mixes.

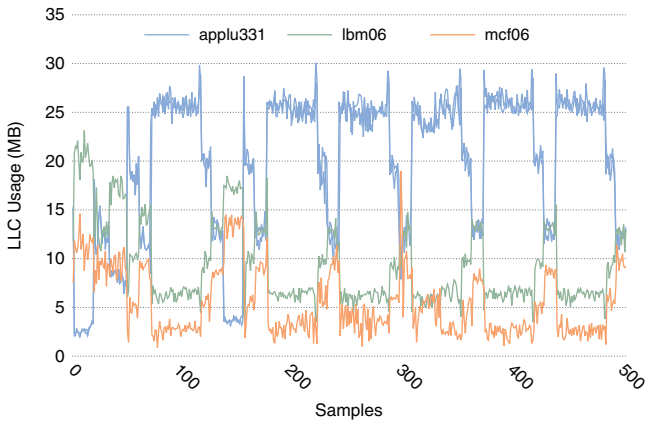

FIGURE 17. LLC occupancy along time for each program within a given workload.

As a case study of PMCTrack, we propose an online technique that leverages Intel's CMT to generate the MRCs of co-running applications. It is inspired by a previous mechanism introduced in [61]. Overall, the technique works as follows. By using PMCTrack we periodically gather the MPKI and the LLC occupancy of the co-running applications, 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 18 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 it saturates from a certain cache size point on. The MRC for the *omnetpp* program, in contrast, shows a linear MPKI drop for the whole range of cache sizes.

As future work we plan to evaluate our MRC generation mechanism in the context of asymmetric multicore systems. In this scenario, perturbations of the equilibrium state occur naturally; when a thread is mapped to a big core it usually makes faster progress and increases its cache access rate. This leads to increased pressure on the LLC, thus affecting the LLC occupancy. Furthermore, to aid in generating MRCs, we plan to use another feature recently introduced on Intel processors and related to CMT, namely Cache Allocation Technology [21]. This feature allows the OS to dynamically specify the amount of cache space into which an application can fill. The system software could periodically vary the

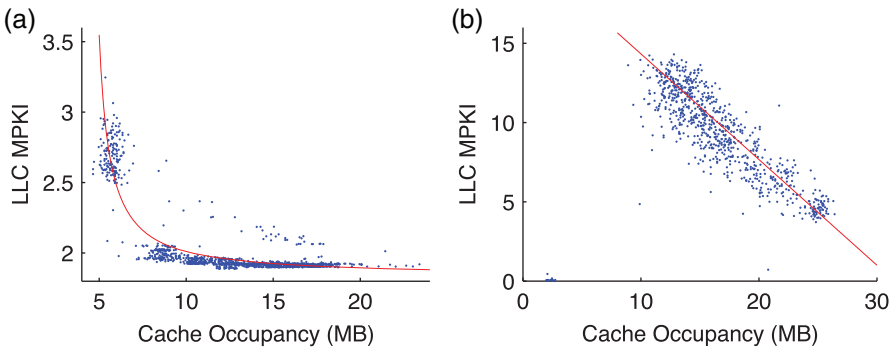

FIGURE 18. MRCs for *lbm* (a) and *omnetpp* (b) applications.

per-application maximum LLC allocation to monitor the MPKI for different cache size configurations. This would allow us to obtain MRC points over the whole cache size range for the co-running applications.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed PMCTrack, a tool that enables the OS scheduler to leverage 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 they 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 designed specifically to aid the OS kernel in runtime optimizations, PMCTrack is also equipped with a set of userland tools that enable the gathering of hardware performance monitoring information from userspace in various ways. These tools may also assist kernel developers during the scheduler's development process (as demonstrated in Section 4.1.2) and provide valuable information to researchers when it comes to analyzing the potential benefits of novel scheduling policies (as shown in Section 4.2.2).

We have illustrated the effectiveness and flexibility of PMCTrack on a wide range of processor models and architectures by means of three case studies. The first case study showcases the potential of PMCTrack's monitoring modules in assisting scheduling algorithms for asymmetric single-ISA multicore systems (AMPs) implemented in the Linux kernel. Notably, PMCTrack's monitoring modules enabled us to extensively evaluate (on real asymmetric hardware) PMC-based asymmetry-aware schedulers that were evaluated before using simulators [17] or using emulated asymmetric hardware [10]. More importantly, the implementation of these modules heavily relies on PMCTrack's in-kernel explicit event-multiplexing capabilities, which are not available in userspace-oriented PMC tools, such as perf [18]. This capability is also leveraged in the second case study (EF estimation), which focuses on power and energy consumption measurement on high-performance multicore systems and embedded boards featuring ARM big.LITTLE processors. Finally, the third case study showcases our proposed scheme for building application MRCs on a real system by leveraging the cache-usage monitoring support available in modern Intel processors [20].

PMCTrack's source code has been released [62] under GPLv2. Additional information on PMCTrack can be found on PMCTrack's official website [28]. We are currently working on a port of PMCTrack for the Android OS, and have plans to augment the tool with support for additional CPU architectures. As for future work, we plan to extensively evaluate resource-contention-aware and energy-cognizant OS scheduling algorithms that leverage PMCTrack information on cache occupancy or power consumption.

ACKNOWLEDGEMENTS

We would like to thank David Koufaty (Circuits and Systems Research Lab at Intel) and Alexandra Fedorova (Simon Fraser University) for enabling us to experiment with the QuickIA prototype system.

FUNDING

EU (FEDER) and the Spanish MINECO (grants TIN2012-32180, TIN 2015-65277-R); HIPEAC-4 European Network of Excellence; University of Costa Rica and the Costa Rican Ministry of Science and Technology MICIT and CONICIT.

REFERENCES

[1] Weaver, V. (2013) Linux Perfevents Features and Overhead. In *Proceeding of Int. Workshop on Performance Analysis of Workload Optimized Systems*, Austin, TX, 21 April, pp. 80–80. IEEE Computer Society Press, Los Alamitos, CA.

[2] Jarp, S., Jurga, R. and Nowak, A. (2008) Perfmon2: a leap forward in performance monitoring. *J. Phys.: Conf. Ser.*, **119**, 042017.

[3] Cohen, W. (2004) Tuning programs with oprofile. *Wide Open Mag.*, **1**, 53–62.

[4] Knauerhase, R., Brett, P., Hohlt, B., Li, T. and Hahn, S. (2008) Using OS observations to improve performance in multicore systems. *IEEE Micro*, **28**, 54–66.

[5] Spiliopoulos, V., Kaxiras, S. and Keramidas, G. (2011) Green Governors: A Framework for Continuously Adaptive DVFS. In *Proc. IGCC 11*, Orlando, FL, July 25–28, pp. 1–8. IEEE Computer Society Press, Los Alamitos, CA.

[6] Zhuravlev, S., Blagodurov, S. and Fedorova, A. (2010) Addressing Cache Contention in Multicore Processors Via Scheduling. In *Proc. ASPLOS 10*, Pittsbourgh, PA, March 13–17, pp. 129–142. ACM, New York.

[7] Merkel, A., Stoess, J. and Bellosa, F. (2010) Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proc. EuroSys 10*, Paris, France, April 13–16, pp. 153–166. ACM, New York.

[8] Zhuravlev, S., Blagodurov, S. and Fedorova, A. (2010) Akula: A Toolset for Experimenting and Developing Thread Placement Algorithms on Multicore Systems. In *Proc. PACT 10*, Vienna, Austria, September 11–15, pp. 249–260. ACM, New York.

[9] Koufaty, D., Reddy, D. and Hahn, S. (2010) Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proc.*

Eurosys 10, Paris, France, April 13–16, pp. 125–138. ACM, New York.

[10] Saez, J.C., Fedorova, A., Koufaty, D. and Prieto, M. (2012) Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Trans. Comput. Syst.*, **30**, 6:1–6:38.

[11] Petrucci, V., Loques, O., Mossé, D., Melhem, R., Gazala, N. A. and Gobriel, S. (2015) Energy-efficient thread assignment optimization for heterogeneous multicore systems. *ACM Trans. Embed. Comput. Syst.*, **14**, 15:1–15:26.

[12] Saez, J.C., Pousa, A., Castro, F., Chaver, D. and Prieto-Matias, M. (2015) ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems. In *Proc. ACM SAC 15*, Salamanca, Spain, April 13–17, pp. 2027–2032. ACM, New York.

[13] Ghiasi, S., Keller, T., and Rawson, F. (2005) Scheduling for Heterogeneous Processors in Server Systems. In *Proc. Comput. Frontiers 05*, Como, Italy, May 16–18, pp. 199–210. ACM, New York.

[14] Singh, K., Bhadauria, M. and McKee, S.A. (2009) Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, **37**, 46–55.

[15] Saez, J.C., Prieto, M., Fedorova, A., and Blagodurov, S. (2010) A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proc. Eurosys 10*, Paris, France, April 13–16, pp. 139–152. ACM, New York.

[16] Li, T., Brett, P., Knauerhase, R. and Koufaty, D. (2010) Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. In *Proc. HPCA 10*, Bangalore, India, January 9–14, pp. 1–12. IEEE Computer Society Press, Los Alamitos, CA.

[17] Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A. and Eeckhout, L. (2013) Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. In *Proc. PACT 13*, Edinburgh, Scotland, September 7–11, pp. 177–187. ACM, New York.

[18] Perf (2015) Perf wiki tutorial on perf. https://perf.wiki.kernel.org/index.php (accessed January 20, 2015).

[19] Lm-sensors (2015) Hardware monitoring by lm-sensors. http://www.lm-sensors.org/ (accessed March 02, 2015).

[20] Nguyen, K. (2014) Intel's cache monitoring technology software-visible interfaces. https://software.intel.com/en-us/blogs/2014/12/11/intel-s-cache-monitoring-technology-software-visible-interfaces (accessed February 10, 2015).

[21] Intel. Intel ® 64 and IA-32 architectures software developer's manual volumes 3A and 3B: system programming guide. http://www.intel.com/products/processor/manuals (accessed January 15, 2015).

[22] ARM. ARM architecture reference manual. armv7-a and armv7-r edition. http://infocenter.arm.com/ (accessed January 15, 2015).

[23] Browne, S., Dongarra, J., Garner, N., Ho, G. and Mucci, P. (2000) A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, **14**, 189–204.

[24] Treibig, J., Hager, G., and Wellein, G. (2010) LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proc. ICPPW 10*, San Diego, CA, September 13–16, pp. 207–216. IEEE Computer Society, Washington, DC, USA.

[25] Tanica, L., Ilic, A., Tomas, P. and Sousa, L. (2014) Schedmon: A Performance and Energy Monitoring Tool for Modern Multi-Cores. In *Proc. Euro-Par 14: Paral. Process. Workshops*, Porto, Portugal, August 25–26, pp. 230–241. Springer, Berlin.

[26] ARM (2014). CoreTile express development board. http://www.arm.com/products/tools/development-boards/versatile-express/coretile-express.php (accessed March 02, 2015).

[27] ARM (2014) ARM Juno development board. http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php (accessed November 16, 2015).

[28] PMCTrack. Project official website. http://pmctrack.dacya.ucm.es/.

[29] PAPI. PAPI-C overview. http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview (accessed January 30, 2015).

[30] Chen, Q. and Guo, M. (2014) Adaptive workload-aware task scheduling for single-ISA asymmetric multicore architectures. *ACM Trans. Archit. Code Optim.*, **11**, 8:1–8:25.

[31] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., and Farkas, K.I. (2004) Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. ISCA 04*, Munich, Germany, June 19–23, pp. 64–75. IEEE Computer Society, Washington, DC, USA.

[32] ARM. Benefits of the big.LITTLE Architecture. http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf (accessed January 10, 2015).

[33] Chitlur, N. *et al* (2012) QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *Proc. HPCA 12*, New Orleans, LA, February 25–29, pp. 1–8. IEEE Computer Society, Washington, DC, USA.

[34] Shelepov, D., Saez, J.C., Jeffery, S., Fedorova, A., Perez, N., Huang, Z.F., Blagodurov, S. and Kumar, V. (2009) HASS: a scheduler for heterogeneous multicore systems. *ACM Oper. Syst. Rev.*, **43**, 66–75.

[35] Becchi, M. and Crowley, P. (2006) Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. CF 06*, Ischia, Italy, May 2–5, pp. 29–40. ACM, New York.

[36] Annavaram, M., Grochowski, E. and Shen, J. (2005) Mitigating Amdahl's Law through EPI Throttling. In *Proc. ISCA 05*, Wisconsin, USA, June 4–8, pp. 298–309. IEEE Computer Society, Washington, DC, USA.

[37] Hill, M.D. and Marty, M.R. (2008) Amdahl's Law in the Multicore Era. *IEEE Comput.*, **41**, 33–38.

[38] Saez, J.C., Pousa, A., Castro, F., Chaver, D. and Prieto-Matias, M. (2014) Exploring the throughput-fairness trade-off on asymmetric multicore systems. In *Proc. Euro-Par 14: Parall. Process. Workshops*, Porto, Portugal, August 25–26, pp. 326–337. Springer, Berlin.

[39] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P. and Emer, J. (2012) Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE). In *Proc. ISCA 12*, Portland, OR, June 9–13, pp. 213–224. IEEE Computer Society Washington, DC, USA.

[40] Pricopi, M., Muthukaruppan, T.S., Venkataramani, V., Mitra, T. and Vishin, S. (2013) Power-Performance Modeling on Asymmetric Multi-Cores. In *Proc. CASES 13*, Montreal, Canada, September 29–October 4, pp. 15:1–15:10. IEEE Press Piscataway, NJ, USA.

[41] Saez, J.C., Shelepov, D., Fedorova, A. and Prieto, M. (2011) Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, **71**, 114–131.

[42] Joao, J.A., Suleman, M.A., Mutlu, O., and Patt, Y.N. (2013) Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. In *Proc. ISCA 13*, Tel-Aviv, Israel, June 23–27, pp. 154–165. ACM, New York.

[43] Petrucci, V., Loques, O. and Mossé, D. (2012) Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems. In *Proc. USENIX HotPower 12*, Hollywood, CA, October 7, pp. 7–7. USENIX Association Berkeley, CA, USA.

[44] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I.H. (2009) The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, **11**, 10–18.

[45] Van der Pas, R. (2005) The OMPlab on Sun Systems. In *Proc. IWOMP'05*, Eugene, OR, (accessed 1–4 June).

[46] Gabor, R., Weiss, S. and Mendelson, A. (2006) Fairness and Throughput in Switch on Event Multithreading. In *Proc. MICRO 06*, Orlando, FL, December 9–13, pp. 149–160. IEEE Computer Society Washington, DC, USA.

[47] Mutlu, O. and Moscibroda, T. (2007) Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proc. MICRO '07*, Chicago, IL, December 1–5, pp. 146–160. IEEE Computer Society Washington, DC, USA.

[48] Ebrahimi, E., Lee, C.J., Mutlu, O. and Patt, Y.N. (2010) Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Proc. ASPLOS 10*, Pittsburgh, PA, March 13–17, pp. 335–346. ACM, New York.

[49] Zhang, Y., Duan, L., Li, B., Peng, L. and Sadagopan, S. (2015) Cross-architecture prediction based scheduling for energy efficient execution on single-ISA heterogeneous chip-multiprocessors. *Microprocess. Microsyst.*, **39**, 271–285.

[50] Horowitz, M., Indermaur, T. and Gonzalez, R. (1994) Low-Power Digital Design. In *Proc. IEEE Symposium on Low Power Electronics*, San Diego, CA, October 10–12, pp. 8–11.

[51] Gonzalez, R. and Horowitz, M. (1996) Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuit.*, **31**, 1277–1284.

[52] Isci, C. and Martonosi, M. (2003) Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proc. MICRO 03*, San Diego, CA, December 3–5, pp. 93–104. IEEE Computer Society Washington, DC, USA.

[53] Friedman, J.H. (2002) Stochastic gradient boosting. *Comput. Stat. Data Anal.*, **38**, 367–378.

[54] Nguyen, K. (2014) Benefits of Intel(R) cache monitoring technology in the Intel(R) Xeon(TM) processor E5 v3 family. https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring (accessed February 10, 2015).

[55] Flemming, M. (2014) perf: Intel cache QoS monitoring support. https://lkml.org/lkml/2015/1/23/590 (accessed February 05, 2015).

[56] Qureshi, M.K. and Patt, Y.N. (2006) Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. MICRO 06*, Orlando, FL, December 9–13, pp. 423–432. IEEE Computer Society Washington, DC, USA.

[57] Tam, D.K., Azimi, R., Soares, L.B., and Stumm, M. (2009) RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *Proc. ASPLOS 09*, Washington, DC, March 7–11, pp. 121–132. ACM, New York.

[58] Sen, R. and Wood, D.A. (2013) Reuse-Based Online Models for Caches. In *Proc. SIGMETRICS 13*, Pittsburgh, PA, June 17–21, pp. 279–292. ACM, New York.

[59] Berg, E. and Hagersten, E. (2004) Statcache: A Probabilistic Approach to Efficient and Accurate Data Locality Anpalysis. In *Proc. ISPASS 04*, Austin, TX, March 10–12, pp. 20–27. IEEE Computer Society, Washington, DC, USA.

[60] Guo, F. and Solihin, Y. (2006) An Analytical Model for Cache Replacement Policy Performance. In *Proc. SIGMETRICS 06*, Saint Malo, France, June 26–30, pp. 228–239. ACM, New York.

[61] West, R., Zaroo, P., Waldspurger, C.A. and Zhang, X. (2010) Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev.*, **44**, 19–29.

[62] PMCTrack (2015) Source code repository at Github. https://github.com/jcsaezal/pmctrack (accessed December 10, 2015).