COLAB: A Collaborative Multi-factor Scheduler for Asymmetric Multicore Processors

Teng Yu
yutxie@hotmail.com
University of St Andrews
St Andrews, UK

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk
University of Manchester
Manchester, UK

Vladimir Janjic
vj32@st-andrews.ac.uk
University of St Andrews
St Andrews, UK

Hugh Leather
hleather@ed.ac.uk
University of Edinburgh
Edinburgh, UK

John Thomson
j.thomson@st-andrews.ac.uk
University of St Andrews
St Andrews, UK

Abstract
Increasingly prevalent asymmetric multicore processors (AMP) are necessary for delivering performance in the era of limited power budget and dark silicon. However, the software fails to use them efficiently. OS schedulers, in particular, handle asymmetry only under restricted scenarios. 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 runtime factors affecting AMP for multi-threaded multi-programmed workloads.

This paper introduces the first general purpose asymmetry-aware scheduler for multi-threaded multi-programmed workloads. It estimates the performance of each thread on each type of core and identifies communication patterns and bottleneck threads. The scheduler then 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 the GEM5 simulator on four distinct big.LITTLE configurations and 26 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 depending on the hardware setup.

CCS Concepts • Computer systems organization → Multicore architectures; Real-time operating systems; • Software and its engineering → Runtime environments.

Keywords Asymmetric Multicore Processor, OS Scheduler, Multi-threaded Multi-programmed Workloads

ACM Reference Format:

1 Introduction
Energy and power constraints are central in designing new processors. Most processors will end up in energy-limited devices, such as smartphones and IoT sensors. The power wall limits how much switching activity we can have on each chip. In such a setting, heterogeneous systems provide energy-efficient processing for different types of workloads [25].

Initial heterogeneous systems combined, usually distinct, devices with different Instruction Set Architectures (ISA) but single-ISA asymmetric multicore processors (AMP) are becoming increasingly popular. AMPs introduce new opportunities and challenges. Since all processors share the same ISA, we do not have to prematurely tie a program’s implementation to a specific type of processor. We can let the OS scheduler make this decision at runtime, based not only on which processor is appropriate for the workload but also based on which processors are under-utilized. On the other hand, this introduces an extra degree of freedom to the already complex scheduling decision space. As a result, efficient AMP scheduling has attracted a lot of attention in the literature [22]. The three main factors influencing the decisions of a general purpose AMP scheduler are:

Core sensitivity: Each core type is designed to handle different kinds of workloads. For example, in ARM big.LITTLE systems big cores 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. To build an efficient AMP scheduler, we need to predict which threads would benefit the most from running on each kind of core.

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

**Fairness:** In multiprogrammed environments, making decisions to accelerate each application in isolation is not enough. Decisions should not only improve the utilization of the system as a whole, but should also not penalize any application disproportionately. Ideally, we need to spread the negative impact of resource sharing equally across all applications, we need fairness. For traditional systems this is easy: just give applications CPU slots of equal time in a round robin manner. AMPs make this simple solution unworkable. The same amount of CPU time results in completely different amounts of work on different processors.

Prior research [7, 8, 10, 13, 27] has explored bottleneck and critical section acceleration, others have examined fairness [20, 21, 29, 30, 33], or core sensitivity [2, 6, 19]. More recent studies [14–16, 24, 28] have improved on previous work by optimizing for multiple factors. Such schedulers are good only for specific kinds of workloads. Only one previous work, WASH [12], can handle general workloads composed of multiple programs, each one single- or multi-threaded, 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 which threads run where, leaving much of the actual decision making to the underlying Linux CFS scheduler.

**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 - \(\alpha\) and \(\beta\) that have two threads, and \(\gamma\) that is single threaded. The first thread of each application, \(\alpha_1\) and \(\beta_1\), blocks the second thread of their application, \(\alpha_2\) and \(\beta_2\), respectively. \(\alpha_1\) and \(\gamma\) enjoy a high speedup when executed on \(P_b\). WASH [12], 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, \(\gamma\) and \(\alpha_1\), to \(P_b\), while we map the other bottleneck thread, \(\beta_1\), to \(P_l\). This will not impact the overall performance of \(\beta\). The thread selector knows \(\beta_1\) is a bottleneck thread and executes it immediately. So, what we lose in execution speed for \(\beta_1\), we gain in not having to wait for CPU time. Similarly, this coordinated policy guarantees that \(\alpha_1\) will be given priority over \(\gamma\).

In this paper, we introduce COLAB, an OS scheduling policy for asymmetric multicore processors that can make such coordinated decisions. Our scheduler uses three collaborating heuristics to drive decisions about core allocation, thread selection, and thread preemption. Each heuristic optimizes primarily one of the factors affecting scheduling quality: core sensitivity, thread criticality, and fairness respectively. Working together, these multi-factor heuristics result in much better scheduling decisions. We integrated COLAB in the Linux scheduler module, replacing the default CFS policy for all application threads. The main contributions of our work are: (1) The first AMP-aware OS scheduler targeting general multi-threaded multiprogrammed workloads. (2) A set of collaborative heuristics for prioritizing thread based on core sensitivity, thread criticality, and fairness respectively. (3) Up to 25% and 21% lower turnaround time, 11% and 5% on average, compared to the Linux CFS and WASH scheduler.

**2 Background and Related Work**

Initially described by Kumar et al. [17, 19], single-ISA heterogeneous multicore processors allow for more efficient processing, but to realize this we need the OS scheduler to match threads with cores more suited to their requirements. A straightforward way to determine good matches is based on the IPC of the application on each kind of core. While
easy to understand and perform, IPC is a reliable metric of performance only for single threaded applications, the evaluation of all possible mappings might be lengthy, and it will be affected by resource sharing and phase changes.

To work around some of these problems, other approaches have used performance models to predict the speedup due to executing a thread on another core type. Saez et al. [24] build such a model based on ILP and LLC miss rates, Craeynest et al. [29] used CPI stack, ILP, and MLP, while Jibaja et al. [12] applied Principal Component Analysis to select the performance counters most closely correlated with performance and built a linear model out of them. In all cases, the predicted speedup is used to decide Core Sensitivity, how sensitive the thread’s performance is on the type of core used. The more sensitive threads are assigned to high performance cores exclusively, the rest can be assigned to any type of core.

Acceleration of bottleneck and critical threads is also necessary for high performance on AMPs. Kumar et al. [18] early on identified the benefit of executing Amdahl’s serial bottlenecks on high performance cores, while executing parallel code in low performance, low power cores. Suleman et al. [27] proposed accelerating critical code sections too, in order to minimize the time a thread works on shared data and keep such data on the big core caches. Joao et al. [13, 14] generalized this idea by identifying and accelerating bottleneck functions dynamically. Using programmer hints and hardware support, they measure the number of cycles spent by each thread waiting on data from a (potential) bottleneck function. If above a waiting cycle threshold, the function is accelerated. Jibaja et al. [12] proposed finding bottleneck Java threads by measuring waiting time on contended locks.

Maintaining scheduling fairness is an additional challenge introduced by AMPs. Kumar et al. [18] early on identified the benefit of executing Amdahl’s serial bottlenecks on high performance cores, while executing parallel code in low performance, low power cores. Suleman et al. [27] proposed accelerating critical code sections too, in order to minimize the time a thread works on shared data and keep such data on the big core caches. Joao et al. [13, 14] generalized this idea by identifying and accelerating bottleneck functions dynamically. Using programmer hints and hardware support, they measure the number of cycles spent by each thread waiting on data from a (potential) bottleneck function. If above a waiting cycle threshold, the function is accelerated. Jibaja et al. [12] proposed finding bottleneck Java threads by measuring waiting time on contended locks.

Figure 2 shows the relationship between runtime performance factors and the scheduler components that address them. In order to achieve runtime collaboration, both the core allocator and the thread selector share information and account for all measured performance factors, including core
sensitivity, bottleneck acceleration and fairness, as illustrated below:

**Core Allocator**: AMP-aware Core allocators are mainly directed by the core sensitivity factor – migrating a high speedup thread (with a large differential between big and little core execution time) from a little core to execute on a big core will generally provide more benefit than migrating a low speedup thread. However, this heuristic is overly simplistic. Issues are revealed when the bottleneck factor is considered simultaneously on multiprogrammed workloads. Previous approaches [12] simply combine the calculation from bottleneck acceleration and predicted speedup together, but this can result in suboptimal scheduling decisions – both locking threads and high speedup threads may be accumulating in the runqueues of big cores as described in the motivating example. More intelligent core allocation decisions can be made by avoiding a simple combination of bottleneck acceleration and speedup – the overall schedule can benefit from a more collaborative execution environment where big cores focus on high speedup bottleneck threads, and little cores handle other low speedup bottlenecked threads without additional migration. Furthermore, core allocators attempt to achieve relative fairness on AMPs by efficiently sharing heterogeneous hardware and avoiding idle resources as much as possible. Simply mapping ready threads uniformly between different type of cores can not achieve true load-balancing – the number of ready threads prioritized on different type of core is different and thus a hierarchical allocation should be applied to guarantee overall fairness, which avoids the need to frequently migrate threads to empty runqueues.

**Thread Selector**: The thread selector makes the final decisions on which thread will be executed during runtime. It is usually the responsibility of the thread selector to avoid bottlenecking by thread blocking. In a multi-thread multiprogrammed environment, multiple bottleneck threads from different programs may need to be accelerated simultaneously with constrained hardware resources. Instead of simply detecting the bottleneck threads and assigning all of them to big cores, as previous bottleneck acceleration schedulers do [12–14], the thread selector needs to make collaborative decisions – ideally, both big cores and little cores select bottlenecks to run simultaneously. Core sensitivity is usually unimportant to the thread selector – whether a thread can enjoy a high speedup from a big core compared with a little core is unrelated to which runqueue it is on, or came from. Therefore the thread selector should separate thread priority caused by core sensitivity and solely base decisions on bottleneck acceleration. One exception is that when the runqueue of a big core is empty and the thread selector is invoked – the speedup factors from core sensitivity of ready threads should be considered only in this case. Big cores may even preempt the execution of little cores when necessary. The final concern of thread selector concerns fairness. Scaling time slice of threads by updating the time interval of thread

---

**Figure 3. Coordinated Model by Multi-factor Collaboration**

The thread selector has been shown to efficiently guarantee the equal progress [28] in multi-threaded single-program workloads and achieve fairness. Problems occur when targeting multi-threaded multi-programmed workloads. Simply keeping a thread-level equal progress is not enough to guarantee the multi-application level fairness – the thread selector should ensure the whole workload is in equal progress without penalizing any individual application. In fact, multi-bottleneck acceleration by both big and little cores does provide an opportunity for this - the thread selector makes the best attempt to keep fairness on all applications by accelerating bottlenecks from all of them and as soon as possible.

### 3.2 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 high performance and high fairness, when compared to WASH [12]. The flowchart of our model is shown in Figure 3. Collaboration is facilitated by periodically labeling ready threads in two different categories, based on runtime models of speedup prediction and bottleneck identification:

**Labels for Core Allocation**: Threads with high predicted speedup between big and little cores will be labeled as high priority on big cores. Threads with both low predicted speedup and blocking levels – 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 on local thread selection. The same priority will be given on these blocking threads whether the issuing cores are big or little, so the labels of thread selection do not distinguish the type of cores. 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.
We first record all 225 performance counters of the simulated WASH re-implementation when updating thread affinities. We implement our approach on the GEM5 simulator [5], works [12, 24, 28]. To construct the training set, we run all 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 can be handled by either the core allocator or the thread selector or both together. Based on this coordinated model, the core allocator and thread selector handle different priorities queues from the set of ready threads – their decisions are not greedy on a mixed multi-factor ranking like WASH, rather 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 3. 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 predicating 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 section below.

4 Runtime Design and Implementation

We implement our approach on the GEM5 simulator [5], modifying the simulator and constructing interfaces between the Linux kernel v3.16 with the CFS scheduler.

4.1 Runtime Factors Implementations

To implement the runtime multi-factor model, we update the main scheduler function __sched__schedule() of the Linux kernel by adding a thread labeling process as described in section 3.2 above. A similar approach is followed by our WASH re-implementation when updating thread affinities.

**Machine Learning based Speedup Prediction:** Predicting the relative speedup of each thread between different core types is central for any scheduler targeting AMPs. Our prediction uses an offline trained speedup model to estimate speedups online. This is a common approach in previous works [12, 24, 28]. To construct the training set, we run all applications in single-program mode with two symmetric configurations, using either only little cores or only big cores. We first record all 225 performance counters of the simulated big cores and the relative speedup between the two configurations. Since on a real system, we do not have access to all performance counters simultaneously, we apply Principal Component Analysis (PCA) technique [31] to select the six performance counters with the largest effect on speedup modeling. We then normalize all counters to the number of committed instructions and use linear regression to build the final model, shown in Table 2.

**Bottleneck Identification:** On modern Linux systems synchronization primitives are almost always implemented using 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 triggers system calls to forces the thread to sleep or to wake up sleeping threads. This gives us a convenient single point for monitoring blocking patterns between threads. We first add code in futex_wait_queue_me() and futex_lock_pi(), right before the active thread starts waiting on a futex. We record the current time and store it in the task_struct of the thread. We then insert code in wake_futex() and wake_futex_pi(), right before the waiting task is woken up by the thread releasing the futex. There we calculate the length of the waiting period and we accumulate it in the task_struct of the thread releasing the futex. This way we are able to measure the cumulative time each thread has caused other threads to wait. We use this as our metric of thread criticality for the rest of the paper.

**Speedup based Scale-slice Preemption:** Although we implement our scheduler on Linux kernel by fully re-writing both the core allocator and thread selector, the underlining preemption mechanism of Linux is applying the virtual runtime vruntime in CFS with red-black tree data structure - whenever a new task is enquedue, a preemption wake-up function is invoked to check whether the new coming task should preempt the current task by computing the difference in vruntime and comparing with a boundary. To achieve equal-progress on AMPs, threads running on different types of cores should have different time slices instead of trying to achieve complete fairness on time. We update the default preemption function wakeup_preempt_entity() in Linux by constructing an interface to the GEM5 simulator. To ensure relative equal progress, we apply our runtime speedup model to update the vruntime of the task by dividing it by the its speedup value if the triggering core is a big core.

### Table 2. Selected performance counters and Speedup Model

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Description by PCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>fp_regfile_writes</td>
<td># integer regfile writes</td>
</tr>
<tr>
<td>B:</td>
<td>fetch.Branches</td>
<td># branches encountered</td>
</tr>
<tr>
<td>C:</td>
<td>rename.SQFullEvents</td>
<td># SQ-full blocks</td>
</tr>
<tr>
<td>D:</td>
<td>quiesceCycles</td>
<td># interrupt waiting cycles</td>
</tr>
<tr>
<td>E:</td>
<td>dcache.tags.tagsinuse</td>
<td># tags of dcache in use</td>
</tr>
<tr>
<td>F:</td>
<td>fetch.1cacheWaitRetryStallCycles</td>
<td># MSR-full stall cycles</td>
</tr>
<tr>
<td>G:</td>
<td>commit.committedinstats</td>
<td># instructions committed</td>
</tr>
</tbody>
</table>

Linear predictive speedup model

\[
2.6109 + ((0.0018 \times -0.185A) + (0.0259 \times 0.187B) + (0.1047 \times 0.194C) + (-0.023 \times 0.238D) + (0.0492 \times -0.299E) + (-0.1388 \times -0.227F)) / G
\]
Our scheduling algorithm (see Alg. 1) is implemented by over-riding the default task selector pick_next_task_fair() and core allocator select_task_rq_fair() in Linux kernel supported by the runtime factors. In line with standard Linux notation, we use rq and cur to represent runqueue and the current task of a core, respectively. We describe the two main functions followed by a discussion on scheduling overhead:

**Hierarchical Core Allocator**: When a spawned or woken thread is ready to be executed, the core allocator will be invoked to assign this thread to a core’s runqueue. To achieve relative load balancing and consider the influence from the core sensitivity factor, we involve a hierarchical round-robin mechanism rr_allocator(). Indicated by the speedup and blocking labels, threads are allocated to different core groups.

Algorithm 1 Collaborative Multi-factor Scheduler targeting Asymmetric Multicore Processors

```c
_core Allocator_(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 }
```

The overhead of the performance model is small. It is updated only every 10 msec and it requires the evaluation of the linear regression equation for each thread. To maintain per task performance counter information we need to access the hardware performance counters every time we context switch. The cost of doing so is negligible, around 100 cycles on big.LITTLE. The rest of scheduling overhead comes from labeling all threads based on predicted speedup and blocking level every 10 msec. This is similar to the scheduling overhead of WASH and is infrequent enough to not affect us.

## 5 Experimental Evaluation

### 5.1 Experimental Setup

**Experimental Environment**: We 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: 2B2S, 2B4S, 4B2S, 4B4S, where B denotes big cores and S denoted little cores. We chose to use a simulated environment to make it easier to evaluate our approach on multiple different hardware configurations. While we targeted simulated ARM cores, the underlying general procedure and model can be implemented on any real processor as long as they provide enough hardware performance monitor units (PMU). All hardware counters used by our model are supported by the real ARM Cortex-A57/A53 PMU.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sync. Rate</th>
<th>Comm/Comp Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>bodytrack</td>
<td>medium</td>
<td>high</td>
</tr>
<tr>
<td>dedup</td>
<td>medium</td>
<td>high</td>
</tr>
<tr>
<td>ferret</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>very high</td>
<td>low</td>
</tr>
<tr>
<td>freqmine</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>swaptions</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>radix</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>lu_ncb</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>lu_cb</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>ocean_cp</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>water_nsquared</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>water_spatial</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>fmm</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>fft</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>
Workloads: For our workloads we used 15 different benchmarks (Table 3), pulled from PARSEC3.0 [3] and SPLASH2 [32]. To keep the simulation time reasonably short, we use the simsmall inputs. We group the benchmarks based on two criteria: a) synchronization intensity and b) communication vs computation intensity. 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. To explore the general case of scheduling for an AMP system, we also randomly generate 10 workloads with benchmarks from all groups. Table 4 shows the selected workloads. For all of them, the experiment starts from a checkpoint taken after all benchmarks have completed their initialization.

Each individual result represents the average over two simulations with different core orders - either big cores first or little cores first. Even small variations in the initial state of the system can have a significant effect on scheduling decisions and thus performance. For the Linux scheduler in particular, the order of starting benchmarks will decide which benchmarks will be initially assigned to big and little cores. By varying the initial state and measuring average runtimes over multiple simulations, we minimize the effect of randomness on our evaluation.

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 when running alone on a big-only system is $T^B_i$, then:

$$H_{\text{ANTT}} = \frac{1}{n} \sum_{i=1}^{n} \frac{T^M_i}{T^B_i}$$

$$H_{\text{STP}} = \sum_{i=1}^{n} \frac{T^B_i}{T^M_i}$$

When we evaluate a single benchmark on its own, we use the Heterogeneous Normalized Turnaround Time (H_NTT):

$$H_{\text{NNTT}} = \frac{T^M_i}{T^B_i}$$

### Table 4. Multi-programmed Workloads Compositions

<table>
<thead>
<tr>
<th>Workload Composition</th>
<th>Synchronizations</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>water_spatial - fmm</td>
<td>intensive</td>
<td>4</td>
</tr>
<tr>
<td>dedup - fluidanimate</td>
<td>intensive</td>
<td>18</td>
</tr>
<tr>
<td>water_spatial - fmm - fluidanimate - bodytrack</td>
<td>intensive</td>
<td>9</td>
</tr>
<tr>
<td>dedup - ferret - fmm - water_nsquared</td>
<td>intensive</td>
<td>20</td>
</tr>
<tr>
<td>water_spatial - lu_cb</td>
<td>non-intensive</td>
<td>4</td>
</tr>
<tr>
<td>blackscholes - swaptions</td>
<td>non-intensive</td>
<td>16</td>
</tr>
<tr>
<td>water_spatial - lu_cb</td>
<td>non-intensive</td>
<td>8</td>
</tr>
<tr>
<td>water_spatial - lu_cb</td>
<td>non-intensive</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload Composition</th>
<th>Comm/Comp</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>water_nsquared - fmm</td>
<td>Communication-intensive</td>
<td>4</td>
</tr>
<tr>
<td>ferret - dedup</td>
<td>Communication-intensive</td>
<td>16</td>
</tr>
<tr>
<td>water_nsquared - fmm - fluidanimate</td>
<td>Communication-intensive</td>
<td>9</td>
</tr>
<tr>
<td>blackscholes - dedup - ferret - water_nsquared</td>
<td>Communication-intensive</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload Composition</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>lu_cb - dedup</td>
<td>19</td>
</tr>
<tr>
<td>lu_nb - bodytrack</td>
<td>10</td>
</tr>
<tr>
<td>fluidanimate - swaptions</td>
<td>9</td>
</tr>
<tr>
<td>ocean_cp - fft</td>
<td>8</td>
</tr>
<tr>
<td>freqmine - water_nsquared</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload Composition</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>water_spatial - fmm - fluidanimate</td>
<td>21</td>
</tr>
<tr>
<td>fmm - water_spatial - ferret - swaptions</td>
<td>20</td>
</tr>
<tr>
<td>water_spatial - water_nsquared - ferret - freqmine</td>
<td>17</td>
</tr>
<tr>
<td>blackscholes - bodytrack - dedup - fluidanimate</td>
<td>55</td>
</tr>
<tr>
<td>lu_cb - lu_nb - bodytrack - dedup</td>
<td>53</td>
</tr>
</tbody>
</table>
has around 100x more lock-based synchronizations than other PARSEC applications. Our collaborative core allocation and thread selection policy is much better than WASH at prioritizing bottleneck threads. As a result, we reduce turnaround time by 30% compared to Linux and 20% compared to WASH. In some cases, such as bodytrack, lu_nicb, or freqmine, AMP-awareness has little effect on performance. Such benchmarks split work dynamically between threads, which then all have the same core sensitivity and the application adapts automatically to asymmetries in processing speed. AMP-aware policies offer no benefit while introducing overheads, as was also noted in [12]. The pipeline benchmark dedup has five stages to stream the input set. When there are more threads than available cores, both heterogeneous-aware schedulers can not service the excess threads in time, resulting in a certain impact on overall system performance. There is only one case where COLAB performs significantly worse than WASH. For swaptions, we perform as well as the AMP-agnostic Linux scheduler while WASH improves turnaround time by 31%. This is because the bottleneck threads of swaptions are core insensitive while the non-bottleneck threads are core sensitive. This being the ideal case for WASH, it improves turnaround time while we fail to do the same.

On average, WASH and COLAB perform similarly well and improve performance by 12% compared to Linux when handling single program workloads. This is a limited scenario, with no need for fairness and a simple decision space. COLAB was not expected to perform much better than the state-of-the-art, doing as well as it is a positive result.

5.3 Multi-programmed Workloads
The main aim of the COLAB scheduler is to target workloads of multiple multi-threaded programs, which represents the most general case for CPU scheduling. In this section, we evaluate the performance of COLAB in this setting. Overall, our scheduler is able to outperform both the Linux CFS and WASH when there is room for improvement. This is particularly true when we have a limited number of big cores and/or many communication-intensive benchmarks. In such cases, we need to consider at the same time both core affinity and thread bottlenecks. COLAB can do that, while CFS and WASH cannot, leading to significant performance improvements. In the rest of this subsection, we examine the behavior of COLAB under four different hardware configurations for the five different classes of workloads shown in Table 4.

Synchronization-intensive vs Synchronization Non-intensive workloads: The synchronization-intensive group contains workloads where all programs have high synchronization rates. Because of this, we expect them to have a large number of bottleneck threads, so COLAB should be able to schedule them better than CFS and WASH. Conversely, synchronization non-intensive workloads should provide few opportunities for COLAB to improve on CFS and WASH.

5.2 Single-programmed Workloads
Much of the research on AMP scheduling focuses on single-programmed workloads, where fairness and load balancing are not important and the focus is on core sensitivity and bottleneck acceleration. In this section, we examine how COLAB fares under this scenario. Figure 4 shows H_NTT under Linux (blue), WASH (red) and COLAB (violet), for our multi-threaded benchmarks when executed alone on a 2-big-2-little hardware configuration. We do not consider the three SPLASH2 benchmarks fmm, water_nsquared and waterSpatial, since they do not support more than 2 threads with simsmall input size on GEM5 and scheduling them optimally for performance is trivial.

The AMP-agnostic Linux scheduler is inappropriate for most benchmarks. COLAB improves H_NTT by up to 58% and by 12% on average and up to 173% over Linux CFS for ferret, where most computation happens in a pipeline pattern with unbalanced stages. AMP-aware schedulers take advantage of that by scheduling the longest stages, the bottleneck threads, on big cores. As a result, COLAB does only 13% worse than running on a system with four big cores. Compared to WASH, COLAB achieves its best result for fluidanimate. Previous work [4] has shown that fluidanimate

Figure 4. Performance of single program workloads on a 2-big 2-little system. Lower is better.

H_ANTT and H_NTT are 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 [23] and a state-of-the-art realistic scheduler based on WASH [12]. CFS is the default Linux scheduler and it provides fairness while trying to maximize the overall CPU resource utilization. The original WASH was implemented inside a Java VM to control Java thread affinities. In our re-implementation of WASH, we use the same heuristic but we drive it with a core sensitivity model that fits the simulated system and we use it for controlling all application threads.

Single-programmed workloads on the 2-big-2-little configuration
Figure 5. 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.

Figure 5 show 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). The left and right half of each plot contain the results for the synchronization-intensive (Sync) and synchronization non-intensive (N_Sync) workload classes, respectively. The results agree with our expectations. We observe that COLAB improves the turnaround time of Sync workloads by around 15% and 4% on average compared to CFS and WASH, respectively. We also see that hardware configurations with low core counts, such as 2B2S, favor COLAB. We reduce turnaround time by up to 20% over CFS and by up to 16% over WASH. With fewer cores, the pressure from co-executed applications rises and properly balancing bottleneck acceleration and core sensitivity across multiple programs becomes increasingly difficult. WASH places all bottleneck threads onto the big cores, which results in these threads having to wait for CPU time in busy run queues, ending up with only 3% of performance improvement over Linux. COLAB handles these bottleneck threads in a more holistic way, improving turnaround time by 20% and system throughput by 27%, compared to Linux.

As for N_Sync workloads, there are few bottleneck threads to be accelerated, making scheduling decisions much easier. As a result, both COLAB and WASH perform similarly to Linux, with COLAB improving average turnaround time by 6% and average system throughput by 12% compared to Linux. An interesting point is that COLAB does significantly better (10% and 15% improvement on turnaround time) than WASH and Linux for N_Sync workloads on the 4B2S configuration. In this case, where we have sufficient big core resources without enough critical threads, WASH keeps migrating predicted critical threads on big cores even when there is no actual need. However, COLAB will make intelligent decisions by keeping relatively more threads on little cores, which gives more chance for big cores to always execute the limited really critical threads as soon as possible.

Communication-intensive vs Computation-intensive workloads: When handling programs with high communication-to-computation ratios, bottleneck threads are likely to arise and accelerating them is critical. This is an ideal scenario for COLAB. On the other hand, workloads with little communication are easier to schedule, so CFS and WASH should do reasonably well, leaving little space for improvement.

Figure 6 shows the evaluation results for these two classes of workloads, Comm and Comp. Both COLAB and WASH improve over the Linux scheduler for communication-intensive workloads. They, however, 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, 2B4S). COLAB improves the turnaround time by up to 21% compared to Linux and 15% compared to WASH on the 2B4S configuration. When more big cores are available, WASH does better as it keeps all bottleneck threads on big cores. On these configurations, WASH improves turnaround time by up to 18% over Linux (on the 4B4S configuration) and up to 10% over COLAB (on the 4B2S configuration). On average, COLAB reduces turnaround time by around 12% compared to Linux and 1% compared to WASH for the communication-intensive workload class. Figure 6 also confirms that there are few opportunities for better scheduling with computation-intensive workloads. Still, COLAB does better than WASH and Linux. Its turnaround time and system throughput are improved by around 10% and 15%, respectively, compared to Linux and 5% compared to WASH. This is, again, due to a fact that multiple bottlenecks are distributed both to big and little cores, which results in more efficient use of the available hardware resources for the few bottlenecks that are present.

**Mixed workloads:** This class of workloads represents the general case of different applications with different needs, affinities, and communication patterns competing for the same cores. Figure 7 shows the results for 10 such workloads. COLAB performs very well for these workloads: more diverse programs mean more asymmetry, more bottlenecks, more critical threads, and more potential for acceleration. Our collaborative multi-factor scheduler carefully balancing all scheduling aims (core sensitivity, thread criticality and fairness) leads to a significant performance gain against WASH and Linux. COLAB improves turnaround time and system throughput by around 12% and 11% compared to Linux and around 8% and 7% compared to WASH.

**Thread and program count:** To examine the impact of thread and program count on the behavior of each scheduler, we grouped our experimental results based on these two properties. Figure 8 shows the performance of all schedulers both for workloads with a low thread count (less than the core count for that hardware configuration) and for workloads with a high thread count (at least double higher than

**Figure 7.** Performance of 2- and 4-programmed 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.** 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.** Performance of 2- and 4-programmed Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.
the maximum core count). We observe that both COLAB and WASH perform significantly better than Linux for workloads with a low number of threads. 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 does much better than WASH since it distributes bottleneck threads on all available cores, avoiding overloading the few big cores and keeping the little cores idle. COLAB outperforms Linux by up to 25% (2B4S) and WASH by up to 21% (2B4S) on turnaround time. On average, COLAB improves turnaround time and system throughput by around 20% and 35% compared to Linux and around 8% and 11% compared to WASH for workloads with a low number of threads. For workloads with a high thread count, neither Linux nor WASH are able to improve much on Linux. Overloading the system with threads means that, regardless of where we place threads, cores will have long runqueues. In this case, all cores will have long run queues and COLAB and WASH increase the management overhead (including more frequent thread migrations) with little benefit, leading to performance degradation. Of the two heterogeneity-aware schedulers, COLAB, with its scale-slice technique, more frequently migrates threads, which 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 Linux, while WASH slightly outperforms COLAB by 2% on turnaround time and 0.2% on system throughput.

We see a similar picture when we considered workloads with different number of programs in them. Figure 9 shows the performance of all schedulers for 2-programmed and 4-programmed workloads. As in the case of high and low thread counts, increasing the number of co-executed programs gives higher pressure on the scheduler, increasing the waiting time of threads in runqueues and reducing the direct benefit of migration between waiting threads. But more programs also cause more bottleneck threads and provide new opportunities for co-acceleration instead of only increasing data-parallel threads. By intelligently distributing bottleneck threads from different programs between big and little cores, COLAB faces less problems than WASH from the pressure of increasing programs.

As a result, both COLAB and WASH outperform Linux by more than 10% on 2-programmed workloads on turnaround time and COLAB can keep the 10% performance gain also on 4-programmed workloads, while WASH reduced to only have 5% performance gain on 4-programmed workloads. As for system throughput, COLAB improves by 23% and 12% on 2-programmed and 4-programmed workloads compared to Linux while improves by 5% and 6% on 2-programmed and 4-programmed workloads compared to WASH.

**Summary of Experiments:** Our experiments showed that the state-of-the-art heterogeneous-aware WASH scheduler struggles to make better scheduling decisions that the Linux schedules for synchronization-intensive workloads, computation-intensive workloads, low threads number workloads, high program number workloads, mixed multi-class workloads and limited big cores configurations. Trying to handle both core sensitivity, bottleneck acceleration and fairness through thread affinity alone may lead to too many threads assigned to big cores. Instead, we assign on big cores only threads which run significantly faster on them and we prioritize running bottleneck threads regardless of their thread affinity. This leads to improved turnaround time, higher throughput, and better use of the processor resources compared to both Linux and WASH. In summary from all 312 experiments, COLAB improves turnaround time and system throughput by 11% and 15% compared to Linux and by 5% and 6% compared to WASH.

6 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 criticality 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 PARSEC 3.0 and SPLASH-2, simulating a number of different AMP configurations using the well-known GEM5 simulator, that the COLAB scheduler outperforms state-of-the-art WASH and Linux CFS schedulers by up to 21% and 25%, respectively, in terms of turnaround time (5% and 11% on the average). We also demonstrate improvements of 6% and 15% in terms of system throughput on the average. This demonstrates the applicability of our approach in realistic scenarios, allowing better execution times for parallel workloads on AMP processors without additional effort from the programmer.

**Acknowledgments**

This work was partially funded by the UK EPSRC grants Discovery: Pattern Discovery and Program Shaping for Many-core Systems (EP/P020631/1) and ABC: Adaptive Brokerage for Cloud (EP/R010528/1). This work was also supported by the Royal Academy of Engineering under the Research Fellowship scheme.

**References**


3rd conference on Computing frontiers (CF). ACM.


