Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems

Carlos Bilbao[1 [0000−0002−4750−5124]], Juan Carlos Saez[1 [0000−0003−1343−7108]], and Manuel Prieto-Matias[1 [0000−0003−0687−3737]]

Facultad de Informática, Universidad Complutense de Madrid, Spain
{cbilbao,jcsaezal,mpmatias}@ucm.es

Abstract. Asymmetric multicore processors (AMPs) couple high-performance big cores and power-efficient small ones, all exposing a shared instruction set architecture to software, but with different microarchitectural features. The energy efficiency benefits of AMPs together with the general-purpose nature of the various cores, have led hardware manufactures to build commercial AMP-based products, first for the mobile and embedded domains, and more recently for the desktop market segment, as with the Intel Alder Lake processor family. This indicates that AMPs may become a solid and more energy efficient replacement to symmetric multicores in a wide range of application domains.
Previous research has demonstrated that the system software can substantially improve scheduling –critical to get the most out of heterogeneous cores– by leveraging hardware facilities that are directly managed by the the OS, such as performance monitoring counters, or the recently introduced Intel Thread Director technology. Unfortunately, the OS-level support enabling to access scheduling-relevant hardware support may take a long time to be adopted in operating systems, or may come in forms that make its utilization challenging from specific levels of the system software stack, especially in production systems. To fill this gap, we propose the PMCSched framework, which enables the creation of custom OS support on Linux to aid in the design of novel scheduling and resource-management policies for multicores implemented at different layers of the system software, but without requiring to patch the kernel. To demonstrate the potential of our framework, we implement a set of OS-level schedulers for AMPs, that make use of custom OS extensions to access scheduling-relevant hardware facilities in an x86 AMP processor.

Keywords: Asymmetric multicore processors · Scheduling · Operating Systems · Runtime Systems · Linux kernel · Intel Alder Lake

1 Introduction

Energy efficiency has become one of the most critical constraints of processor design [14]. The quest for improved energy efficiency substantially contributed

to the proliferation of heterogeneous architectures that combine within the same platform different types of cores and processing units for diverse and specialized uses [10]. Asymmetric multicore processors (AMPs) constitute an attractive type of heterogeneous architecture where high-performance big cores and power-efficient small ones –all exposing a shared ISA (instruction set architecture)– are combined on the same system. The common ISA in conjunction with the general-purpose nature of the AMP cores, allows the execution of legacy (unmodified) software. These facts, along with AMPs' energy efficiency benefits, have drawn the attention of major hardware players, leading to the massive release of commercial AMP products for mobile platforms, such as those based on the ARM big.LITTLE processor [28]. Today, the Intel Alder Lake processor family and the Apple M1 SoC, are clear examples of the expansion of AMPs toward the desktop market segment [31]. Moreover, in the high performance computing (HPC) domain, the combination of different core types with a shared ISA has also been explored; the Sunway TaihuLight supercomputer is a representative case [6,8].

Despite the remarkable benefits of AMPs [19], effectively scheduling diverse programs/tasks on heterogeneous cores poses a significant challenge to the various system software layers [21,25,10,5,6]. When a single multithreaded application runs alone on an AMP system, smart user-level scheduling within the runtime system is the key to making the most out of its heterogeneous cores [6,30]. However, in multi-application scenarios, and especially under the presence of legacy programs, the OS scheduler plays an essential role in transparently delivering the benefits of AMPs to the end user [18,25,10,16].

Previous research has demonstrated that the runtime system and the OS scheduler can perform optimizations on AMPs by leveraging hardware features that are directly controlled by the OS kernel and exposed to user space, such as Performance Monitoring Counters (PMCs) [10,36,18] or Dynamic Voltage and Frequency Scaling (DVFS) [35,7]. Often, the support to conveniently access new scheduling-relevant hardware features from the system software may take time to be adopted in operating systems [31], or it may come in the form of architecture-specific interfaces that limit application portability or make its utilization impossible from particular levels of the software stack [11,17]. Take for instance the Linux kernel, that does not currently feature support for the Intel Thread Director (TD) technology [31], unlike the proprietary Windows 11 kernel. TD is a set of scheduling-related hardware facilities –introduced with Alder Lake processors– that provide the system software with performance and energy efficiency hints to aid in carrying out effective thread-to-core mappings on Intel AMPs. Implementing custom mechanisms in the OS kernel to leverage these new –yet unsupported– features directly from the OS scheduler, or exposing them to user space involves a substantial development effort, due to the inherent challenges associated with kernel-level programming [26,11]. At the same time, custom OS-level extensions could be difficult to be adopted in production systems, where patching the OS kernel may be impractical.

To address these issues, we propose PMCSched, a framework for the Linux kernel that enables rapid development of the OS-level support required to create

custom scheduling and resource-management schemes on both symmetric and asymmetric multicore systems. Unlike other existing frameworks that require patching the Linux kernel to function [4,20,26,39], PMCSched makes it possible to incorporate new scheduling-related OS-level support in Linux via a kernel module that can be loaded in unmodified kernels, making its adoption easier in production systems. Notably, the main focus of this framework is to simplify the creation of novel scheduling and resource-management strategies that are either implemented entirely in the OS kernel, or require changes in different layers of the system software, so as to benefit from coordinated decisions between the runtime system and the OS scheduler [13,10,30].

As a proof of concept of our framework, in this work we implement different asymmetry-aware OS-level schedulers on top of an unmodified Linux kernel v5.16, and evaluate their effectiveness by running different multi-application workloads on an Intel Alder Lake processor. These schedulers make use of PMCs and leverage the Intel Thread Director technology [15,16], by accessing such hardware facilities directly from kernel space.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of PMCSched design and introduces its main implementation challenges. Section 4 covers the experimental case study on scheduling for Alder Lake processors, and Section 5 concludes the paper.

2 Related Work

A large body of work has proposed asymmetry-aware scheduling strategies for adoption on either runtime systems [5,37,25] or OS kernels [18,10,31]. Frequently, such endeavors culminate in tools and frameworks that aim to ease the development and analysis of new scheduling algorithms; these are likewise some of the main goals of this paper.

Recent studies have shown that scheduling algorithms that come in stock general-purpose OSs exhibit suboptimal behavior for different workloads on a wide range of processor architectures [10,23,6]. At the same time, making the required changes in an OS kernel to build effective scheduling policies specifically tailored to custom workloads or microarchitectures may be a significant burden to the average developer [26,39]. On many monolithic kernels such as Linux, the development of new OS scheduling policies constitutes a labor intensive task, as the kernel itself needs to be modified. More specifically, testing any scheduling-related kernel modification requires compiling and reinstalling the kernel, and finally rebooting the machine for the changes to take effect. Testing an individual change in this way may as well take a full a coffee break, depending on the features and resources of the target platform and the development host.

To overcome these problems, some researchers have resorted to evaluating their proposed OS-level schedulers via simplistic user space prototypes [3,35,9]. Even though this approach may allow to draw interesting insights and also benefit from leveraging application-level metrics, strategies implemented in this way suffer from the limitations imposed to userland, such as the additional overhead

of context switches and extra system calls required for dynamic thread affinity and performance monitoring [26], or the inability to quickly react to low-level scheduling-related events (e.g., a thread blocks due to I/O or a page fault), thus wasting CPU cycles [11]. In addition, user-level scheduling prototypes cannot access hardware extensions not currently exposed by the OS kernel.

Scheduling frameworks, such as those proposed in [39,26,4] or PMCSched itself, aim to overcome some of the aforementioned limitations. LUSH permits the creation of user-level schedulers for AMPs without special execution privileges, and introduces kernel-level changes to allow fine-grained access to PMCs from user space. Mvondo et al. [26] propose the extension of existing OS APIs, so as to allow the development of kernel-level schedulers programmable from user-space using a safe and controlled environment. LITMUS [4], by contrast, constitutes a substantial fork of the Linux kernel with extensions to facilitate programming of real-time kernel-level scheduling algorithms. Contrary to such solutions –some of them restricted to specific domains [39,4]– PMCSched allows to create custom scheduling-related OS-level modifications without actually patching the kernel. This constitutes a major advantage, as getting profound modifications of the kernel accepted upstream is an arduous task; so much so that researchers tend to forget about that possibility altogether and treat their software as research prototypes with no hope of production integration in sight [26], even after conducting the required security audits. Conversely, the big effort required to maintain multiple project forks for various releases of the Linux kernel often shortens the lifespan of the associated projects [38].

Other studies explore the challenges of OS scheduling on highly heterogeneous architectures [25]. Of special attention is the case of Popcorn Linux [2], which targets heterogeneous systems consisting of nodes with different ISAs, opening the door to parallel ISA-heterogeneous runtime scheduling [24]. These efforts are orthogonal to ours, formulating a problem with several interconnected computing nodes with different processor architectures (e.g., x86 and ARM).

3 PMCSched: Implementation challenges and design

Motivation and challenges. PMCSched is implemented on top of PMCTrack, a performance monitoring tool [33] for Linux that was open sourced back in 2015 [32]. Unlike Perf Events [38] –the default Linux subsystem to access hardware facilities, such as performance monitoring counters (PMCs)– PMCTrack was not primarily designed to only expose hardware monitoring facilities to user space, but to assist the system software when performing runtime optimizations based on these hardware facilities. The operations for which the system software can benefit from PMCTrack include scheduling [34,10] and resource management [11]. The main advantages of relying on PMCTrack for such tasks are its ability to foster new OS-level features as part of an extensible loadable kernel module, and its efficient architecture-independent API to access PMCs within the kernel on a wide range of architectures (x86, ARMv7, ARMv8, etc.). Fig. 1

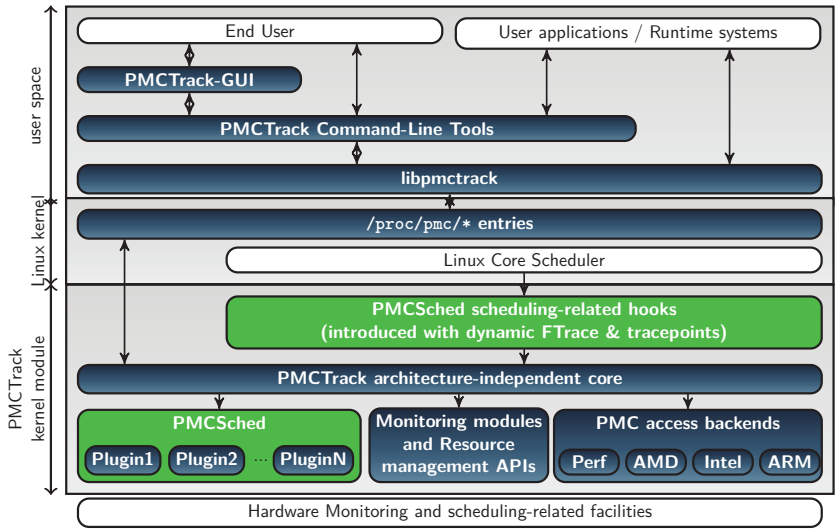

Fig. 1. PMCSched components (in green) inside PMCTrack's architecture.

depicts the various components of PMCTrack and their relationship, described in detail in [33].

With the PMCSched framework we take PMCTrack's potential one step further by enabling rapid development of OS support for scheduling and resource management for Linux *within a loadable kernel module*. We need to highlight this because hitherto new scheduling policies could not be implemented as a kernel module [20,26], since no specific API exists for that purpose within the Linux scheduler. When creating novel OS-level schedulers for Linux without modifying the kernel, three main challenges have repeatedly appeared: (1) the inability to execute code in a kernel module in immediate response to the occurrence of key scheduling-relevant events –context switches, thread creation/destruction, etc.– (2) the lack of a standardized method to seamlessly extend the Linux task structure with new per-thread scheduling related fields that custom schedulers typically require to function, and (3) how to efficiently customize the behavior of the Linux load balancer. Notably, the first two barriers also arise when attempting to manage performance counters at the low level, and for that reason, most PMC tools require changes in the kernel; PMCTrack adds the associated functionality via a small portable kernel patch [33].

Our solution. PMCSched addresses the three aforementioned issues without patching the kernel as follows. First, to be aware of key scheduling events from a kernel module, PMCSched installs scheduling-related hooks (callbacks) leveraging two modern tracing facilities of the Linux kernel: *dynamic ftrace* [29] and *tracepoints* [22]. These two tracing technologies rely on dynamic and static kernel instrumentation, respectively. Noticeably, both are supported on a wide range of processor architectures, and can be found enabled by default on the most popular Linux distributions. Unlike other kernel instrumentation facilities (like Kprobes), these technologies make it possible for a module to be notified when

a kernel function is invoked or when a static tracepoint is reached with virtually no overhead [29,22]. Not only do PMCSched hooks –depicted in Fig. 1– enable the implementation of custom schedulers in a kernel module, but also allowed us to eliminate the need for the PMCTrack kernel patch entirely [27]. Secondly, PMCSched provides a seamless mechanism to extend the task structure with new thread-specific data without modifying the kernel. To this end, whenever a thread enters the system, PMCSched associates a *dummy* software event from the Perf Events subsystem to the thread, by inserting the event into the event list present in Linux's task structure (`perf_event_list` field). The structure of this dummy event (`struct perf_event`) contains a void pointer field (`pmu_private`) that can be utilized to point to any other structure. To simplify the integration of PMCSched in PMCTrack, we use the event's void pointer to point to PMCTrack's per-thread structure (`pmon_prof_t`). PMCSched scheduling fields can be seamlessly added without modifying the kernel, by extending the structures definition inside PMCTrack's kernel module sources.

To make it possible to implement custom load balancing policies, PMCSched introduces the *core group* abstraction. Essentially, cores in the system are organized into different sets (or *core groups*) based on their type (for AMP systems) or their hierarchical relationship in the platform's topology (e.g., cores sharing a last-level cache, or part of the same NUMA node). PMCSched automatically divides cores into different core groups based on system topology, but considering a configurable granularity (LLC, socket or NUMA domain). To implement custom and scalable OS-level load balancing policies or perform specific thread-to-core mappings, a scheduler implemented in PMCSched must assign threads to specific core-groups by using affinity masks. In using this approach, enforcing load balancing across cores within the same group is up to the Linux load balancer, which respects affinity masks. We should also highlight that PMCSched associates a set of linked lists to each core group (spin-lock protected), making it possible to keep track of active threads or multithreaded processes associated with each core group. This design approach allows to make scheduling decisions independently for threads assigned to different core groups, and favors scalable designs that reduce contention in accesses to core-group specific data structures.

A new scheduling or resource management algorithm can be implemented by creating a *scheduling plugin*, which –as illustrated in Fig. 1– becomes a part of the PMCSched subsystem within PMCTrack's kernel module. Building a plugin boils down to instantiating an interface of scheduling operations and implementing the corresponding interface functions in a separate ".c" file within the module sources. The various algorithm-specific operations are invoked from the core part of the scheduling framework when a key scheduling-related event occurs, such as when a threads enters the system, terminates, becomes runnable/non-runnable, or when tick processing is due to update statistics. The framework also provides a set of callbacks to carry out periodic scheduling activations from interrupt (timer) and process (kernel thread) context on each core group separately, thus making it possible to invoke a wide range of blocking and non-blocking scheduling-related kernel API calls, such as those to map a thread to a specific

CPU or core group. This modular approach to creating scheduling algorithms resembles the one used by *scheduling classes* (algorithms) inside the Linux kernel, but with a striking advantage: PMCSched scheduling plugins can be bundled in a kernel module that can be loaded on unmodified kernels. Moreover, plugin developers have access to a rich set of APIs available in PMCTrack, empowering them to configure performance counters seamlessly and retrieve PMC values in a per-thread fashion, to gather data from other hardware monitoring features [33,31], or to govern hardware facilities for shared-resource contention mitigation (e.g., LLC partitioning) available on Intel and AMD processors [11,1].

OS-runtime interaction and Future Work. As previously stated, PMCSched could also be used as a tool to perform system-software optimizations that exploit synergistic interactions between a user-level runtime system and the OS [13,30]. To allow different types of interaction between user space and the kernel, the current version of PMCSched exports a set of special files under the /proc filesystem. For example, the value of configurable parameters of the currently active scheduling plugin can be retrieved/altered by reading/writing from/to those special files. PMCSched also supports the creation of a per-thread page-sized memory region that can be shared between kernel and user space, so as to allow the runtime system to share critical application-level metrics with the OS (e.g., QoS metrics for throughput or latency constraints) and, at the same time, enable the OS to expose information not directly accessible from the runtime system, such as Thread Director performance and energy-efficiency estimates for the current core type where the thread runs [31]. As for future work, and by leveraging this or other communication features –such as netlink sockets–, we plan to implement an OS/runtime interaction scheme to enable efficient execution of multiple data-parallel OpenMP programs on an AMP system, where both layers of the system software play an essential role [30,6].

4 Experimental case study

To demonstrate the applicability of the PMCSched framework, we experimented with a system equipped with an Intel Core i9-12900K "Alder Lake" processor and 32GB DDR4 SDRAM. This AMP processor combines 8 "Golden Cove" big (P) cores, and 8 "Gracemont" small (E) cores. E-cores are grouped into two 4-core clusters, each group sharing a 2MiB L2 cache. P-cores, by contrast, have a private 1.25MiB L2 cache. Every core in the platform integrates a private L1 cache, but shares a 30MiB L3 (LLC) with the remaining E and P cores. With our experiments we evaluate how effectively an OS-level scheduler implemented with our framework can improve the overall system throughput on an Intel Alder Lake processor.

Maximizing throughput on AMPs. Previous research has demonstrated that, to maximize throughput in the context of multi-program workloads, the scheduler needs to be able to (1) determine at runtime the performance benefit that each thread in the workload derives from running on a big core relative to a small one, and then (2) use big cores for running threads that exhibit

a larger relative performance benefit from such cores, while possibly readjusting the mappings dynamically based on program-phase changes. Henceforth, we will refer to the big-to-small performance benefit as the thread's Speedup Factor (SF). Similarly, we will now use the acronym HSF (i.e., High SF) to refer to a dynamic scheduling strategy that aims to maximize throughput by mapping high-SF threads to big cores. While this experimental analysis focuses on workloads consisting of compute-intensive single-threaded applications, it is worth noting that other factors beyond the SF need to be considered for multi-threaded programs, such as latency constraints [12], load balancing and synchronization [6,30], along with other interdependencies among tasks/threads in the application [5].

Implementation of scheduling algorithms. One of the main deltas among the various HSF implementations [19,18,34] is the underlying method employed to determine the SF online. In this work we explore the effectiveness of two SF prediction methods: PMC-based estimation models [18,28,34], and reliance on specific hardware support for SF estimation [16,15]. Regarding the first prediction method, we use the two SF-estimation models proposed in our earlier work [31], which were specifically built for SF prediction from the big and small cores of an Intel Alder Lake processor. The methodology used to build these estimation models [34], the specific performance events they depend upon, and a detailed discussion on their accuracy can be found in [31]. For the hardware-aided SF prediction we leverage the Intel Thread Director (TD) technology, a set of hardware facilities –first introduced in Alder Lake processors– enabling to guide the OS in making thread scheduling decisions on Intel hybrid multi-cores [15,16]. To predict a thread's current SF with TD, the OS must retrieve its TD class (i.e., an integer in {0..3} in the Alder Lake processor we used) by reading a model-specific register, and then calculate the ratio of two performance estimates (for big and small cores) associated with the current TD class; these performance estimates are stored in a memory-resident table that the hardware maintains, which is directly readable from the OS kernel alone.

We experimented with several asymmetry-aware schedulers implemented in PMCSched: an Asymmetry-Aware Round-Robin (AARR) scheduler [21] that equally shares big and small cores among applications; and three variants of the HSF scheduler, which optimize throughput. The first variant of HSF –referred to as HSF/TD– employs Thread Director (TD) to obtain SF estimates. Because in the Alder Lake processor we used such estimates are only accessible directly when the thread runs on a big core (i.e., a valid TD class is not reported from E-cores [31]), our implementation continuously stores TD-based big-core SF estimates on a per-thread history table for different program phases, making it possible to obtain SF predictions indirectly from small cores by accessing the history table. The utilization of history tables to observe patterns from previous samples and predict current and future performance has been widely explored by previous work [39,10]. To deal with frequent *phase misses* when accessing the history table from small cores, our implementation triggers migrations to big cores to gather new big-core estimates, and also implements a throttling

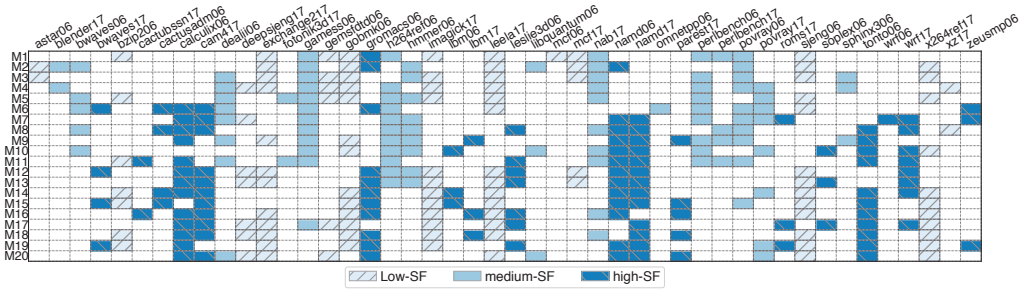

Fig. 2. Workloads used for our experiments. Each row M_i depicts the composition of the i-th workload. A blank cell indicates that the associated program is not included in the workload. Applications whose average SF is lower than 1.7 are considered low-SF programs in our platform, and those with an SF value greater than 2.05 are classified as high-SF. The remaining ones are labeled as medium-SF.

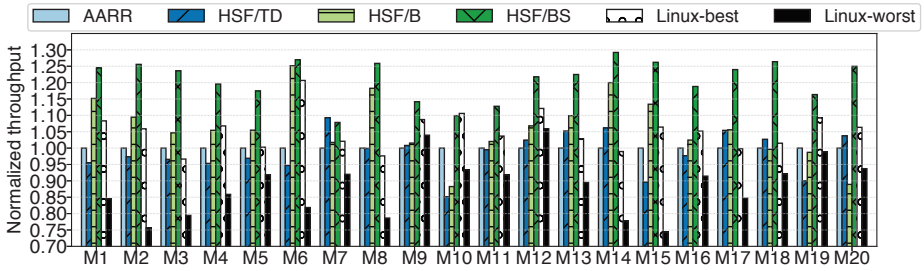

Fig. 3. Normalized throughput delivered by the various scheduling algorithms

mechanism to limit the number of profiling-related migrations, as in [10]. In the second HSF variant, denoted as HSF/BS, the OS continuously gathers a number of per-thread PMC metrics; an up-to-date SF prediction is obtained for a thread by using the metric values as input to the core-specific models proposed in [31] for the big and the small core (the model to use depends on the thread's current core type). Lastly, under the HSF/B variant, SF predictions on big cores are obtained via the same big-core model used by HSP/BS; however, on the small core, predictions are obtained indirectly by reading a history table, populated with past SF estimates retrieved on the big core. Note that this variant was implemented to conduct a fairer comparison with HSF/TD, where direct SF predictions on the small core are unavailable.

Experiments and discussion. For our experiments we randomly generate 20 diverse workloads, comprising of 16 single-threaded programs each. The composition of the various program mixes (Mi) is depicted in Fig. 2, and covers 46 different SPEC CPU applications in total. In launching each program mix, we follow a similar procedure to that of previous works [3,34], so as to ensure the machine's load is constant throughout the experiment. All applications in the workload are started simultaneously, and when one of them completes, the program is restarted repeatedly until the slowest application completes three times. We use the geometric mean of the completion times for each program to calculate the degree of throughput, by using the Aggregate Speedup metric [34,10,31]. All

programs were compiled with GCC 11.2 with the `-O3` and `-mtune=alderlake` compiler switches.

Fig. 3 shows the normalized throughput for the various scheduling algorithms relative to AARR. As a reference, we also provide the best and worst results obtained by Linux default scheduler (CFS) across 10 runs of each experiment, referred to as Linux-best and Linux-worst, respectively. This scheduler is designed to minimize the number of thread migrations, but it is still largely asymmetry unaware [10], and provides highly variable completion times for the same application across multiple runs of the same experiment on Intel Alder Lake processors. Essentially CFS may map an application to a big core for a certain run, and then to a small core in another run, irrespective of its co-runners. This causes large throughput differences across runs, making CFS a misleading baseline [10].

These experimental results undoubtedly reveal that HSP/BS outperforms the other schedulers for most workloads, achieving up to a 30% throughput gain w.r.t. AARR, and providing a 22.9% average improvement against the TD variant. These numbers are tightly related to the superior SF-estimation accuracy provided by the PMC-based models for the big and small core, relative to that of Thread Director, as shown in [31]. Overall, a higher SF-prediction accuracy allows HSF to identify programs with a truly high SF better, and, as a result, the scheduler can grant more big-core cycles to them than to other threads. We further observe that using the big-core model in combination with the history table (HSF/B variant), provides substantially better throughout figures than HSP/TD (averaging 7.9% improvement). However, in a few workloads, such as M10 and M20, HSF/B fails to yield comparable performance to that of AARR. We found that this is caused by the extra thread migrations (and hence the overheads), triggered in response to frequent table phase misses, and aimed at refreshing the history table on big cores. Despite this fact, we conclude that the PMC-based big-core model alone provides superior accuracy than TD, and that the per-thread history table constitutes a reasonably effective method to deal with scenarios where direct SF estimation is not available on certain core types.

5 Conclusions and Future Work

In this paper we have presented PMCSched, a framework for Linux that enables to implement the custom OS kernel support required by new scheduling and resource-management policies for multicore systems. A key distinctive feature of our framework is that it empowers developers and researchers to add new kernel-level scheduling-related support via a loadable module that can be inserted in vanilla (unmodified) versions of the Linux kernel. This favors the adoption in production systems of custom, and potentially sophisticated, scheduling strategies implemented at one or multiple levels of the system software stack. To demonstrate the flexibility of the framework, we leveraged PMCSched's modular plugin-based design to implement several asymmetry-aware OS-level schedulers, and evaluated their ability to improve system throughput under multi-application workloads on an Intel Alder Lake (hybrid) multicore processor.

As for future work, we plan to design novel scheduling and resource management strategies to improve performance when both single-threaded and multithreaded programs are present on the system, making emphasis on potential optimizations that come from the synergistic cooperation between the runtime system and the OS. Lastly, we should highlight that part of the core functionality of PMCSched is already publicly available in PMCTrack's source code repository [27], but that the full framework will be open sourced with the next public release of PMCTrack, scheduled for late 2022.

Acknowledgements Work supported by the EU (FEDER), the Spanish MINECO and CM, under grants RTI2018-093684-B-I00 and S2018/TCS-4423.

References

1. AMD: AMD64 Technology Platform QoS Extensions. `https://developer.amd.com/wp-content/resources/56375.pdf`
2. Barbalace, A., Lyerly, R., Jelesnianski, C., Carno, A., Chuang, H.R., Legout, V., Ravindran, B.: Breaking the boundaries in heterogeneous-ISA datacenters. In: ACM SIGPLAN Notices. vol. 52, pp. 645–659. ACM (2017)
3. Blagodurov, S., et al.: A case for NUMA-aware contention management on multicore systems. In: Proceedings of USENIX ATC '11. USA (2011)
4. Calandrino, J.M., et al.: LITMUS-RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In: 2006 27th IEEE Int'l Real-Time Systems Symposium (RTSS'06). pp. 111–126 (2006)
5. Chronaki, K., et al.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on Int'l Conference on Supercomputing. pp. 329–338. ICS 2015 (2015)
6. Chronaki, K., et al.: On the maturity of parallel applications for asymmetric multicore processors. J. Par. Distrib. Comput. **127**, 105–115 (2019)
7. Costero, L., et al.: Energy efficiency optimization of task-parallel codes on asymmetric architectures. In: Proc. of HPCS '17. pp. 402–409 (jul 2017)
8. Dongarra, J.: Report on the sunway taihulight system. Tech Report University of Tennessee: UT-EECS-16-742 (2016)
9. Feliu, J., et al.: Perf&fair: A progress-aware scheduler to enhance performance and fairness in smt multicores. IEEE Trans. Comput. **66**(5), 905–911 (May 2017)
10. Garcia-Garcia, A., et al.: Contention-aware fair scheduling for asymmetric single-ISA multicore systems. IEEE Transactions on Computers **67**(12) (Dec 2018)
11. Garcia-Garcia, A., et al.: LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores. In: Proc. of ICPP'19. pp. 14:1–14:10 (2019)
12. Haque, M.E., et al.: Exploiting heterogeneity for tail latency and energy efficiency. In: 50th Ann. IEEE/ACM Int'l Symp. on Microarchitecture. pp. 625–638 (2017)
13. Harris, T., Maas, M., Marathe, V.J.: Callisto: Co-scheduling parallel runtime systems. In: Proc. of 9th European Conf. on Comput. Systems. EuroSys '14 (2014)
14. Hennessy, J.L., Patterson, D.A.: A new golden age for computer architecture. Commun. ACM **62**(2), 48–60 (Jan 2019)
15. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 3: System Programming Guide (2021)
16. Intel: Optimizing software for x86 hybrid archiecture. Intel White Paper (Oct 2021)

17. Intel: User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat (2022)
18. Koufaty, D., Reddy, D., Hahn, S.: Bias Scheduling in Heterogeneous Multi-core Architectures. In: Eurosys 10. pp. 125–138 (2010)
19. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: 31st Ann. Int'l Symp. Computer Architecture (ISCA 04). pp. 64–75 (2004)
20. Lepers, B., et al.: Provable multicore schedulers with Ipanema: Application to work conservation. In: Proc. of Eurosys '20 (2020)
21. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multicore architectures. In: Proc. of HPCA '10. pp. 1–12 (2010)
22. Linux: Using the linux kernel tracepoints. https://www.kernel.org/doc/html/latest/trace/tracepoints.html
23. Lozi, J.P., et al.: The linux scheduler: A decade of wasted cores. In: Proceedings of the 11th ACM European Conference on Computer Systems (Eurosys '16) (2016)
24. Lyerly, R., et al.: An OpenMP Runtime for Transparent Work Sharing Across Cache-Incoherent Heterogeneous Nodes. ACM Trans. Comput. Syst. (dec 2021)
25. Mittal, S.: A survey of techniques for architecting and managing asymmetric multicore processors. ACM Comput. Surv. **48**(3), 45:1–45:38 (Feb 2016)
26. Mvondo, D., et al.: Towards user-programmable schedulers in the operating system kernel. In: Proceedings of the 11th Workshop on Systems for Post-Moore Architectures, SPMA 2022 (Apr 2022)
27. PMCTrack: Github repository. https://github.com/jcsaezal/pmctrack (2015)
28. Pricopi, M., et al.: Power-performance modeling on asymmetric multi-cores. In: Proc. of CASES '13. pp. 15:1–15:10 (2013)
29. Rostedt, S.: "ftrace: Where modifying a running kernel all started" https://kernel-recipes.org/en/2019/talks/ftrace-where-modifying-a-running-kernel-all-started/
30. Saez, J.C., Castro, F., Prieto-Matias, M.: Enabling performance portability of data-parallel openmp applications on asymmetric multicore processors. In: 49th Int'l Conference on Parallel Processing. ICPP '20 (2020)
31. Saez, J.C., Prieto-Matias, M.: Evaluation of the Intel Thread Director technology on an Alder Lake processor. In: 13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22) (2022)
32. Saez, J.C., et al.: An OS-oriented performance monitoring tool for multicore systems. In: Proc. of Euro-Par 2015: Parallel Processing Workshops. pp. 697–709 (2015)
33. Saez, J.C., et al.: PMCTrack: Delivering performance monitoring counter support to the OS scheduler. The Computer Journal **60**(1), 60–85 (2017)
34. Saez, J.C., et al.: Towards completely fair scheduling on asymmetric single-ISA multicore processors. Journal of Parallel and Distributed Computing **102** (2017)
35. Salami, B., et al.: Online energy-efficient fair scheduling for heterogeneous multicores considering shared resource contention. J. Supercomput. **78**(6) (apr 2022)
36. Servat, H., et al.: On the instrumentation of OpenMP and OmpSs tasking constructs. In: Euro-Par 2012: Parallel Processing Workshops. pp. 414–428 (2013)
37. Torng, C., Wang, M., Batten, C.: Asymmetry-aware work-stealing runtimes. In: Proc. of ISCA '16. pp. 40–52 (2016)
38. Weaver, V.M.: Linux perf event features and overhead. FastPath Workshop (2013)
39. Xu, V.M., et al.: Lush: Lightweight framework for user-level scheduling in heterogeneous multicores. In: 2021 IEEE 14th Int'l Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). pp. 396–404 (2021)