Evaluation of the Intel Thread Director technology on an Alder Lake processor

Juan Carlos Saez
Facultad de Informática
Universidad Complutense de Madrid
Spain
jcsaezal@ucm.es

Manuel Prieto-Matias
Facultad de Informática
Universidad Complutense de Madrid
Spain
mpmatias@ucm.es

ABSTRACT

Asymmetric multicore processors (AMPs) combine high-performance big cores with more energy-efficient small cores, all exposing a shared instruction-set architecture but different features, such as clock frequency or microarchitecture. In the last decade, most commercial AMP products have mainly targetted the embedded and mobile domains. Today, major hardware players are releasing new AMP-based products that aim to move beyond the mobile niche, towards the desktop/server segments. The Apple M1 SoC or the recent Intel Alder Lake processor family are clear examples of these new AMP systems. Despite their energy-efficiency benefits, AMPs pose significant challenges to the operating system scheduler.

In this paper, we assess the effectiveness of the Thread Director (TD) technology, a set of hardware facillities –first introduced in Alder Lake processors– that provide the OS with hints on the performance and energy efficiency that a thread delivers when running on the various core types. The main focus of our analysis is to evaluate how effectively the OS can drive scheduling decisions with TD's performance hints. To this end, we incorporated support in Linux to conveniently access TD facillites from the OS kernel.

Motivated by various TD's limitations identified with our analysis, we opted to build hardware-counter based prediction models (generated via machine-learning methods) to better aid the OS in making throughput-oriented and fairness-aware scheduling decisions. The effectiveness of both TD and the hardware-counter based models for performance prediction is evaluated both via offline monitoring, and also online, by utilizing our implementation of various asymmetry-aware schedulers in the Linux kernel.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**; • **Computer systems organization** → **Multicore architectures**; **Heterogeneous (hybrid) systems**.

KEYWORDS

Asymmetric multicore processors, Hybrid processors, Scheduling, Operating Systems, Linux kernel, Alder Lake, Thread Director

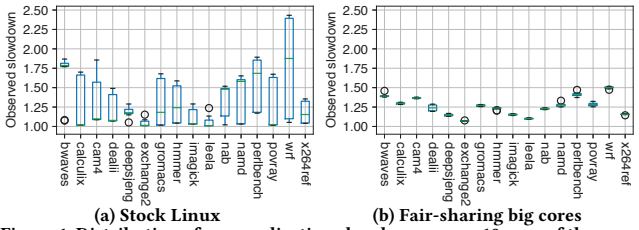
1 INTRODUCTION

Asymmetric multicore processors (AMPs), which integrate high-performance big cores and power-efficient small cores, are capable to deliver higher performance per watt than symmetric multicores for diverse workloads [12, 18, 24, 35]. The general-purpose nature of the various cores, coupled with their shared ISA (instruction set architecture) allows the execution of unmodified (legacy) programs, making AMPs an attractive heterogeneous architecture [20, 21].

In the last decade, the main commercial niche of AMPs was in the embedded and mobile domains. The clear role of big and small cores in this context coupled with the surgent need for extended battery life led to the widespread utilization of AMP processors, such as the ARM big.LITTLE [3]. Specifically, in mobile workloads, big cores are predominantly used for running foreground and latency-sensitive tasks, while background tasks could be mapped to power-efficient small cores. More recently, major hardware players are releasing new products that bring AMP processors to the desktop segment; this is the case of the Apple M1 SoC [2] or the Intel Alder Lake processor family [1]. Despite the potential of AMPs, transparently delivering their benefits to unmodified applications poses big challenges to the system software, and in particular to the OS scheduler, which has to effectively distribute big and small-core cycles among the various threads in the workload [11, 28].

Previous research has demonstrated that to optimize key system metrics, such as throughput, fairness or energy efficiency, the scheduler must factor in the performance benefit that each thread in the workload derives when it runs on a big core, relative to a small one [17, 32, 36]. Henceforth we will refer to this relative benefit as the thread's Speedup Factor (SF). Obtaining an accurate prediction for threads' SFs on-line is generally a challenging task [4, 25, 27], and the fact that the SF may vary over time for a thread across different program phases requires the underlying prediction method to be efficient enough for its practical utilization from the OS [11].

In this paper we evaluate the effectiveness of Thread Director (TD), a set of hardware facillities introduced in Intel Alder Lake processors to aid the OS in making thread scheduling decisions on AMPs [15, 16]. Among other things, TD enables the OS to seamlessly obtain an estimate for the SF of any thread online. Windows 11 is the first operating system that leverages TD features from the process scheduler. Unfortunately, the associated kernel-level implementation remains proprietary, making our study in this OS impractical. In this work, we conduct our analysis in the Linux kernel. We should highlight that Linux default scheduler –the Completely Fair Scheduler (CFS)– does not currently leverage TD and is still largely asymmetry unaware [11]. The latest scheduling-related changes to improve performance on Alder Lake processors (introduced in Linux v5.16) make the load balancer populate idle big cores first, maximizing its utilization. However, despite these changes, CFS still provides highly variable completion times of an application across different runs of the same workload. To illustrate this fact, we measured the completion time of 16 single-threaded SPEC CPU applications, when running together on a 16-core Intel Alder Lake processor. Fig. 1a plots the distribution of slowdowns (i.e., performance degradation vs. isolated big-core execution) for each



(a) Stock Linux (b) Fair-sharing big cores

Figure 1: Distribution of per-application slowdown across 10 runs of the same workload under (a) the default Linux scheduler, and (b) an asymmetry-aware round-robin scheduler, which we implemented in Linux.

application observed across 10 runs of the multi-program workload. As it is evident, an application's completion time may increase by up to 2.3x relative to the fastest run, leading to highly variable system throughput and uneven degree of fairness across executions. This high variability –also present in the latest stable version of Linux (v5.17)– makes CFS misleading when considered as a baseline for experimental analyses on AMPs [11]. The behavior of CFS stands in contrast with the more consistent completion times provided by an asymmetry-aware scheduler that equally-shares big-core cycles among applications, as the results of Fig. 1b reveal.

The major contributions of this paper are as follows:

- We implement the necessary support in Linux to access TD-provided information from the kernel and user space.
- We conduct an offline analysis to assess the degree of accuracy of SF estimations provided by TD over time for a diverse set of compute-intensive programs. Motivated by various TD's limitations, and for comparison purposes, we also opted to build performance-counter based prediction models generated via machine learning.
- We created kernel-level implementations for several existing asymmetry-aware scheduling algorithms [17, 27] that leverage threads' SFs to optimize different system metrics.
- To analyze the impact of the different methods for SF estimation, we carried out an experimental study using a wide range of multi-program workloads running on an Intel Alder Lake processor under various asymmetry-aware schedulers.

The remainder of the paper is organized as follows. Section 2 discusses the closest related works. Section 3 describes Intel Thread Director, and presents our analysis on the accuracy of SF estimation. Section 4 outlines the implementation of the various asymmetry-aware algorithms considered. Section 5 covers the experimental evaluation, and Section 6 concludes the paper.

2 RELATED WORK

Our main goal is to analyze how accurate SF predictions provided by Thread Director are, and how effectively the OS can drive scheduling decisions with these predictions. Existing methods to determine threads' SFs at run time can be grouped in three categories: direct measurement, utilization of performance-counter based prediction models, and reliance on specific hardware support for SF estimation.

Direct measurement, also referred to as IPC sampling [18, 32], comes down to measuring the number of instructions per cycle (IPC) of the thread on both core types with performance monitoring counters (PMCs), and then approximating the SF with the IPC ratio, while factoring in the cores' frequencies. Despite the simplicity of this approach, it usually incurs higher overhead than the other techniques, due to the additional thread migrations across core types required for gathering IPC values [27]. More importantly, this technique is known to provide inaccurate SF predictions due to (1) the utilization of IPC values from potentially different program phases to build an SF estimate [30], and (2) the fact that the IPC may suffer frequent oscillations, even within the same program phase, due to shared-resource contention effects [11].

The second approach consists in gathering different runtime metrics (IPC, cache miss rate, etc.) with PMCs as the thread runs on the *current* core type, and then feeding a prediction model with these metrics to obtain SF estimates [17, 25, 27, 30]. This technique removes the need for thread migrations to read PMCs, but requires building two platform-specific models: one for prediction of the SF from the big core, and another for SF prediction from the small core. Different machine-learning methods have been explored to aid in the generation of SF estimation models [4, 19, 22, 27]. In this work, we opted to utilize the methodology proposed in our earlier work [27], which –to the best of our knowledge– is the only one whose associated models were evaluated and implemented in a kernel-level scheduler for AMPs, like the ones we used.

Lastly, hardware-based mechanisms specifically designed for SF prediction have been also proposed [15, 31]. Intel Thread Director constitutes a clear example of this type of hardware support, and it is the first one of its kind that has been adopted in comercial AMP processors. Unlike Windows 11's, the Linux kernel does not currently feature support for TD. A recent kernel patch from Intel [23] that provides access to the hardware feedback interface (HFI) [15] –a simpler version of TD– will be included in the next stable version of Linux (v5.18). However the purpose of this patch is to expose HFI-provided information only to user space. Our goal, however, is to leverage this information directly from asymmetry-aware schedulers, so as to perform thread-to-core mappings transparently by the OS kernel, without any kind of user intervention.

3 PREDICTING THE SPEEDUP FACTOR

In this section we begin by describing how Intel Thread Director works, and by conducting an experimental analysis on its SF prediction accuracy. Lastly, we outline the mechanism we used to build SF prediction models via machine learning, and assess the accuracy of the models obtained for our Alder Lake platform.

A) Intel Thread Director (TD). TD is a set of hardware facillities enabling to guide the OS in making thread scheduling decisions on Intel hybrid multicores [15, 16]. The TD hardware interface –only directly accessible from the OS kernel– consists of a memory-resident table maintained by the hardware, and a set of per-core and socket-wise registers. Henceforth, we will refer to the memory-resident table as the TD table. As a thread runs on a given logical processor (LP) the hardware provides the OS with feedback on the thread's execution characteristics. To obtain feedback, the OS must periodically read the `IA32_THREAD_FEEDBACK_CHAR` register, which reports the *class ID* (index) of the thread running on the current LP. Scheduling-relevant performance and energy-efficiency values associated with the thread's execution can be retrieved by accessing entries of the TD table associated with that class ID.

In the 16-core Alder Lake processor used in our experiments (Intel Core i9-12900K), TD supports 4 class IDs (0–3), and provides

two feedback values (aka capabilities) for each class and LP: performance (Pe) and energy efficiency (Ef). Pe and Ef values range between 0 and 255. As described in detail in [15] the TD table consists of a global header –made up of a timestamp and several flags that signal recent table changes–, and a set of feedback entries for various LPs that contain the actual feedback values for each class ID. In our platform, the TD table features ten 8-byte feedback entries (1 byte for each capability and class). While each (big) P-core features a separate entry –first eight entries–, a shared feedback entry exists for each group consisting of 4 (small) E-cores –last two entries.

This work focuses on analyzing the effects of throughput and fairness-oriented scheduling decisions made by catering to the SF of the various threads. With Thread Director, an SF estimate can be obtained by dividing the values of the Pe capabilities for the thread's current class ID (as reported by the hardware) stored in P-core and E-core entries of the TD table. So for example, let us consider that, while a thread runs on a P-core, its current class ID is 1. On our system, the Pe values for this class ID on a P-core and on E-core are 77 and 39 respectively, so the estimated SF is 1.97.

We should highlight that reading the `IA32_THREAD_FEEDBACK_CHAR` register does not always provide a usable class ID, as the hardware may not always have enough telemetry information to determine it [15]. In particular, the lower byte of the register, which reports the class ID, is only considered valid when bit 63 of the same register (valid bit) is 1. When the valid bit is 0, the retrieved class ID should be discarded, and then the OS must use the last valid class ID obtained for its scheduling decisions [15]. A major limitation of the hardware TD implementation in our platform, is that while reading the `IA32_THREAD_FEEDBACK_CHAR` register reports a valid class ID most of the time for the current thread on P-cores, an invalid class is always obtained on E-cores. This is the case for all the programs we used (including all SPEC CPU2006 and CPU2017 benchmarks). Being unable to obtain a direct TD estimate for the thread's SF from a E-core poses a challenge for the implementation of TD-based asymmetry-aware schedulers, as discussed in Sec 4.

To evaluate the accuracy of the SF prediction provided by Thread Director, we ran all the benchmarks in the SPEC CPU2006 and CPU2017 suites on both core types for 300B instructions, and compared the actual SF values observed over time with the prediction obtained with TD (P-cores only). To determine the actual SF over time we monitored the IPC with PMCs during the execution of each benchmark on a E-core and a P-core in isolation, and gathered the IPC every 500M retired instructions. The SF for a certain instruction window is calculated with the observed IPC on each core type for that specific window, and by factoring in each core's frequency. The TD-prediction used for the comparison is the average of the SF estimates gathered with during every tick (4ms in our platform) that falls within the current instruction window. Reading TD information at this rate provides negligible overhead [7].

Fig. 2 shows the actual and TD-predicted SF values over time for 20 representative benchmarks, which cover a wide spectrum of profiles regarding prediction accuracy and microarchitectural behavior. TD's implementation in our platform provides a fixed SF estimate for each class: 1.67 for Class 0, 1.97 for Class 1, 2.62 for Class 2, and 1.31 for Class 3. However, 99.9% of the TD class readings in our experiments resulted in Class 0 or 1, so the predicted SF values reported in the figures are mostly 1.67 or 1.97. In fact, Class 3 was never reported for any program, and Class 2 was assigned only to a handful of samples of the `milc` program. This range of SF estimates allows TD to get close to the actual average SF for a few benchmarks (see Figs. 2(a)-(f)), but clearly overpredicts the SF of many other programs (see Figs. 2(g)-(i)). Notably, for programs like `deepsjeng`, `bzip2`, `sjeng` and `exchange` –which exhibit a CPU-intensive execution profile but incur a high number of mispredicted branches per 1K instructions– TD tends to overestimate the SF. The results also reveal that TD may also underpredict the SF for some programs (Figs. 2(m)-(p)), and even obtain predictions that greatly differ from the actual values (Figs. 2(q)-(t)). All in all, TD provides a mean absolute error of 0.38 in the SF prediction across all SPEC CPU benchmarks, and a very low correlation coefficient (<0.1) is observed when comparing predictions and actual values.

B) PMC-based SF models for Intel Alder Lake. The inaccuracies observed in TD-based SF predictions for some programs, motivated us to build PMC-based prediction models by leveraging machine learning techniques. To this end, we used the *Phase-SF* methodology [27], which was proven succesful in building prediction models for the Intel QuickIA prototype system. Phase-SF consists of the following steps. First, a representative set of single-threaded applications (A), and a diverse collection of performance metrics (PM) for comprehensive microarchitectural characterization of programs on both core types must be selected. Second, all applications in A must be run on big and small cores in isolation so as to enable the gathering of all performance metrics in PM over time with PMCs. Notably, PMC values must be collected and reset every time a fixed-size instruction window completes, so as to later match PMC samples for the same program's instruction window gathered on different core types. Third, after merging the per-program PMC data obtained on both core types, and calculating the value of the various performance metrics and the SF for each instruction window, these merged execution profiles are broken down into a set of coarse-grained SF phases using an offline method. Fourth, a summary file is generated for each program based on the previous offline analysis; this file consists of a set of tuples –one for each SF phase–, each including the average of the SF and that of the values for each metric in PM for the PMC samples that belong to the same program phase. Finally, the data of the summary files for all programs is used as input to the additive-regression engine provided by the WEKA machine-learning tool [13]. This results in the generation of two prediction models, enabling SF estimation from the big and from the small core, respectively.

Table 1 enumerates the set of performance metrics that the final estimation models for the big and the small core depend upon, which constitute only a subset of all the metrics we gathered offline. Note that the machine-learning method we used automatically assigns low additive-regression coefficients to less relevant performance metrics in the models, so irrelevant metrics could be automatically discarded from the models [27]. Notably, most of the metrics identified as relevant by the machine-learning engine for SF prediction on the big core are based on the TMA (Top-Down Microarchitecture Analysis) event type, which were recently introduced by Intel to aid in the fine-grained identification of application performance bottlenecks [34]. As a new feature of Intel Alder Lake processors, these TMA metrics can be monitored altogether on

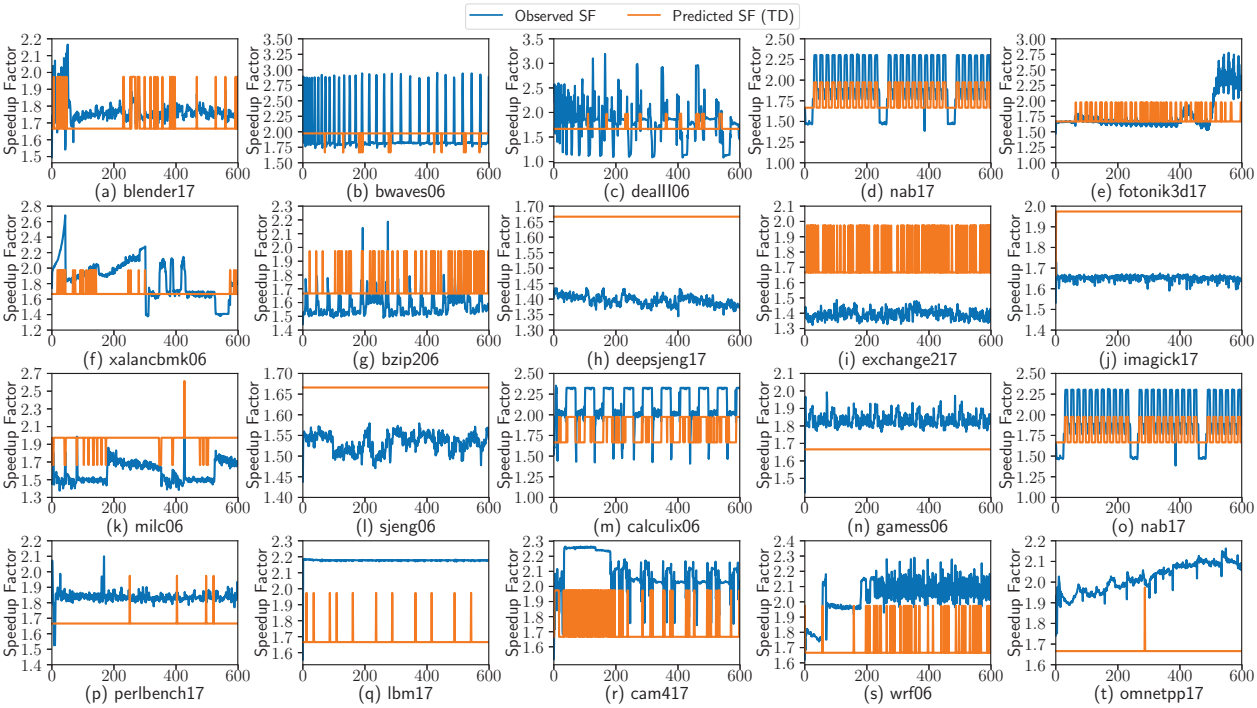


Figure 2: Observed SF values vs. SF predictions over time provided by Thread Director on the P core for various SPEC CPU programs.

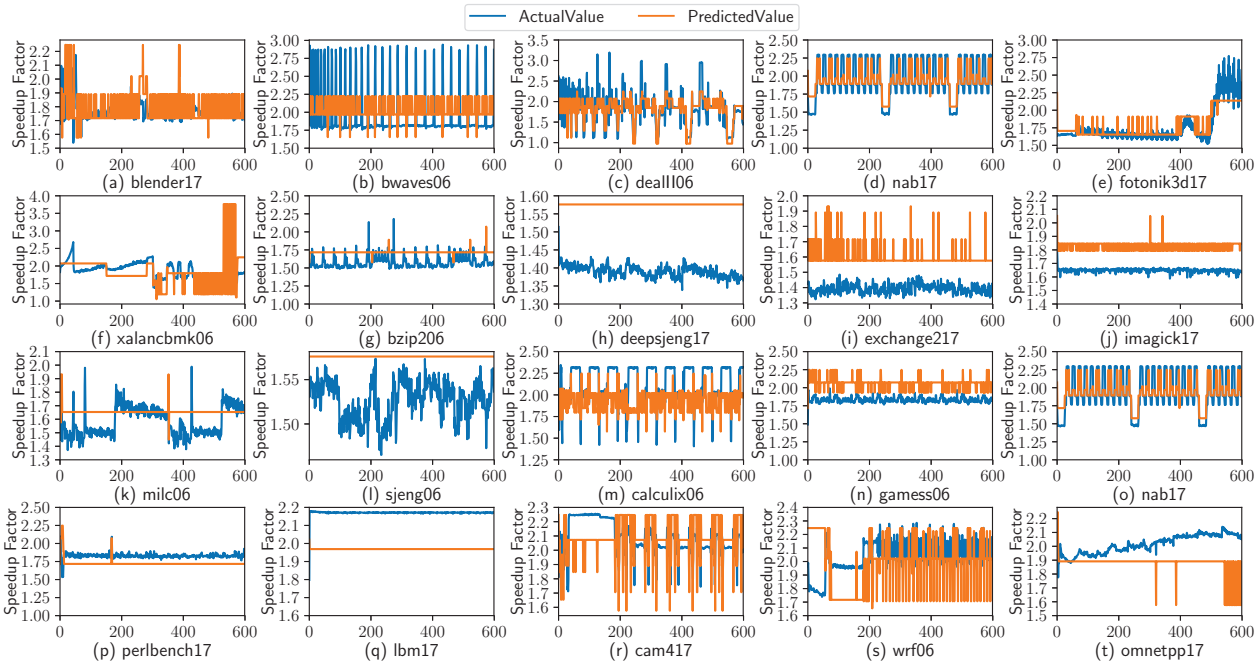


Figure 3: Observed SF values vs SF predictions over time provided by the big-core PMC-based estimation model for various SPEC CPU programs.

P-cores by using a single PMC [15]. In any case, the amount of required PMC events for both estimation models do not exceed the number of physical PMCs available, which greatly simplifies the implementation.

Fig. 3 shows PMC-based SF predictions on the big core for the same benchmarks shown in Fig. 2. The corresponding predictions on the small core, which show slightly better accuracy, were omitted due to space constraints. Although the PMC-based models still lead to important SF mispredictions for some programs (see Figs. 3(i), (j) or (q)), they enable us to get closer to the average SF of the various applications. The correlation coefficients (and mean absolute error) for the estimation on the big and the small core across all 54 SPEC benchmarks are 0.8 (0.19) and 0.84 (0.13), respectively. Note that in generating the models, we used performance data from 150 SF phases corresponding to 15 SPEC programs with a wide range of behaviors regarding branch prediction, memory intensity and cache reuse. In our experimental evaluation of Sec. 5 we use additional applications different from those used to build these models.

Performance metrics	Small core	Big core
Instructions retired per cycle (IPC)	✓	
L3 cache accesses per 1K retired instr.	✓	
Mispredicted branches per 1K retired instr.	✓	
Fraction of pipeline (TMA) slots where no micro-ops are being issued due to lack of back-end resources.	✓	
Fraction of pipeline (TMA) slots wasted due to lack of core front-end resources	✓	✓
Fraction of pipeline (TMA) slots wasted due to incorrect speculation		✓
Fraction of pipeline (TMA) slots wasted due to branch mispredictions		✓
Fraction of pipeline (TMA) slots wasted due to misses in the instruction cache		✓
Ratio of the number of cache lines brought into the L2 and into the L3 (cache reuse indicator proposed in [14])		✓

Table 1: Metrics used for SF prediction on an Intel Core i9-12900K processor.

4 ASYMMETRY-AWARE SCHEDULERS

For our experimental analysis we considered three previously-proposed scheduling algorithms: an Asymmetry-Aware Round Robin policy (AARR), Throughput-Optimized scheduling (TO), and the ACFS algorithm, which strives to optimize fairness.

AARR [20, 32] equally shares big-core cycles among threads. To this end, it keeps track of the amount of clock ticks each thread has consumed on a big core, and triggers thread migrations every so often to ensure an even distribution of big core cycles among threads. Specifically, two threads running on different core types are swapped when RR detects that the difference between the cycles consumed on a big core by both threads exceeds a certain threshold.

TO [17, 18, 32] optimizes throughput by using big cores to run those threads that currently exhibit the highest SF values.

ACFS [27] was proposed to improve the degree of fairness provided by AARR, while delivering better system throughput. Its rationale is to even out the progress made by the various threads by granting a greater fraction of big-core cycles to threads whose performance is more likely to be highly degraded when mapped to a small core. To this end, ACFS assigns each thread an `amp_vruntime` counter, which tracks its progress on the AMP throughout the execution. This counter is incremented every tick consumed by the thread on a big or a small core, and the increment applied reflects both the current core's performance, and the thread's current SF [27]. Because threads mapped on small cores tend to make slower progress than big-core threads, ACFS evens out the progress by swaping threads running on opposite core types when the difference between the associated `amp_vruntime` counters is greater than a certain threshold. We should highlight that in our experiments we did not consider other recently proposed fairness-aware schedulers [11, 29], as they rely on specific assumptions on the cache hierarchy that do not hold on Alder Lake processors.

To develop the various schedulers in the Linux kernel v5.16, we used the PMCSched framework [5], implemented on top of the PMCTrack open-source monitoring tool [26]. This novel scheduling framework makes it possible to implement asymmetry-aware scheduling algorithms in a kernel module that can be loaded in *vanilla* (unpatched) kernels with standard tracing support enabled [5]. To evaluate how sensitive the TO and ACFS algorithms are to the choice of the underlying method of SF prediction, we added support in our framework for three SF-estimation methods, referred to as *Thread Director (TD) based*, *Model based*, and *Big Model based*.

The TD-based method relies on the threads' SF estimates provided by Intel Thread Director, which –in the Alder Lake processor we used– are available directly on big cores only (as stated in Sec. 3A). We observed that triggering periodic migrations from small to big cores so as to obtain up-to-date SF values for threads assigned to small cores, leads to the same program-phase related mispredictions issues of IPC sampling (see Sec. 2). To address these issues –and inspired by previous works [11, 33]– our TD-based method uses a per-thread history table to aid in SF prediction from small cores. A thread's history table –empty when the thread enters the system– stores SF values corresponding to past execution phases obtained with TD on a big core. As in [11], we represent each phase in the table by means of a pair of control metrics gathered on-line: the number of L1 cache accesses per 1K instructions, and the fraction of branch instructions retired. Good properties of these metrics are that they do not vary significantly across core types for a specific phase, and remain stable even if shared-resource contention exists [11]. The scheduler accesses the history table as soon as new values of the control metrics are obtained with PMCs. If the thread is currently running on a big core, the information for the current phase is simply updated with the latest average SF obtained with TD by using the procedure described in Sec. 3A. When the thread runs on a small core, the table is accessed to retrieve an SF prediction. If no information is found for the current phase (i.e., the pair of control metrics is not close enough to any of those registered in the table) the estimated SF used by the scheduler is the average across SF samples in the history table. Notably, when the *phase miss* rate is too high the scheduler forces a migration of the thread onto a big core, thus making it possible to refresh the history table with recent SF estimates. To prevent threads requiring frequent history-table refresh operations from monopolizing big cores, their migration rate is throttled using the mechanism proposed in [11].

The Model-based method leverages the PMC-based SF models presented in Sec. 3B, which were built using machine learning. When the Model-based method is enabled, the OS continuously gathers the PMC metrics that the prediction model of the current core type depends upon (Table 1), and obtains the SF prediction by using the metric values as input to the model's inference function.

Lastly, **the Big-Model based method** was exclusively created to conduct a fairer comparison with the TD-based method under similar restrictions: SF estimates are only available directly on the big core, and SF predictions on the small core are obtained indirectly based on a history table. Note that the history table here stores SF estimates provided by the PMC-based big-core model.

5 EXPERIMENTAL EVALUATION

For the evaluation we used a 16-core platform featuring an Intel Core i9-12900K processor and 32GB DDR4 SDRAM. The processor integrates 8 "Golden Cove" big (P) cores, and 8 "Gracemont" small (E) cores, all sharing a 30MiB L3 LLC. Each P-core has a private 1.25MiB L2 cache. E-cores are grouped into two 4-core clusters; cores in each cluster share a 2MiB L2 cache (L1 is private to each core). To assess the effectiveness of Thread Director in aiding the

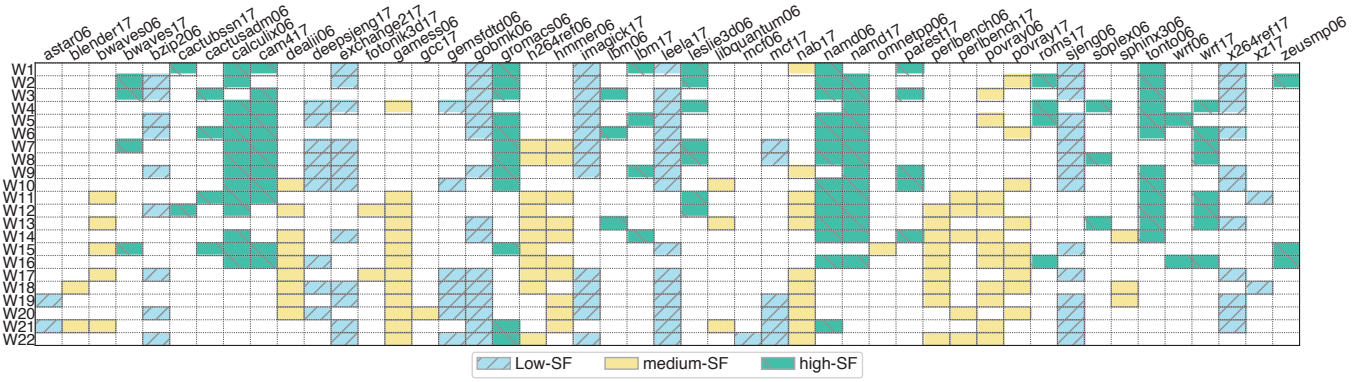


Figure 4: Multiprogram workloads used for our experiments. A blank cell indicates that the associated program is not included in the workload. Applications whose average SF is lower than 1.7 are considered low-SF programs in our platform, and those with an SF value greater than 2.05 are classified as high-SF. The remaining programs are labeled as medium-SF.

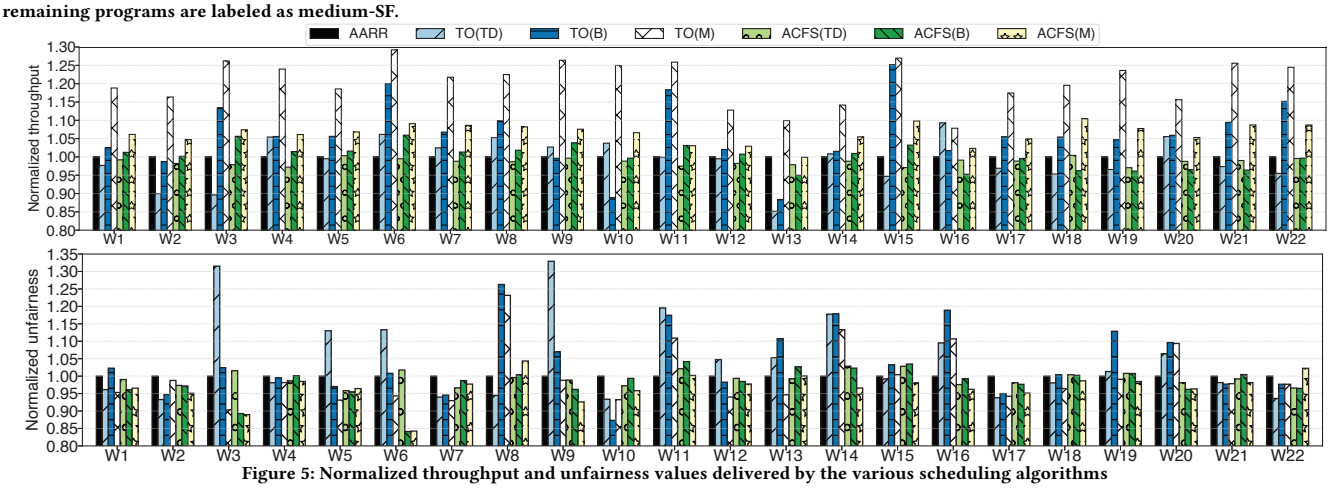


Figure 5: Normalized throughput and unfairness values delivered by the various scheduling algorithms

OS to improve throughput and fairness, we experimented with the AARR, TO and ACFS scheduling algorithms, presented in Sec. 4.

For our experiments we ramdomly built 22 diverse workloads using 47 SPEC CPU programs. The total thread count in each compute-intensive workload was set to match the total number of cores in the AMP, as done in [11, 17, 32, 36]. The composition of the various program mixes is depicted in Fig. 4. The first 10 mixes (W1-W10) consist of programs that cover a wide range of average SF values. The remaining workloads include mostly medium-SF and high-SF programs (W11-W16), or predominantly low-SF and medium-SF applications (W17-W22). In running the workloads, we follow a similar methodology to that of previous works [6, 27, 30]. All applications in the mix are started simultaneously, and when one of them completes, the program is restarted repeatedly until the slowest application in the set completes three times. We use the geometric mean of the completion times for each program to calculate the degree of fairness and throughput, by using the Unfairness [8–10, 33] and Aggregate Speedup [11, 27] metrics, respectively.

Fig. 5 shows the degree of throughput (the higher the better) and unfairness (the lower the better) delivered by the different scheduling algorithms normalized to AARR. The suffix in parentheses in the names of the different variants of TO and ACFS denotes the underlying method used to determine SFs at run time: "TD" –Thread Director (TD) based–, "B" –Big-Model based method–, and "M" for the Model-based method. All in all, the inaccuracies in the TD-based SF prediction do not enable TO and ACFS to accomplish their goals. Specifically, TO(TD) degrades throughput vs. AARR by 1% on average, and provides only a 9.3% maximum throughput improvement (W16). Similarly, ACFS(TD) increases AARR's average unfairness figures by 5%. By contrast, when TO uses the PMC-based estimation models on both core types (M variant), substantial throughput improvements are obtained over the baseline (by up to 29.2%, and by 20.6% on average). These estimation models are also effective for ACFS(M), which achieves up to a 15.8% unfairness reduction (W6), while reaping non-negligible throughput improvements over AARR (10.4% on average). These results come from the higher SF-prediction accuracy provided by the Model-based method, which allows schedulers to better identify actual high-SF programs, and, in turn, to run them on big cores for longer time periods than the other threads. In addition, the figures of TO(M) and ACFS(M) also underscore that minimizing unfairness and maximizing throughput are often conflicting optimization objectives on AMPs [27].

Lastly, we observe that the "B" method is in general more effective than TD, especially when it comes to assisting the TO scheduler; in fact, TO(B) improves throughput by 7,43% on average vs. TO(TD). This observation suggests that the utilization of the history table to retrieve past-SF predictions is effective as long as big-core estimates are minimally accurate, such as the ones we provided by our PMC-based model. However, the modest fairness improvements of ACFS in its different variants also indicate that enforcing fairness requires SF prediction models with a higher degree of accuracy.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have performed an experimental analysis to determine the accuracy of big-to-small speedup (or SF) predictions provided by Thread Director (TD), and also assessed how effectively the OS utilizes these predictions to make scheduling decisions on an Intel Alder Lake processor. For comparison purposes, we also built SF-prediction models for this processor based on performance monitoring counters (PMCs). To this end, we use a methodology that exploits machine-learning methods [27]. To carry out our evaluation we implemented the necessary support for TD in the Linux kernel, to allow different kernel-level scheduling algorithms [17, 20, 27] to access TD-provided SF estimates. Our experiments reveal that the PMC-based models provide better SF-estimation accuracy than TD, and are especially well suited to aid the OS in optimizing throughput. As for future work, we plan on exploiting the hardware cache-partitioning support present in the L2 cache shared by E-cores in Intel Alder Lake to conduct scheduling optimizations. Another interesting research avenue would be the experimentation with future TD-enabled processors that allow the direct and independent gathering of SF predictions from any core type.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Jani Aakash. 2021. Alder Lake Extends Battery Life. *Microprocessor Report* (Sept. 2021).
[2] Jani Aakash. 2021. Apple Ships Its First PC Processor. *Microprocessor Report* (Jan. 2021).
[3] ARM. 2012. Benefits of the big.LITTLE Architecture. http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf. Accessed: 2015-01-10.
[4] Gomatheeshwari B and J. Selvakumar. 2020. Appropriate allocation of workloads on performance asymmetric multicore architectures via deep learning algorithms. *Microprocessors and Microsystems* 73 (2020), 102996.
[5] Carlos Bilbao, Juan Carlos Saez, and M. Prieto-Matias. 2022. Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems. In *Proceedings of Euro-Par 22: Parallel Processing Workshops (to appear)*.
[6] Sergey Blagodurov et al. 2011. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of USENIX ATC '11*.
[7] Ian Cutress and Andrei Frumusanu. 2021. Intel Architecture Day 2021: Alde Lake, Golden Cove and Gracemont Detailed. https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures. Accessed: 2022-03-19.
[8] Reetuparna Das et al. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 107–118.
[9] E. Ebrahimi et al. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *15th Int'l Conf. Architectural Support Programming Lang. and Oper. Syst. (ASPLOS 10)*. 335–346.
[10] J. Feliu et al. 2016. Perf & Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Trans. Comput.* PP, 99 (2016).
[11] A. Garcia-Garcia et al. 2018. Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems. *IEEE Trans. Comput.* 67, 12 (Dec 2018), 1703–1719.
[12] Vishal Gupta et al. 2012. HeteroMates: Providing high dynamic power range on client devices using heterogeneous core groups. In *2012 International Green Computing Conference (IGCC)*. 1–10.
[13] Mark Hall et al. 2009. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.* 11 (2009), 10–18. Issue 1.
[14] Alexandros-Herodotos Haritatos et al. 2016. Contention-Aware Scheduling Policies for Fairness and Throughput. In *Co-Scheduling of HPC Applications (COSH@HiPEAC 2016) (Advances in Parallel Computing, Vol. 28)*. 22–45.
[15] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2022-03-15.
[16] Intel. 2021. Optimizing software for x86 Hybrid Archiecture. *Intel White Paper* (Oct. 2021).
[17] David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Eurosys 10*. 125–138.
[18] Rakesh Kumar et al. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *31st Ann. Int'l Symp. Computer Architecture (ISCA 04)*. 64–75.
[19] Cha V. Li, Vinicius Petrucci, and Daniel Mossé. 2017. Exploring Machine Learning for Thread Characterization on Heterogeneous Multiprocessors. *SIGOPS Oper. Syst. Rev.* 51, 1 (sep 2017), 113–123.
[20] Tong Li et al. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *16th Intl. Symp. High-Performance Computer Architecture (HPCA 10)*. 1–12.
[21] Sparsh Mittal. 2016. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *ACM Comput. Surv.* 48, 3, Article 45 (Feb. 2016), 38 pages.
[22] Daniel Nemirovsky et al. 2017. A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs. In *29th International Symposium on Computer Architecture and High Performance Computing*. 121–128.
[23] Ricardo Neri. 2022. Thermal: Introduce the Hardware Feedback Interface for thermal and performance management. https://lwn.net/Articles/880587/. Accessed: 2022-03-19.
[24] V. Petrucci et al. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *21st Int'l. Symp. High-Performance Comp. Architecture*. 246–258.
[25] Mihai Pricopi et al. 2013. Power-performance modeling on asymmetric multicores. In *Proc. 2013 Int'l Conf. Compilers Architectures and Synthesis for Embed. Syst. (CASES 13)*. 15:1–15:10.
[26] Juan Carlos Saez et al. 2017. PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler. *Comput. J.* 60, 1 (2017), 60–85.
[27] Juan Carlos Saez et al. 2017. Towards completely fair scheduling on asymmetric single-ISA multicore processors. *J. Parallel and Distrib. Comput.* 102 (2017), 115 – 131.
[28] Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. 2020. Enabling Performance Portability of Data-Parallel OpenMP Applications on Asymmetric Multicore Processors. In *49th International Conference on Parallel Processing - ICPP*. Article 51, 11 pages.
[29] Bagher Salami, Hamid Noori, and Mahmoud Naghibzadeh. 2022. Online Energy-Efficient Fair Scheduling for Heterogeneous Multi-Cores Considering Shared Resource Contention. *J. Supercomput.* 78, 6 (apr 2022), 7729–7748.
[30] Daniel Shelepov et al. 2009. HASS: a Scheduler for Heterogeneous Multicore Systems. *Oper. Syst. Review* 43, 2 (2009), 66–75.
[31] K. Van Craeynest et al. 2012. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE). In *39th Ann. Int'l Symp. Computer Arch. (ISCA 12)*. 213–224.
[32] K. Van Craeynest et al. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *22nd Int'l Conf. Parallel Arch. Compilation Techniques (PACT 13)*. 177–187.
[33] D. Xu et al. 2012. Providing Fairness on Shared-memory Multiprocessors via Process Scheduling. In *Proc. ACM Int'l Conf. Measurement and Modeling Comp. Syst. (SIGMETRICS 12)*. 295–306.
[34] Ahmad Yasin et al. 2019. A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors. *ACM Trans. Archit. Code Optim.* 16, 4, Article 46 (dec 2019), 25 pages.
[35] Teng Yu et al. 2020. COLAB: A Collaborative Multi-Factor Scheduler for Asymmetric Multicore Processors. In *18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*. 268–279.
[36] Ying Zhang et al. 2015. Cross-architecture Prediction Based Scheduling for Energy Efficient Execution on single-ISA Heterogeneous Chip-multiprocessors. *Microprocess. Microsyst.* 39, 4 (June 2015), 271–285.