A Collaborative Multi-factor Scheduler for Asymmetric Multicore Processors

Teng Yu*, Pavlos Petoumenos†, Vladimir Janjic*, Mingcan Zhu†, Hugh Leather†, John Thomson*
*University of St Andrews, UK †University of Edinburgh, UK
Email: j.thomson@st-andrews.ac.uk

***Abstract*—Asymmetric multicore processors (AMP) are necessary for extracting performance in an era of limited power budget and dark silicon. We have efficient symmetric schedulers, efficient asymmetric schedulers for single-threaded workloads, and efficient asymmetric schedulers for single program workloads. What we do not have is a scheduler that can handle all three factors affecting AMP scheduling: core affinity, thread criticality, and scheduling fairness.**

To address this problem, this paper introduces the first general purpose asymmetry-aware scheduler targeting multi-threaded multi-programmed workloads. It estimates the performance of each thread on each type of core and it identifies communication patterns and bottleneck threads. With this information, the scheduler makes coordinated core assignment and thread selection decisions that still provide each application its fair share of the processor's time. We evaluated our approach on GEM5 through four distinct big.LITTLE configurations and multi-threaded multi-programmed workloads composed of PARSEC and SPLASH2 benchmarks. Compared against the Linux CFS scheduler and a state-of-the-art AMP-aware scheduler, we demonstrate performance gains of up to 25% and 5% to 15% on average depending on the hardware setup.

I. Introduction

Over 90% of the processor chips are incorporated into embedded devices, such as smartphones and IoT sensors, which are by nature energy limited. This makes energy efficiency and power distribution crucial considerations in the design of new processor chips. Heterogeneous systems, combining processors of different types, provide such energy efficiency for different types of workloads. Among heterogeneous systems, single-ISA asymmetric multicore processors (AMPs) are becoming increasingly popular. They provide more flexibility in terms of runtime assignment of threads to cores, based on which core is the most appropriate for the workload, as well as on the current utilization of cores. As a result of this, efficient scheduling for AMP processors has attracted a lot of attention in the literature [8]. The three main factors that influence the decisions of a general purpose AMP scheduler are:

Core sensitivity. Cores of different types are designed for different workloads. For example, in ARM big.LITTLE systems big cores are designed to serve latency-critical workloads or workloads with Instruction Level Parallelism (ILP). Running other kinds of workloads on them would not improve performance significantly while consuming more energy.

Thread criticality. Executing a thread faster does not necessarily translate into improved performance. An applications might contain *critical* threads, the progress of which determines the progress of the whole application and it is these threads that we want to pay special attention to.

Fairness. In multiprogrammed environments, scheduling decisions should not only improve the utilization of the system as whole, but should also ensure that no application is penalized disproportionately. Achieving fairness in the AMP setting is non-trivial, as allocating equal time slices in a round robin manner to each application does not necessarily result in the same amount of work for each application.

II. Background and Related Work

TABLE I
Qualitative Analysis on Related Work

Approaches	Core Sens.	Fairness	Bottle-neck	Collaborative
Saez, et al. [9]	✓	✓		
Craeynest, et al. [10]	✓	✓		
Cao, et al. [3]	✓			
Joao, et al [6]	✓		✓	
Kim, et al [7]	✓	✓		
Jibaja, et al [5]	✓	✓	✓	
COLAB	✓	✓	✓	✓

A summary with qualitative comparison on the related work is shown in Table I. Among all previous work on AMP schedulers, only Kim and Huh [7] and Jibaja et al. [5] targeted the general case of multi-threaded multi-programmed workloads. The uniformity fairness policy [7] focuses only on fairness and core sensitivity, without provision for bottleneck acceleration. WASH [5] is the closest existing scheduler to ours. It handles core sensitivity, bottlenecks, and maintains fairness for the general scheduling case but controls only core affinity, leaving all other decisions to the baseline Linux scheduler. We use a WASH-like implementation for the Linux scheduler as our state-of-the-art.

III. Multi-factor Coordinated Scheduler

Our scheduling algorithm is implemented by overriding the default Linux task selector `pick_next_task_fair()` and core allocator `select_task_rq_fair()`.

Hierarchical Core Allocator: Our core allocator is primarily guided by the core sensitivity of each thread. To estimate core sensitivity, we use an offline trained model which predicts the slowdown of moving a thread from a big to a little core. We first run each benchmark in single-threaded mode on both full

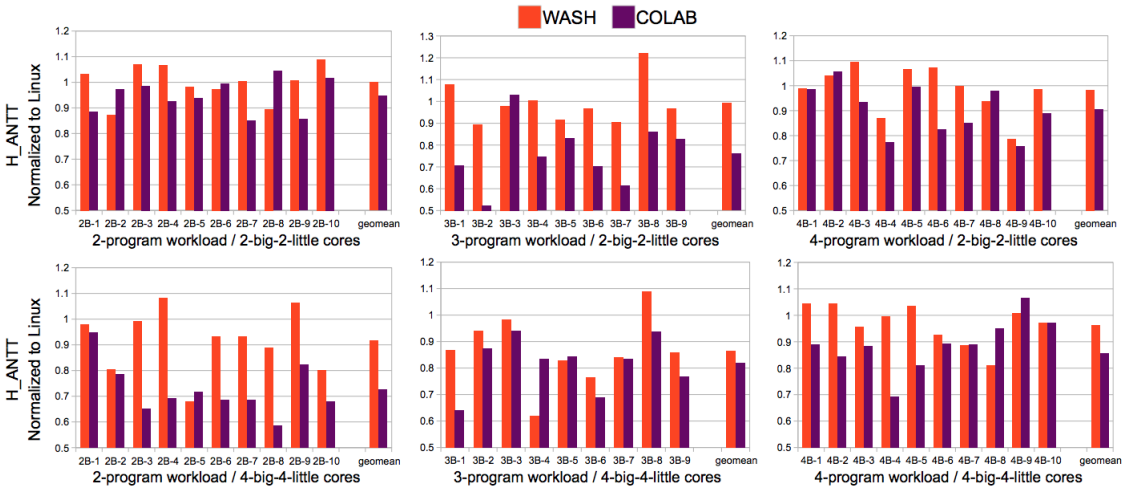

Fig. 1. Heterogeneous Average Normalized Turnaround Time (H_ANTT) of multiprogrammed workloads. All results are normalized to the Linux CFS ones. Lower is better.

little and full big core configurations. We record their runtime and performance counter values, then use PCA and regression to build the predictive model. This predicted slowdown is used to classify them into three classes of increasing core sensitivity. The most sensitive threads are assigned to big cores, the least sensitive to little cores, and the middle group threads are assigned to either big or little cores to keep the load balanced. Finally, we scale the virtual time-slices of threads running on big cores by dividing them by their predicted slowdown to keep relative progress between asymmetric cores equal.

Bias-global Task Selector: The proposed task selector aims to improve criticality in a global way while also balancing it with core sensitivity. During execution there might be multiple bottlenecks that need to be accelerated simultaneously. Other approaches for dealing with thread criticality, such as WASH [5] or ARM GTS [4], migrate all detected bottlenecks to big cores, regardless of core sensitivity and processing load on the destination core. This might result in a costly migration for little to no speedup. Instead, we prefer to accelerate core insensitive bottleneck threads in-place by prioritizing their execution on their assigned core. To monitor communication patterns and identify bottleneck threads, we instrumented the Linux futex. For each thread holding a futex, we measure the cumulative time other threads blocked on that futex. The threads causing the most waiting are labeled as bottleneck.

IV. Experimental Evaluation

We ran our experiments on GEM5 [2], simulating an ARM big.LITTLE-like architecture. The big cores are similar to out-of-order 2 GHz CortexA57 cores. The little cores are similar to in-order 1.2 GHz CortexA53 ones. The OS is Linux v3.16. We cross-compiled the kernel with gcc v5.4.0. We used different benchmarks pulled from PARSEC3.0 [1] and SPLASH2 [11] to generate multi-threaded multi-programmed workloads for evaluation. The results are shown in figure 1. We achieve performance gains of up to 25% and around 5% to 15% in terms of Heterogeneous Average Normalized Turnaround Time compared with the default Linux CFS scheduler and the heterogeneity-aware WASH scheduler. Our proposed scheduler achieves much better performance gain on synchronous-intensive and communication-intensive workloads compared with WASH and CFS, while it cannot outperform others well on workloads with low communication-to-computation ratio.

References

[1] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[4] Brian Jeff. big.little technology moves towards fully heterogeneous global task scheduling. In *ARM White Paper*, 2013.

[5] Ivan Jibaja, Ting Cao, Stephen M Blackburn, and Kathryn S McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*, 2016.

[6] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[7] Changdae Kim and Jaehyuk Huh. Exploring the design space of fair scheduling supports for asymmetric multicore systems. *IEEE Transactions on Computers (TC)*, 2018.

[8] Sparsh Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CSUR)*, 48(3):45, 2016.

[9] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 30(2):6, 2012.

[10] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[11] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA)*, 1995.