Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors

Teng Yu, Runxin Zhong, Vladimir Janjic, Pavlos Petoumenos,
Jidong Zhai, Hugh Leather, John Thomson

Abstract—Asymmetric multicore processors (AMP) offer multiple types of cores under the same programming interface. Extracting the full potential of AMPs requires intelligent scheduling decisions, matching each thread with the right kind of core, the core that will maximize performance or minimize wasted energy for this thread. Existing OS schedulers are not up to this task. While they may handle certain aspects of asymmetry in the system, none can handle all runtime factors affecting AMPs for the general case of multi-threaded multi-programmed workloads.

We address this problem by introducing COLAB, a general purpose asymmetry-aware scheduler targeting multi-threaded multi-programmed workloads. It estimates the performance and power of each thread on each type of core and 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 evaluate our approach using both the GEM5 simulator on four distinct big.LITTLE configurations and a development board with ARM Cortex-A73/A53 processors and mixed workloads composed of PARSEC and SPLASH2 benchmarks. Compared to the state-of-the art Linux CFS and AMP-aware schedulers, we demonstrate performance gains of up to 25% and 5% to 15% on average, together with an average 5% energy saving depending on the hardware setup.

Index Terms—Asymmetric Multicore Processors, Operating System, Scheduling, Performance Model, Energy Efficiency

1 Introduction

Balancing between performance and energy consumption is one of the central issues in designing new processors as over 90% of all processors end up in embedded energy-limited devices, such as smartphones and IoT sensors. *Heterogeneous systems*, that combine different processor types, provide energy-efficeint processing for different types of workloads [26]. The first heterogeneous systems combined processors with different Instruction Set Architectures (ISA). More recently, single-ISA asymmetric multicore processors (AMPs) have gained popularity. Their advantages are obvious - because the processors share the same architecture, the decisions about what task/thread to map to what processor/core can be made at runtime by the OS thread scheduler, guided not only by the characteristics of the workload, but also by the runtime load of individual processors/cores. On the other hand, this introduces an extra degree of freedom to the scheduling problem, making it even more complex. As a result, efficient AMP scheduling has attracted a lot of attention in the literature [23]. The three main factors influencing the decisions of a general purpose AMP scheduler on a heterogeneous system are:

Core sensitivity: Each type of a core is designed to handle different kinds of workloads. For example, in ARM big.LITTLE systems, big cores are mainly used for performance-critical workloads or workloads with Instruction Level Parallelism (ILP). Executing other kinds of workloads on them would not improve the performance significantly, but would significantly increase energy consumption. To build an efficient AMP scheduler, we need to predict which threads are suitable for which kind of core.

Thread criticality: Executing a single thread of a workload faster does not always translate into better performance of the whole workload. If the threads of an application are unbalanced or are executed at different speeds, e.g. because different threads run on different types of cores, the application will run only as fast as its slowest or most critical thread (the thread that blocks most of the other threads). A good AMP scheduler would accelerate these threads as much as possible, regardless of core sensitivity.

Fairness: In multiprogrammed workloads, accelerating an individual application in isolation is not enough if it penalizes other applications. Ideally, we need to have *fair scheduling* that will balance the negative impact of resource sharing uniformly across all applications. In homogeneous sytems, this is easily achieved by giving each application a fixed-size time slice on a CPU in a round-robin way. AMPs make this simple solution unworkable. The same amount of CPU time on different core types results in completely different amounts of work performed, due to difference in performance of each core.

Prior research [7], [8], [11], [14], [28] has explored bottleneck and critical section acceleration, others have examined fairness [21], [22], [30], [31], [35], or core sensitivity [1], [6], [20]. More recent studies [15]–[17], [25], [29] have improved on previous work by optimizing for multiple factors. Such

- *T.Yu, R.Zhong, J.Zhai are with Tsinghua University, China. E-mail: yuteng,zhongrx17,jidongzhai@tsinghua.edu.cn; P.Petoumenos is with University of Manchester, UK. E-mail: pavlos.petoumenos@manchester.ac.uk; V.Janjic is with University of Dundee, UK. E-mail: vjanjic001@dundee.ac.uk; H.Leather is with University of Edinburgh, UK. E-mail: hleather@inf.ed.ac.uk; J.Thomson is with University of St Andrews, UK. E-mail: j.thomson@st-andrews.ac.uk.*
- *J.Zhai is the corresponding author.*

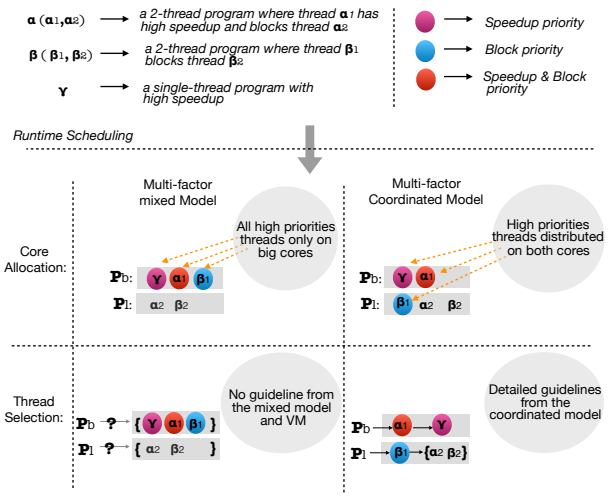


Fig. 1. Motivating Example: Multi-threaded multiprogrammed workload on asymmetric multicore processors with one big core P_b and one little core P_l. Controlling only core affinity results in sub-optimal scheduling decisions.

schedulers are good only for specific kinds of workloads. Only one previous work, WASH [13], can handle general workloads composed of multiple single- or multi-threaded applications with potentially unbalanced threads, and with a total number of threads that may be higher than the number of cores. While a significant step forward, WASH only controls core affinity and does so in a limited way. The former means that we cannot handle core allocation and thread dispatching in a holistic way to speed up the most critical threads. The latter means that WASH only really controls the scheduling domain for each thread, i.e. the group of cores that the thread is allowed to use. The actual core for each thread is chosen by the underlying Linux CFS scheduler with a heuristic that ignores heterogeneity and thread criticality.

In this paper, we introduce COLAB, an OS scheduling policy for asymmetric multicore processors that makes coordinated decisions targeting all three main factors in thread scheduling - core sensitivity, thread criticality and fairness. Our scheduler uses three collaborating heuristics to drive decisions about mapping threads to cores, each of the heuristics foucing primarily on optimisation of one of the factors. Collectively, these multi-factor heuristics result in better thread schedules compared to the Linux and WASH schedulers, therefore improving both performance and energy consumption.

The main contributions of our work are:

- We present the design of a novel AMP-aware OS scheduler that targets general multi-threaded multiprogrammed workloads. The scheduler is based on a set of novel collaborative heuristics for addressing core sensitivity, thread criticality, fairness *and* energy efficiency.
- We present an implementation of the COLAB scheduler both on a real chip and in the simulation settings.
- We evaluate the effectiveness of the COLAB scheduler on a range of standard workloads, demonstrating improvements of up to 25% and 21% (11% and 5% on the average) in the turnaround time compared to the Linux CFS and WASH schedulers in the GEM5 simulator and up to 27% and 10% performance gain (together with 5% energy saving) compared to the ARM GTS and WASH scheduler on a real big.LITTLE development board.

Motivating Example

To demonstrate the problem, consider the example shown in Figure 1, with an AMP system that has a high performance big core, P_b, and a low performance little core, P_l. Three applications are being executed - and that have two threads, and that is single threaded. The first thread of each application, $_1$ and $_1$, blocks the second thread of their application, $_2$ and $_2$, respectively. $_1$ and enjoy a high speedup when executed on P_b. WASH [13], the existing state-of-the-art multi-factor heuristic, would be inclined to assign the high speedup thread and the two blocking threads to the big core. The thread selector of P_b has no information about the criticality of the threads assigned to it, so the order of execution depends on the underlying Linux scheduler. A much better solution is possible if we control both core allocation and thread selection in a coordinated, AMP-aware way. In this case, we map the two threads that benefit the most from the big core, and $_1$, to P_b, while we map the other bottleneck thread, $_1$, to P_l. This will not impact the overall performance of . The thread selector knows $_1$ is a bottleneck thread and executes it immediately. So, what we lose in execution speed for $_1$, we gain in not having to wait for CPU time. Similarly, this coordinated policy guarantees that $_1$ will be given priority over .

2 Background and Related Work

TABLE 1
Qualitative Analysis on Related Work

Approaches	Core Sens.	Fairness	Bottleneck	Collaborative
Kumar, et al [20]	✓			
Li, et al [21]		✓		
Suleman, et al. [28]			✓	
Saez, et al. [25]	✓	✓		
Craeynest, et al. [29]	✓	✓		
Cao, et al. [6]	✓			
Joao, et al [15]	✓		✓	
ARM [12]			✓	
Kim, et al [17]	✓	✓		
Jibaja, et al [13]	✓	✓	✓	
COLAB	✓	✓	✓	✓

Single-ISA heterogeneous processors allow for more efficient processing by using the right kind of core for each workload, while still relying on a single contract between hardware and software [18], [20]. The problem is deciding what "the right kind of core" is. For general purpose systems, where the applications sharing the computational resources are not known at design time and change rapidly, the optimal assignment of threads cannot be made statically. A scheduler has to make this decision at runtime.

The most direct approach, *core sensitivity*, assigns threads to cores where they will experience the highest speedup.

At its simplest, this might mean measuring the number of committed instructions per second on all core types and choosing the highest. In most cases though, this is inefficient. Existing approaches use performance models to predict the speedup from moving a thread to another core without having to execute the thread on that core. They first measure certain aspects of the thread's execution on its current core, *ILP* and *LLC miss rates* in Saez et al. [25], *CPI stack*, *ILP*, and *MLP* in Craeynest et al. [30], empirically selected performance counters in Jibaja et al [13]. Then they use the performance model to estimate the effect of moving the thread elsewhere. The threads most sensitive to their placement are assigned to their preferred core type, the rest are assigned to any type of core.

Whole-program performance depends not only on accelerating core sensitive threads but also on accelerating the threads that slow down the program, the *bottleneck and critical threads*. For example, previous work has shown the benefit from accelerating Amdahl's serial bottlenecks [19] and critical code sections [28]. Joao et al. [14], [15] further showed that functions that cause threads to wait above a certain threshold should be accelerated. Jibaja et al. [13] similarly proposed finding bottleneck Java threads by measuring waiting time on contended locks.

On top of accelerating core sensitive and bottleneck threads, a general purpose AMP scheduler needs to *maintain fairness*, that is to balance the processing resources given to each thread and process when they have to share these resources. Traditional fair schedulers balance only processing time, assuming that time is equally important on all cores. This is not true for AMPs. Li et al [22] introduced some sense of fairness by loading each core proportionally to its processing power. Craeynest et al. [29] instead proposed an *equal progress* scheduler. It uses a performance model to express the progress of every thread in terms of the time required for the same progress on the small core. The scheduler then prioritizes threads so that the small core equivalent time of all threads is the same. Other heuristics have tried to maximize fairness on AMPs [17], [31], [35] but for restricted scenarios.

Only a small number of schedulers are designed to handle the general case of multi-threaded multi-programmed workloads. ARM Global Task Scheduler (GTS) [12] focuses on energy efficiency. Threads run on low power cores unless they are servicing interrupts or have a high load. Core sensitivity is not a concern, while bottlenecks are only accelerated when they happen to be caused by high load threads. Kim and Huh [17] focuses only on core sensitivity and fairness. Inter-thread communication is not a concern. WASH [13] is to our knowledge the only existing scheduler that handles core sensitivity, bottlenecks, and maintains fairness. Still, it is limited to only controlling affinity and requires the OS scheduler to independently dispatch threads to cores. As we see later, this lack of collaboration between the two subsystems is sub-optimal. In the rest of this paper, we use a WASH-like implementation that relies on the baseline Linux scheduler as our state-of-the-art. A qualitative comparison of the related work is shown in Table 1.

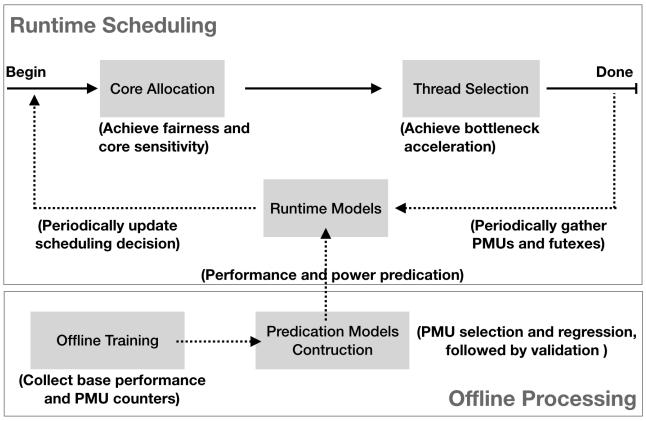


Fig. 2. System Overview

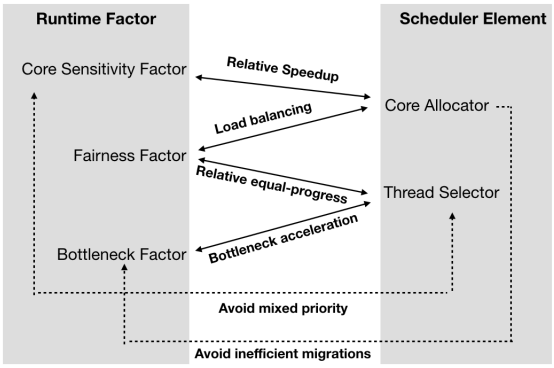


Fig. 3. A diagram of Performance Factors and Relationships with Scheduling Functions

3 System Overview

We provide a high level COLAB system overview as shown in Figure 2. The system is divided into a runtime scheduling process and an offline modelling processes. The runtime scheduler is built inside OS kernel and composed of i) a core allocator to handle fairness and core sensitivity; ii) a thread selector to achieve bottleneck acceleration; and, iii) machine learning based runtime models, which predicate speedup and power consumption of threads on heterogeneous cores. The offline modelling processing is applied to construct runtime models by offline training.

The main novelty of COLAB scheduler is that it can handle multiple runtime factors (core sensitivity, bottleneck acceleration and fairness) in a collaborative way to achieve high system performance and energy efficiency. To easy understand the runtime collaboration, we first analyze the performance impact of multiple runtime factors and their relationship with different scheduler components. We then discuss how to build the scheduler which addresses these performance problems in a coordinated way. After that, we present the detailed design and implementation of the proposed COLAB scheduler, beginning from the supporting offline modelling process to the runtime scheduling process.

4 Runtime Analysis

Figure 3 shows sharing of information between the core allocator and the thread selector in addressing the runtime performance factors. In this section, we analyse the relationship between the runtime factors and the scheduler components that address them before analysing how the information between the scheduler components is exchanged.

4.1 Runtime Factors Analysis

Core Allocator: AMP-aware core allocators are mainly guided by the core sensitivity factors of threads, which quantify how much performance benefit would migrating a thread from a little core to a big core bring. Migrating a *high-speedup thread* (which enjoys large speedups on big cores) from a little core to a big core will generally provide more benefit than migrating a *low-speedup thread*. However, taking into account also the *bottleneck factor*, which quantifies how much the thread blocks other threads, reveals problems with this heuristic on multiprogrammed workloads. Previous approaches [13] simply combine the predicted bottleneck acceleration and speedup together. This can result in sub-optimal scheduling decisions where both bottleneck threads and high speedup threads accumulate in the runqueues of big cores, as described in the motivating example. A better core allocation policy would avoid a simple combination of bottleneck acceleration and speedup, focusing instead on a collaborative environment where big cores focus on high-speedup bottleneck threads and little cores handle low-speedup bottleneck threads without additional migration. Furthermore, core allocators attempt to achieve relative fairness on AMPs by efficiently sharing heterogeneous hardware and avoiding leaving resources idle as much as possible. Simply mapping ready threads uniformly over different types of cores can not achieve true fairness as different types of cores end up having different number of threads prioritized on them. Therefore, a hierarchical allocation should be applied to guarantee the overall fairness, which avoids the need to frequently migrate threads to empty runqueues.

Thread Selector: The *thread selector* decides which thread from the runqueue of each core would be executed next. The thread selector usually prioritizes the bottleneck threads in order to avoid performance penalty from threads being blocked for too long. In a multi-thread multiprogram environment, multiple bottleneck threads from different programs may need to be accelerated simultaneously. Instead of simply detecting the bottleneck threads and assigning them all to big cores, as the previous bottleneck acceleration schedulers do [13]–[15], the thread selector needs to make collaborative decisions – ideally, both big cores and little cores would simutaneously run the bottleneck threads. Core sensitivity is usually unimportant to the thread selector and the decisions it makes are guided solely by bottleneck acceleration. One exception is when the runqueue of a big core is empty and the thread selector is invoked. Only in this case the speedup factors from core sensitivity of ready threads should be considered. Big cores may even preempt the execution of threads on little cores when necessary. Finally, the thread selector is also concerned with fairness. Scaling time slice of threads by updating the time interval of thread selector has been shown to guarantee the equal progress of threads and achieve fairness in multi-threaded single-program workloads [29]. Problems occur with multi-threaded multi-program workloads. Simply ensuring equal progress of all threads is not enough is not enough to guarantee fairness accross programs. The thread selector should ensure that each individual program progresses equally. Using both big and little cores for bottleneck acceleration provides an opportunity for this. The thread selector makes attempt to ensure fairness across programs by accelerating bottleneck threads from all of them as soon as possible.

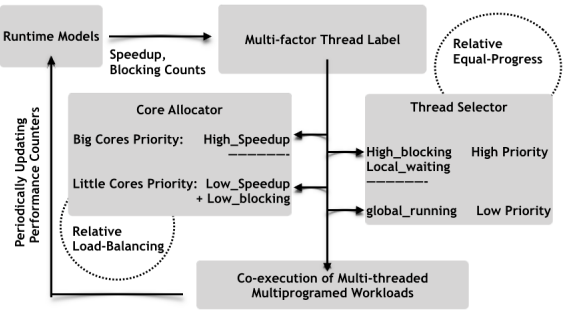


Fig. 4. Coordinated Runtime Scheduling by Multi-factor Collaboration

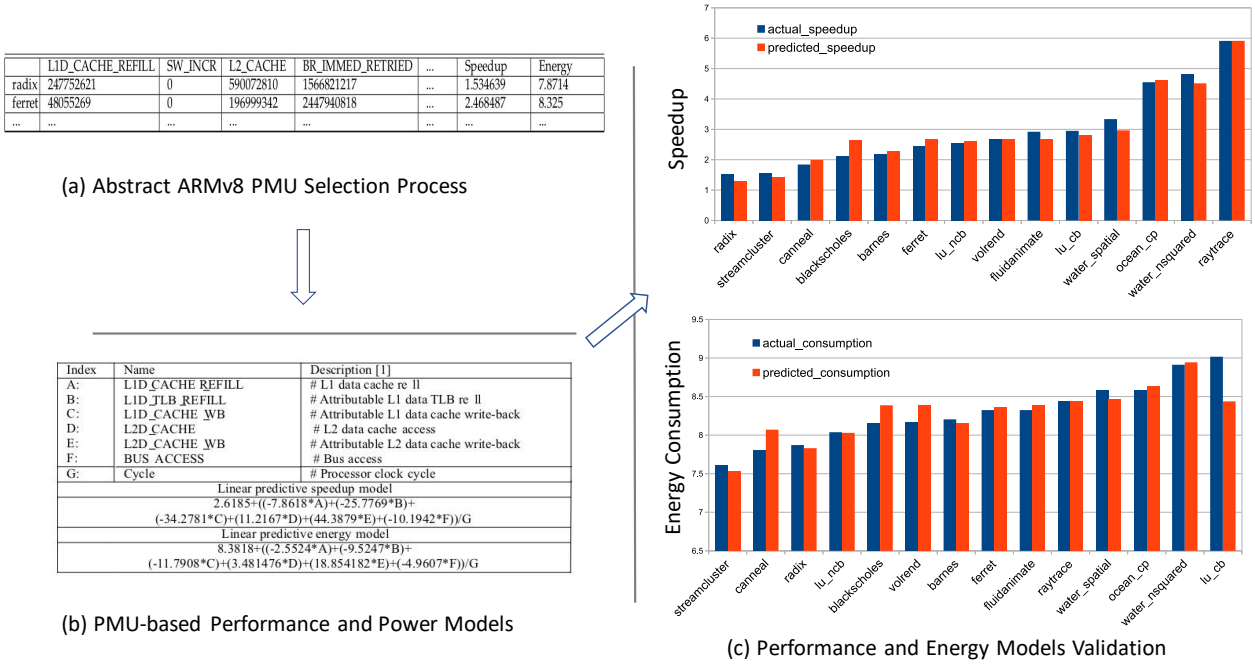
4.2 Runtime Collaboration

To address the problems detailed above, we designed a coordinated multi-factor scheduler in which the core allocator and the thread selector collaborate to achieve a good tradeoff between performance and energy consumption while also ensuring fairness. The flowchart of our model is shown in Figure 4. Collaboration is facilitated by periodically classifying (using labels) ready threads into two different categories, based on runtime models of speedup prediction and bottleneck identification:

Labels for Core Allocation: Threads with high predicted speedup on big cores will be labeled as high priority on big cores. Threads with both low predicted speedup and blocking levels, i.e. non-critical threads, will obtain high priority on little cores (and low priority on big cores). Remaining threads obtain equal priority on either big or little cores – these threads can then be allocated freely to balance the load of cores.

Labels for Thread Selection: Threads with high blocking level will be labeled as high priority for local thread selection. The same priority will be given to these threads regardless of whether they are executed on a big or little core. The label nevertheless records the type of the current core – threads always have priority to be selected by the same type of cores if there exists a core of the same type with an empty runqueue. Running threads on little cores are also labeled as they may be preempted to migrate and execute on big cores when suited, but running threads will never have priority over waiting ready threads.

After the labeling process, fairness, core sensitivity and bottleneck acceleration are represented by labels on threads and can be handled by either the core allocator, the thread selector or both together. Based on this coordinated model, the core allocator and thread selector handle different priority queues from the set of ready threads – their decisions



	L1D_CACHE_REFILL	SW_INCR	L2_CACHE	BR_IMMED_RETRIED	...	Speedup	Energy
radix	247752621	0	590072810	1566821217	...	1.534639	7.8714
ferret	48055269	0	196999342	2447940818	...	2.468487	8.325
...	...	...	...	...	...	...	...

(a) Abstract ARMv8 PMU Selection Process

Index	Name	Description [1]
A:	L1D_CACHE_REFILL	# L1 data cache re ll
B:	L1D_TLB_REFILL	# Attributable L1 data TLB re ll
C:	L1D_CACHE_WB	# Attributable L1 data cache write-back
D:	L2D_CACHE	# L2 data cache access
E:	L2D_CACHE_WB	# Attributable L2 data cache write-back
F:	BUS_ACCESS	# Bus access
G:	Cycle	# Processor clock cycle
Linear predictive speedup model		
2.6185+((-7.8618*A)+(-25.7769*B)+ (-34.2781*C)+(11.2167*D)+(44.3879*E)+(-10.1942*F))/G		
Linear predictive energy model		
8.3818+((-2.5524*A)+(-9.5247*B)+ (-11.7908*C)+(3.481476*D)+(18.854182*E)+(-4.9607*F))/G		

(b) PMU-based Performance and Power Models

(c) Performance and Energy Models Validation

Fig. 5. ARMv8 Offline Performance and Energy Modelling

are not greedy on a mixed multi-factor ranking like WASH. Instead, they provide a collaborative schedule. Another important issue handled by the collaborative multi-factor model is to ensure equal-progress of threads as shown in the upper-right corner of Figure 4. Instead of interfering with the priority and decisions of thread selection, we achieve equal progress in threads by our scaled time slice approach, based on the predicted speedup value of threads running on big cores. The slices of threads on big cores are relative shorter than on little cores. The thread selection function is triggered more often to swap executing threads on big cores, which guarantees the relative equal-progress of threads executed on all cores. The runtime model periodically extracts the performance counters, which represents the current execution environment of multi-threaded multi-programmed workloads on the AMPs. The model then computes the updated runtime factors, including the predicated speedup value and blocking counts. This information is attached to the threads and reported back to the multi-factor labeler for next round. We present our runtime implementations in the next section.

5 COLAB Scheduler Design and Implementation

This section explores the design and implementation of COLAB inside the Linux kernel. We describe our performance and energy models, our modifications of the kernel to support COLAB, and finally the scheduling algorithm itself, including its runtime overheads.

5.1 Offline Performance and Energy Modelling

The decisions of our Core Allocator are primarily driven by core sensitivity. To predict whether a thread is core sensitive or not, we develop a machine learned performance model similar to the ones used in previous works in this area [13], [25], [29]. The model is constructed offline once and is kept purposefully simple, a linear regression model with seven parameters, to minimize the runtime overhead.

The abstract offline PMU selection process is shown in Fig. 5(a). The training data we collect are execution time, energy consumption, and event counts from all performance monitor units (PMU). We run each training application in isolation under two different configurations, first on the little core cluster and then the big core cluster of our evaluation system. We ignore the PMUs for which their value keep to be zero, such as `SW_INCR` (Instruction architecturally executed, Condition code check pass, software increment).

Since we only have access to seven PMUs at any given moment, we repeatedly execute each application while collecting different PMUs, until we have reliable information for all of them. At the end of this process, we have performance event counts, performance scaling between big and little cores, and energy scaling between big and little cores for 14 programs. Through Principal Component Analysis (PCA) [32], we select the six performance events that correlate the most with performance and energy scaling. For example, it is easy to know that the performance and energy will be more dominated by the total data movement than some branch behaviours. As a result, although `BR_IMMED_RETIRED` is a meaningful PMU which counts all immediate branch instructions that are architecturally

executed, but it is not selected after the PCA process as there are more important PMUs such as `L1D_CACHE_REFILL` (L1 data cache refill) and `L2_CACHE` (L2 data cache access). Finally, we use linear regression to associate the six selected events, as well as the clock cycle count, with performance and energy scaling. This results in the two models shown in Table 5(b).

We validate the accuracy of our trained models on each benchmark as shown in Fig. 5(c). The average error across all benchmarks is around 7% for performance scaling prediction and around 1.5% for big core energy consumption prediction.

5.2 Runtime Supporting Techniques

Bottleneck Identification: On modern Linux systems thread synchronization primitives are almost always implemented on top of kernel futexes, regardless of the threading library used. Futex-based mechanisms use a single atomic instruction in user space to acquire or release the futex, if it is uncontested. Otherwise, it calls the kernel which forces the thread to sleep or wakes up sleeping threads respectively. This means that monitoring the blocking patterns between threads requires instrumenting only this interface. Right before an active thread starts waiting on a futex, in `futex_wait_queue_me()` and `futex_lock_pi()`, we record the current time and store it in the thread's `task_struct`. We mirror this with code right before a waiting task is woken up, in `wake_futex()` and `wake_futex_pi()`. At this point we calculate the length of waiting time for the thread and we accumulate it in a field of the `task_struct` of the thread releasing the futex. This enables us to measure the cumulative time each thread has caused other threads to wait. This is our thread criticality metric for the rest of the paper.

Speedup based Scale-slice Preemption: The default preemption mechanism of Linux is triggered every time a new task in enqueued at which point it checks whether the virtual runtime *vruntime* of the incoming task is significantly lower than that of the running task. If this is the case, sharing the processing time fairly requires preempting the running task. While our approach keeps this mechanism intact, we modify vruntime so that equal vruntimes represent equal progress in an AMP system instead of just equal runtime. We do this in the default preemption function `wakeup_preempt_entity()`. If the triggering core is a big one, then the vruntime of the task is divided by the model predicted speedup for the task. This is equivalent to predicting the vruntime required to achieve the same progress on a little core.

5.3 Scheduling Algorithm Implementation

We implement our scheduling algorithm, shown in Figure 5.3, by overriding the default Linux task selector `pick_next_task_fair()` and core allocator `select_task_rq_fair()`. In line with standard Linux notation, we use *rq* and *cur* to represent runqueue and the current task of a core, respectively. In the following paragraphs, we describe the two main functions as well as an energy-aware extension.

```
_core_alloctor_(thread_struct t):
 if t.high_speedup
  return rr_allocator_(big_cores)
 if t.low_speedup & t.low_block
  return rr_allocator_(little_cores)
 else return rr_allocator_(cores)

_thread_selector_(core_struct c):
 if !empty(c.rq)
  return max_block_(c.rq)
 if !empty(c.sched_domain.rq)
  return max_block_(c.sched_domain.rq)
 if c.cpu_mask == big
  return max_block_(c.sched_domain_little.cur)
 else return idle
```

Fig. 6. Collaborative Multi-factor Scheduler Algorithm

```
_core_alloctor_e(thread_struct t):
 if t.high_hyper
  return rr_allocator_(big_cores)
 if t.low_hyper & t.low_block
  return rr_allocator_(little_cores)
 else return rr_allocator_(cores)
```

Fig. 7. Energy-aware COLAB Extension Algorithm

Hierarchical Core Allocator: The core allocator's role is to assign newly ready threads to core runqueues. Threads become ready to be executed, either when they are woken or spawned. Our implementation first assigns threads to clusters, the big core or the little core one, based on the speedup and blocking labels selected for each thread at runtime. Threads labeled as high speedup are assigned to big core cluster, while low speedup and low blocking are assigned to the little core cluster. All other ready threads, either low speedup and high blocking or average speedup and low blocking, are assigned to both clusters. Finally, a hierarchical round-robin mechanism `rr_allocator_()` chooses one core within the selected cluster for the thread. This helps keep the system load balanced.

Biased-global Thread Selector: The thread selector's primary objective is to accelerate the most critical/blocking threads as soon as possible. The selector always searches for a ready thread from the local runqueue first, preferably a high blocking one. If there are no ready threads and migration is beneficial, the core triggers the migration of a candidate thread waiting in another runqueue. The highest blocking thread will be selected for migration. To reduce the overhead of accessing state in other runqueues, we follow the same principle as the default Linux CFS scheduler, returning the best candidate thread from the local core group first. Further, we allow a big core to select and preempt a running thread on a little core to accelerate it. Big cores are allowed to go idle only when there are no ready threads left. The converse, little cores preempting big cores is not allowed. The equal-progress for achieving fairness is addressed by the scale-slice preemption checker described earlier: we give each thread a maximum time slice according to its expected performance on a little core.

Energy-aware Algorithm Extension Until now, we have only considered how to make scheduling decisions to maximize overall performance. Given the different performance-power characteristics of different cores in an AMP system, scheduling decision can be also used to maximize energy efficiency or to achieve a certain trade-off between performance and energy. So, we extend the original scheduler to consider both optimization targets. We use a hyper-heuristic to guide the core assignment of a thread t:

$$hyper\_(t) = w_s * speedup(t) + w_e * energy(t)$$

where w_s and w_e are the pre-defined weights of the speedup factor and energy scaling factor, which shows user specified trade-off between performance gain and energy saving; $energy(t)$ is the energy scaling factor for the thread t between big and little cores, practically the ratio of consumed energy for the same amount of work between the two core types. We train the factors of this hyper-heuristic using a similar approach as for the performance and energy models. We collect data (speedup and energy efficiency scaling) for our training programs running on the two core types and then use combined regression to obtain the hyper-heuristic.

The extended energy-aware core allocator `rr_allocator_e()` for the COLAB scheduling algorithm is presented in Figure 5.3. The hyper-heuristic essentially replaces the speedup factor in the original `rr_allocator_()`. This allocation policy avoids placing threads on the big cores if their performance does not improve significantly or their energy efficiency deteriorates, instead of considering performance only.

5.4 Scheduling Overhead Analysis

The overhead of the scheduling algorithms themselves is negligible. Collecting the information needed to make our scheduling decisions has some overhead though. Our performance and energy models require accumulating per task performance event counts from seven PMUs. This means that we have to access these PMUs per context switch. Reading each one takes 4 cycles on the Cortex-A53 and 14 cycles on the Cortex-A73. Since we need to read seven units, the worst case overhead is within 100 cycles per context switch. The typical time between context switches for a single thread is orders of magnitude more than a hundred cycles, so the cost of reading the PMUs should be negligible too.

Identifying blocking threads requires instrumentation in the latency-sensitive futex code. Still, the code we added is short and relatively infrequently called, and the time of labeling blocking level for a thread is about 512 nsec.

Finally, we need to use this information to label all threads as high/low speedup, high/low energy scaling, and high/low blocking. This does requires some processing time but we purposefully kept the model simple, a linear regression with seven parameters. Additionally, labels are updated once every 10 msec, and the process of labeling only costs 50-260 nsec, so the total observed overhead is low.

6 Experimental Evaluation

6.1 Experimental Setup

Experimental Environment: We first ran our experiments on GEM5, simulating an ARM big.LITTLE-like architecture. The big cores are similar to out-of-order 2 GHz CortexA57 cores, with a 48 KB L1 instruction cache, 32 KB L1 data cache and 2 MB L2 cache. The little cores are similar to in-order 1.2 GHz CortexA53 ones, with a 32 KB L1 instruction cache, 32 KB L1 data cache and 512 KB L2 cache. We evaluated four distinct hardware configurations on GEM5: 2B2S, 2B4S, 4B2S, 4B4S, where B denotes big cores and S denoted little cores. We then validate COLAB and its energy-aware extension on a HiHope Hikey 970 development board with 4 Cortex-A73 big cores at 2.36GHz and 4 Cortex-A53 little cores at 1.8GHz. As there is only 4 big cores in total to produce baseline performance, we evaluate four configurations: 1B1S, 1B3S, 2B2S and 3B1S. The OS is Linux v4.9. We cross-compiled the kernel with gcc v5.4.0. The power of whole board is measured by a power meter.

TABLE 2
Benchmarks categorization [3], [27], [33]

Name	Sync. Rate	Comm/Comp Ratio
blackscholes	low	high
bodytrack	medium	high
dedup	medium	high
ferret	high	medium
fluidanimate	very high	low
freqmine	high	high
swaptions	low	low
radix	low	high
lu_ncb	low	low
lu_cb	low	low
ocean_cp	low	low
water_nsquared	medium	medium
water_spatial	low	low
fmm	medium	low
fft	low	high

Workloads: For our workloads we used 15 different benchmarks (Table 2), pulled from PARSEC3.0 [2] and SPLASH2 [33]. We only use the *simsmall* inputs on GEM5 as it is well-known that the simulation is extremely slow. We group the benchmarks based on two criteria: a) synchronization intensity and b) communication vs computation intensity. The grouping of the benchmarks into these categories only relates to how we structure our evaluation. It does not inform how WASH or COLAB will handle them.

For each group, we randomly generate workloads with variable numbers of benchmarks and threads. These workloads allow us to investigate the behavior of the three scheduling policies under different extremes. We then use large mixed workloads with the *simlarge* inputs on HiHope Hikey 970 to explore the general cases and validate COLAB with its energy-aware extension on the real chip with asymmetric multi-core processors. Table 3 shows the selected workloads.

Fairness Metrics: Fairness is significant for an effective scheduler, especially for an asymmetry-aware one. Unfairness can bring a number of undesirable effects the whole system. In order to show the fairness of COLAB, we follow the notion of fairness from previous work [10], and use the corresponding *Unfairness* metric:

TABLE 3
Multi-programmed Workloads Compositions

Synchronization-intensive VS Non-synchronization-intensive Workloads			
Index	Workload Composition	Synchronizations	Threads
Sync - 1	water_nsquared - fmm	intensive	4
Sync - 2	dedup - fluidanimate	intensive	18
Sync - 3	water_nsquared - fmm - fluidanimate - bodytrack	intensive	9
Sync - 4	dedup - ferret - fmm - water_nsquared	intensive	20
NSync - 1	water_spatial - lu_cb	non-intensive	4
NSync - 2	blackscholes - swaptions	non-intensive	16
NSync - 3	radix - fft - water_spatial - lu_cb	non-intensive	8
NSync - 4	blackscholes - ocean_cp - lu_ncb - swaptions	non-intensive	20
Communication-intensive VS Computation-intensive Workloads			
Index	Workload Composition	Comm/Comp	Threads
Comm - 1	water_nsquared - blackscholes	Communication-intensive	4
Comm - 2	ferret - dedup	Communication-intensive	16
Comm - 3	water_nsquared - fft - radix - bodytrack	Communication-intensive	9
Comm - 4	blackscholes - dedup - ferret - water_nsquared	Communication-intensive	20
Comp - 1	water_spatial - fmm	Computation-intensive	4
Comp - 2	fluidanimate - swaptions	Computation-intensive	17
Comp - 3	lu_ncb - fmm - water_spatial - lu_cb	Computation-intensive	8
Comp - 4	fluidanimate - ocean_cp - lu_ncb - swaptions	Computation-intensive	20

Large Mixed Multi-programmed Workloads for Real System Validation					
Index	Workload Composition	Threads	Index	Workload Composition	Threads
Large - 1	radix - ocean_cp	5	Large - 3	blackscholes - radix - fluidanimate - water_spatial	15
Large - 2	ferret - swaptions	24	Large - 4	ocean_cp - ferret - lu_cb - swaptions	29

$$Unfairness = \frac{MAX(Slowdown_1; :::; Slowdown_n)}{MIN(Slowdown_1; :::; Slowdown_n)}$$

Where n is the number of programs in the workload and the unfairness is mainly measured by the difference from program slowdown. The slowdown of each program is defined by its execution time when it run alone over the time when it run in a mixed workload under the same hardware configuration. The Unfairness metric is better when lower.

Performance Metrics: Our evaluation uses two metrics to quantify scheduling efficiency: *Heterogeneous Average Normalized Turnaround Time* (H_ANTT) and *Heterogeneous System Throughput* (H_STP). They are based on ANTT and STP, as introduced in [9]. Both ANTT and STP use as their baseline the runtime of each application when executed on its own, i.e. when there is no resource sharing and scheduling decisions have little effect. ANTT is the average slowdown of all applications in the mix relative to their isolated baseline runtime. STP is the sum of the throughputs of all applications, relative to their isolated throughput.

For AMPs, these two metrics fail to work as intended. The runtime when executed alone is still affected by scheduling decisions, e.g. which threads to run on big cores. To overcome the problem, our modified metrics H_ANTT and H_STP use the runtime of each application in the mix when executed alone *on a system where there are only big cores*. If the turnaround time of each application i while being co-scheduled is T_i^M and the turnaround time for the same application when running alone on a big-only system is T_i^{SB}, then:

$$H\_ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i^M}{T_i^{SB}};\;\; H\_STP = \sum_{i=1}^{n}\frac{T_i^{SB}}{T_i^M}$$

H_ANTT is better when lower, H_STP is better when higher. For most figures, we further normalize our results relative to the Linux CFS results for the same configuration and workload.

Schedulers: We evaluate COLAB by comparing it against the default Linux CFS scheduler [24] and a state-of-the-art realistic scheduler based on WASH [13]. Linux CFS is the default scheduler on the GEM5 simulator and ARM GTS [12] is the default scheduler on the development board. They provide fairness while trying to maximize the overall CPU resource utilization. In this work, we do not compare against EAS, the Energy Aware Scheduling system, which is included with recent versions of the Linux kernel to improve energy efficiency on heterogeneous systems. While an improvement over GTS in general, it falls back to CFS by design when CPU utilization is high. This is the case for almost all of our experiments, so EAS would have little effect on the results.

6.2 Experiments on GEM5

In this section, we evaluate the performance of the COLAB scheduler for multi-threaded multi-program workloads using the GEM5 simulator. We demonstrate that COLAB outperforms both the Linux CFS and WASH when there is room for improvement. In particular, where there is a limited number of big cores and/or where the workload contains communication-intensive benchmarks, it is beneficial to consider core affinity and thread bottlenecks *at the same time*. In this setting, the advantages of COLAB over Linux CFS and WASH become apparant. In the rest of this subsection, we examine in more detail the behavior of COLAB under four different hardware configurations (2B2S, 2B4S, 4B2S, 4B4S) for the five different classes of workloads shown in Table 3.

Synchronization-intensive vs Synchronization Non-intensive workloads: The *synchronization-intensive* workloads comprise benchmarks with high synchronization rates caused by locks, barriers and conditions, having a large number of bottleneck threads. We expect that COLAB should be able to schedule them better than CFS and WASH. Conversely, *synchronization non-intensive* workloads should provide fewer opportunities for COLAB to improve on CFS and WASH.

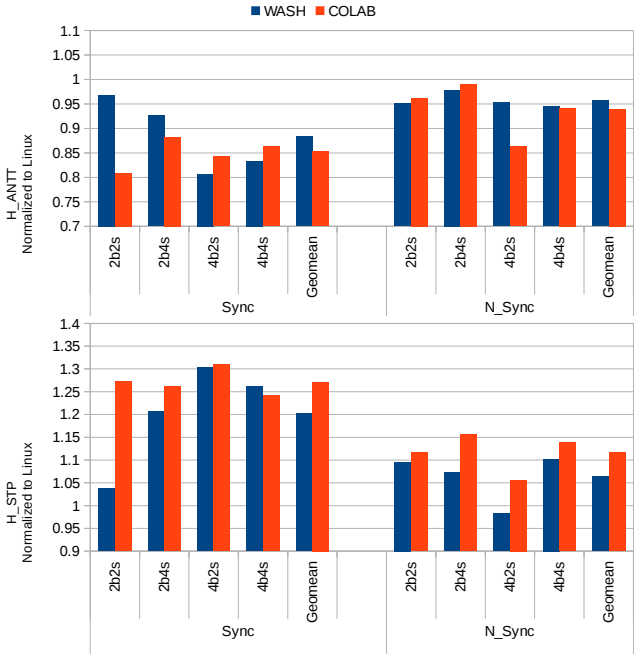


Fig. 8. Performance of Synchronization-Intensive and Non-Synchronization-Intensive Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

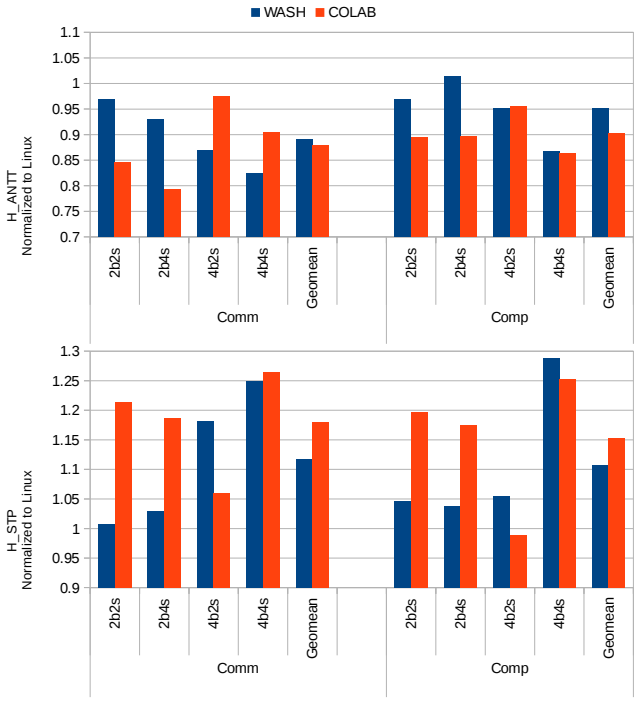


Fig. 9. Performance of Communication-Intensive and Computation-Intensive Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

Figure 8 shows the performance of all three schedulers for each workload class and hardware configuration. The two plots show the average H_ANTT (top) and the average H_STP (bottom). In each plot, we show the results for both the synchronization-intensive (*Sync*, left half) and synchronization non-intensive (*N_Sync*, right half) classes of workloads. The results confirm our expectations. We can observe that COLAB improves the turnaround time of *Sync* workloads by around 15% and 4% on average compared to CFS and WASH, respectively. We can also see that hardware configurations with low core counts (e.g. 2B2S) favor COLAB, allowing it to reduce turnaround time by up to 20% compared to CFS and by up to 16% compared to WASH. With fewer cores, run queues of cores become longer, requiring careful balancing between bottleneck acceleration and core sensitivity. WASH places all bottleneck threads onto the big cores, making these cores congested and ending up with only 3% performance improvement over CFS. COLAB handles these bottleneck threads in a more holistic way, improving turnaround time by 20% and system throughput by 27%, compared to CFS. On the other hand, the *N_Sync* workloads contain fewer bottleneck threads, making scheduling decisions much easier. As a result of this, both COLAB and WASH perform similarly to CFS, with COLAB improving average turnaround time by 6% and average system throughput by 12% compared to CFS. An interesting point is that COLAB significantly outperforms (by 10% and 15% in turnaround time) WASH and Linux for *N_Sync* workloads on the 4B2S configuration. In this case, there is not enough critical threads to utilize big cores and WASH keeps needlessly migrating predicted critical threads between big cores. COLAB, on the other hand, makes intelligent decisions by keeping more threads on little cores, giving more more chance to big cores to execute the few *really critical* threads as soon as possible.

Communication-intensive vs Computation-intensive workloads: Benchmarks with high communication-to-computation ratio are likely to have multiple bottleneck threads. It is critical to accelerate these theads in order to achieve a good performance, making this an ideal scenario for COLAB. On the other hand, workloads with lower communication-to-computation ratio are easier to schedule, so CFS and WASH should do reasonably well there, leaving little space for improvement.

The results for these two classes of workload, *Comm* and *Comp*, are given in Figure 9. We can see that both COLAB and WASH improve over CFS for communication-intensive workloads. However, they offer different advantages on different hardware configurations. COLAB distributes the bottleneck threads to both big and little cores, which is extremely important when having only two big cores (2B2S and 2B4S configurations). COLAB improves the turnaround time up to 21% compared to CFS and up to 15% compared to WASH on 2B4S configuration. When more big cores are available, WASH outperforms COLAB, as it keeps all bottleneck threads on big cores. On these configurations, WASH improves turnaround time by up to 18% over CFS (on the 4B4S configuration) and up to 10% over COLAB (on 4B2S configuration). On average, COLAB reduces turnaround time by around 12% compared to CFS and 1% compared to WASH for the communication-intensive workload class.

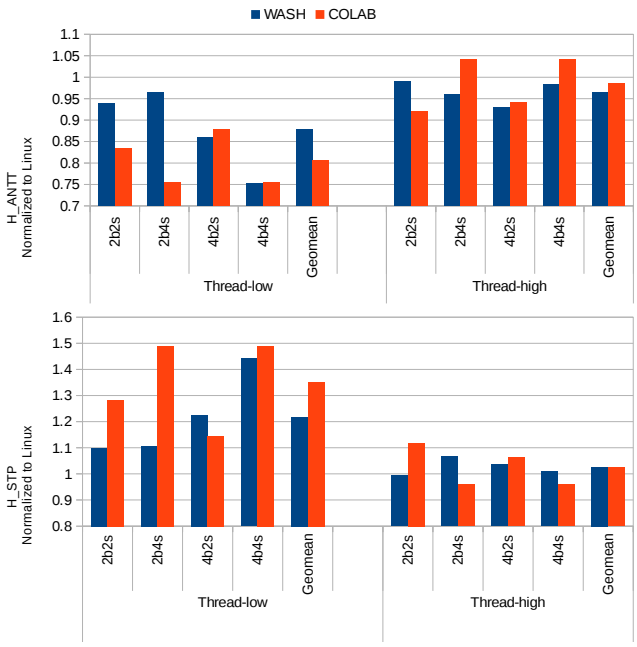


Fig. 10. Performance of low number of application threads and high number of application threads Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

Figure 9 also shows that there are few opportunities for better scheduling of computation-intensive workloads. However, even in this setting, COLAB does better than WASH and Linux. Its turnaround time and system throughput are improved by around 10% and 15%, respectively, compared to CFS and 5% compared to WASH. This is, again, due to a fact that multiple bottleneck threads are distributed both to big and little cores, which results in more efficient use of the available hardware resources for the few bottleneck threads that are present.

Thread and program count: The final set of simulation results show the impact of the number of threads in a workload to the performance of all the schedulers. In Figure 10, Thread-low denotes the workload where there are less threads than cores in the system, whereas Thread-high denotes the workloads where there are at least twice more threads than cores available. We observe that both COLAB and WASH perform significantly better than CFS for Thread-low workloads. Fewer threads make it easier to identify bottleneck threads and give them the resources they need - either by migrating them to big cores (WASH and COLAB) or by prioritizing them on little cores (COLAB). With limited big core resources, COLAB outperforms WASH because it distributes bottleneck threads to all available cores, avoiding overloading the few big cores and leaving the little cores idle. COLAB outperforms Linux up to 25% (2B4S) and WASH by up to 21% (2B4S) in turnaround time. On average, COLAB improves turnaround time and system throughput by around 20% and 35% compared to CFS and around 8% and 11% compared to WASH for workloads with a low number of threads. For workloads with a high thread count, neither COLAB nor WASH are able to improve much on Linux. Overloading the system with threads means that,

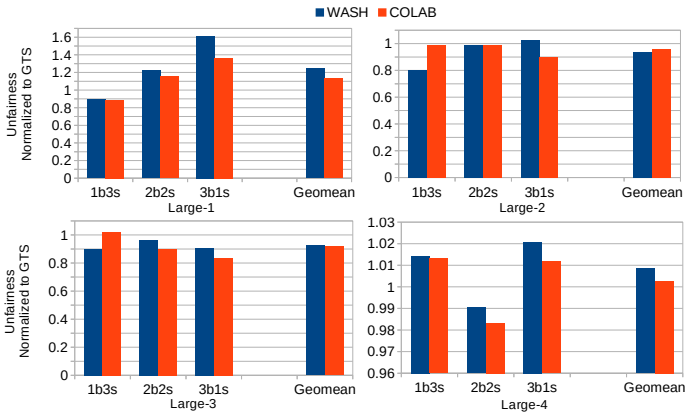


Fig. 11. Unfairness of the Large-1,2,3,4 workloads using different 4-core configurations. All results are normalized to the ARM GTS one. Lower is better.

regardless of where we place threads, all cores will have long runqueues. In this case, COLAB and WASH increase the management overhead (including more frequent thread migrations) while offering little benefit, leading to performance degradation. Of the two, COLAB migrates threads more frequently, due to its scale-slice technique. This results in a slightly worse performance than WASH. On average, COLAB improves turnaround time and system throughput by less than 2% and 3% compared to CFS, while WASH slightly outperforms COLAB (by 2% in turnaround time and 0.2% in system throughput).

6.3 Experiments on HiHope Hikey 970

In this section, we validate the performance and the energy efficiency of COLAB with energy-aware extension under large (*simlarge*) mixed multi-threaded multi-programmed workloads on the real ARM big.LITTLE architecture using a HiHope Hikey 970 development board.

Fairness on large mixed workloads: The result for Unfairness is shown in Figure 11. Although COLAB is not specially designed to achieve perfect fairness, the result shows that the unfairness variance is within 10% compared to ARM GTS. Therefore, it can safely conclude that the unfairness brought by COLAB is acceptable. Besides, Previous work has illustrated that fairness and throughput are largely conflicting optimization goals on AMPs [10]. As a result, in order to achieve higher throughput, some loss for fairness is necessary.

Portable performance on large mixed workloads: Performance variance is significant under the default Linux scheduler (with ARM GTS enable) on the real board compared to checkpoint-based GEM5 simulation. CFS-based ARM GTS scheduler without a core sensitivity aware technique might randomly allocate a high big core speedup thread on either a big core or a little core. While WASH and COLAB could make their intelligent decisions if there is a detected high big core speedup thread during runtime. To validate this portable performance, we report performance from 10 tests on two distinct large workloads using a basic 1B1S configuration and then the give the Geo-mean result.

As shown in the left hand side of Figure 12, H_ANTT results on the Large-1 (radix+ocean_cp) worklaod for ARM

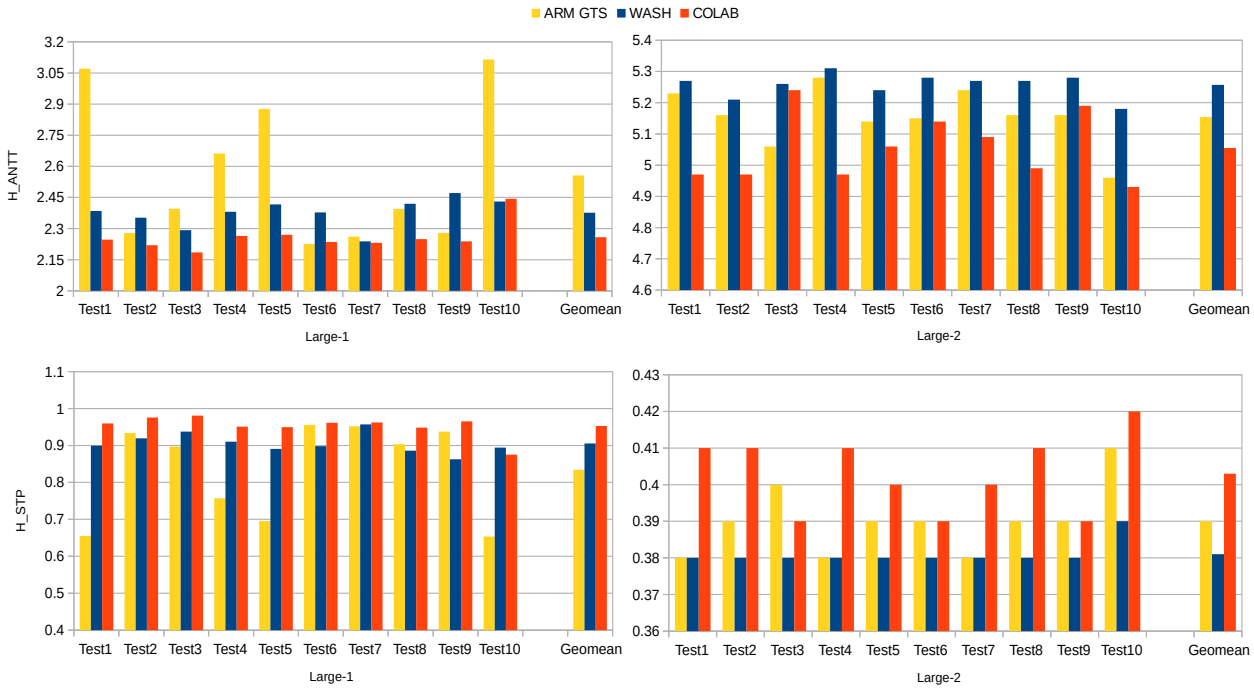


Fig. 12. Performance of the Large-1 (radix+ocean_cp) and the Large-2 (ferret+swaptions) workloads using 1 Cortex-A73 core and 1 Cortex-A53 core. Detailed original results from 10 tests with the same 1b1s configuration are presented to show performance variance of ARM GTS scheduler on the real chip. Lower is better for H_ANTT and higher is better for H_STP.

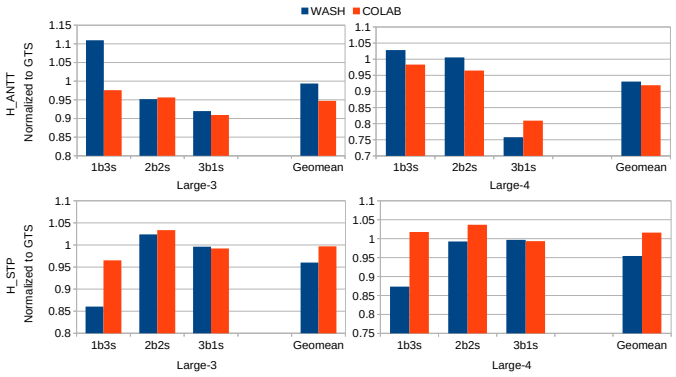


Fig. 13. Performance of Large-3 (blackscholes + radix + fluidanimate + water_spatial) and Large-4 (ocean_cp + ferret + lu_cb + swaptions) Workloads using different 4-core configurations. All results are normalized to the ARM GTS ones. Lower is better for H_ANTT and higher is better for H_STP.

GTS are variant significantly from 2.2 (Test5) to 3.1 (Test10). While for both WASH and COLAB, the H_ANTT results keep between 2.15 and 2.45. Similar with the results for H_STP. The Large-1 is a representative mixed workload compared by a high core sensitivity program, ocean_cp and a low core sensitivity program, radix. The actual speedup between big and little core for ocean_cp is more than 5x as validated in the performance modelling section. So if ARM GTS unfortunately allocates the high big core speedup threads from ocean_cp onto the little (Test1, Test4, Test5 and Test10) during execution, COLAB will result in a up-to 27% (Test10) performance gain compared to ARM GTS. In average, COLAB results in 12% and 14% performance gain on H_ANTT and H_STP compared to ARM GTS, respectively. WASH can also make kinds of intelligent decisions on keeping the high big core speedup threads from ocean_cp on the big core. But it will also schedule bottleneck threads from radix, which do not have a good speedup, to occupy the limited big core resources. As a result, COLAB can outperform WASH with an average 5% performance gain in this workload.

While the main problem of WASH appears when the workload is not core sensitivity as shown in the right hand side of Figre 12. Large-2 workload is composed by ferret and swaptions, application threads from both of them don't have significant speedup between big cores and little cores. While, ferret is a synchronization-intensive parallel program which means there will be bottleneck threads during runtime. WASH simply schedules these bottleneck threads to accumulate runqueue of the big core, which can not actually achieve acceleration but make additional system overhead. COLAB shows its unique advantage in this case by in-place accelerate the bottleneck threads on local cores. As a result, COLAB achieves a up-to 6% (2% in average) performance gain while WASH suffers a up-to 4.5% (2% in average) slowdown on H_ANTT compared to ARM GTS. The difference is larger for H_STP, where COLAB achieves a up-to 8% (3% in average) performance gain while WASH suffers a up-to 5% (2% in average) slowdown compared to ARM GTS.

To further validate the performance of COLAB on general cases, we test two larger workloads (Large-3, Large-4) each composed by 4 programs using distinct 4-core configurations (1B3S, 2B2S and 3B1S). The results are shown in Figure 13, where each bar is the average value of multiple

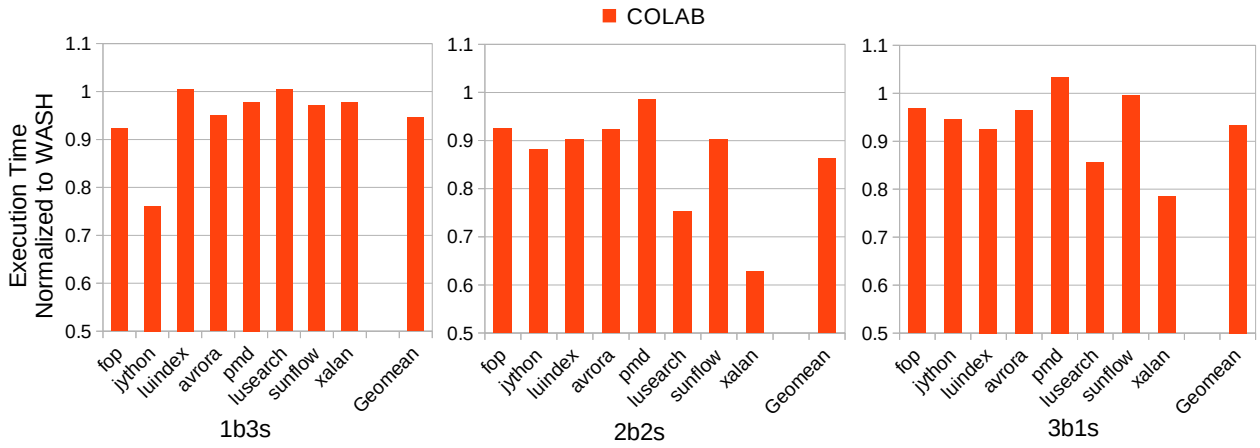


Fig. 14. Performance of Java Workloads(Dacapo: fop, jython, luindex, avrora, pmd, lusearch, sunflow, xalan) using different 4-core configurations. All results are normalized to the ARM GTS ones. Lower is better.

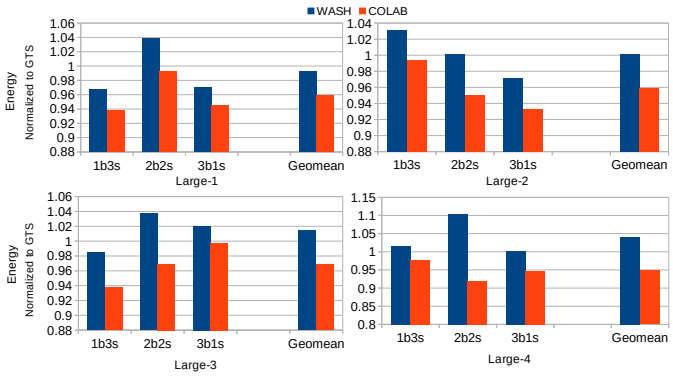


Fig. 15. Energy consumption of large multi-programmed Workloads using different 4-core configurations. All results are normalized to the ARM GTS ones. Lower is better for energy.

tests by a certain configuration. Similar with the results from GEM5 simulation, COLAB shows its best advantage against WASH on limited big core resource (1B3S). When there is only 1 big core, the bottleneck in-place acceleration technique of COLAB make its intelligent decisions and results in a up-to 12% (Large-3) performance gain on H_ANTT compared with WASH. When there are more big core resources, both WASH and COLAB show more advantage as there will be more opportunities to accelerate the needed threads on big cores under core sensitivity aware solutions than CFS-based ARM GTS. For example, WASH and COLAB achieve 24% and 20% performance gain on H_ANTT when running Large-4 on 3B1S configuration compared to ARM GTS. WASH even do better than COLAB as the amount of big cores is sufficient to accelerate both the high speedup and bottleneck threads for the given workload. In average, COLAB achieves 5%-9% and 3%-6% performance gain on H_ANTT on the large workloads compared to ARM GTS and WASH, respectively. WASH can not outperform ARM GTS on H_STP in average based on the problematic scheduling decisions on the 1B3S configuration, while COLAB can still keep good performance.

Portable performance on Java workloads: The experiment in WASH [13] uses Java benchmarks taken from Dacapo [4] [5]. Therefore, we design a similar experiment to shows the performance results for COLAB. Dacpo benchmark suit contains many multi-thread benchmarks. However, some used in WASH original experiments are not available in the current Dacpo edition. So we only compare against the remaining benchmarks: fop, jython, luindex, avrora, pmd, lusearch, sunflow, xalan. The metric is the execution time reported by the benchmark and is normalized to WASH. The results are shown in Figure 14. It shows COLAB outperforms WASH on all core configurations with 10%(up-to 15%).

Energy efficiency on large mixed workloads: Figure 15 shows the performance energy consumption results for large workloads scheduled by COLAB extension and WASH against ARM GTS. In Figure 15, WASH shows up-to 5%(Large-4) energy cost. When allocating tasks, WASH simply schedules bottleneck threads to the big core, regardless of how much energy they consume. An application might cost more energy when running in the big core. For ARM GTS, it cannot be aware of energy consumption of tasks. So its result is not very stable and in some cases (Large-1 2B2S and Large-2 1B3S), ARM GTS can outperform WASH with nearly 2%. However, the energy-aware COLAB scheduler takes the advantages of its energy model, uses the predicted energy and allocates each application to proper cores to decrease the consumption. For example, with 1B3S configure, WASH tries to allocate tasks to the big core, which makes the full use of it. But it cost more energy because many tasks are pushed into the run queue of big core. For COLAB, it makes more intelligent decision by considering the energy consumption and schedule those energy-intensive task to little cores in order to reduce the energy consumption.

As shown in Figure-15, COLAB outperforms ARM GTS and WASH with an average 5% (up-to 8%) energy saving in all the large mixed multi-programmed workloads on distinct 4-core based configurations.

7 Conclusion

We presented the novel **COLAB** scheduling framework that targets multi-threaded multiprogrammed workloads on asymmetric multicore processors (AMPs) which occupy a

significant part of the processor market today, especially in embedded systems. COLAB is the first general-purpose scheduler that, by making *collaborative* decisions on core sensitivity, thread criticallity and scheduling fairness, optimises all these three factors that affect the AMP scheduling - core affinity, thread criticality, and scheduling fairness.

We have demonstrated on a number of different workloads comprised of benchmarks taken from the state-of-the-art parallel benchmark suites PARSEC3.0 and SPLASH-2, simulating a number of different AMP configurations using the well-known GEM5 simulator and then testing on a ARMv8-based HiHope Hikey 970 development board, that the COLAB scheduler outperforms state-of-the-art WASH, ARM GTS and Linux CFS schedulers by up to 21%, 20% and 25%, respectively, in terms of turnaround time (5%, 9% and 11% on the average). We also demonstrate improvements of 6%, 2% and 15% in terms of system throughput on the average. Finally, we show that COLAB achieves an average 5% energy saving compared to both WASH and ARM GTS. This demonstrates the applicability of our approach in realistic scenarios, allowing better execution times and energy efficiency for parallel workloads on AMP processors without additional effort from the programmer.

This work is extended from the previous work published in CGO 2020 [34].

ACKNOWLEDGMENTS

This work is supported in part by the China Postdoctoral Science Foundation (Grant No. 2020TQ0169), the ShuiMu Tsinghua Scholar fellowship (2019SM131), National Key R&D Program of China (2020AAA0105200), National Natural Science Foundation of China (U20A20226), Beijing Natural Science Foundation (4202031), Beijing Academy of Artificial IntelligenceBAAI), the UK EPSRC grants *Discovery: Pattern Discovery and Program Shaping for Manycore Systems* (EP/P020631/1). This work is also supported by the Royal Academy of Engineering under the Research Fellowship scheme. Jidong Zhai is the corresponding author of this paper (Email:zhaijidong@tsinghua.edu.cn)

REFERENCES

[1] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers (CF)*. ACM, 2006.

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

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, 2006. http://www.dacapobench.org.

[6] 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.

[7] Kallia Chronaki, Alejandro Rico, Marc Casas, Miquel Moretó, Rosa M Badia, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(7):2074–2087, 2017.

[8] Kristof Du Bois, Stijn Eyerman, Jennifer B Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[9] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[10] Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto-Matias. Contention-aware fair scheduling for asymmetric single-isa multicore systems. *IEEE Transactions on Computers*, 67(12):1703–1719, 2018.

[11] Jian-Jun Han, Xin Tao, Dakai Zhu, Hakan Aydin, Zili Shao, and Laurence T Yang. Multicore mixed-criticality systems: Partitioned scheduling and utilization bound. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(1):21–34, 2018.

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

[13] 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.

[14] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[15] 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.

[16] Changdae Kim and Jaehyuk Huh. Fairness-oriented os scheduling support for multicore systems. In *Proceedings of the 2016 ACM International Conference on Supercomputing (ICS)*, 2016.

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

[18] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[19] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, November 2005.

[20] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[21] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[22] Tong Li, Dan Baumberger, David A Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Supercomputing, 2007. (SC). Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 2007.

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

[24] Ingo Molnar. Cfs scheduler. In *Linux*, volume 2, page 36, 2007.

[25] 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.