Asymmetric Multicore Processor Scheduling

Gavin Austin, Nicholas Rust, Ren Wall

12 December 2019

1 Introduction

1.1 Why We Chose Asymmetric Multicore Processors

Asymmetric multiprocessors, that is CPUs with multiple cores of different power or capability, are becoming popular in many mobile devices with the continued development of ARM big.LITTLE. It can even be argued that modern high end desktop and server oriented processors with many cores fit this definition since often one or a small subset of the cores run at a higher clock speed than the rest. Current implementations of asymmetric aware scheduling are extremely focused on power management and targeted at mobile devices.

We wanted to better understand the existing scheduling algorithms for these processors as well as where asymmetric aware scheduling could go in the future. Since both types of processor despite having very different target applications share the same inherent tradeoffs with regards to scheduling we can learn about both by examining where these algroithms lie on the tradeoff space.

2 Optimization Targets

2.1 Power vs. Performance

The two main targets to be considered when determining a scheduling policy are power and performance. These also represent the primary tradeoff in scheduling on asymmetric multiprocessors. There can be two extremes regarding these targets: 100% performance and 0% power efficiency, or 0% performance and 100% power efficiency. If a scheduling policy is executing to achieve maximum power efficiency, tasks will execute sequentially on the most power-efficient core. This policy will result in the longest runtime of all possible options. If a scheduling policy is executing to achieve maximum performance, tasks will get allocated on the fastest core first with other tasks placed on the slower cores if possible. This is the fastest naïve policy, and will use the most power and energy. [1]

Ideally, the scheduling policy should lie somewhere in the middle between these two extremes, at least for mobile devices [1]. This tradeoff can is visualised in figure 1. Note that power is measured in bogowatts and time is measured in seconds.

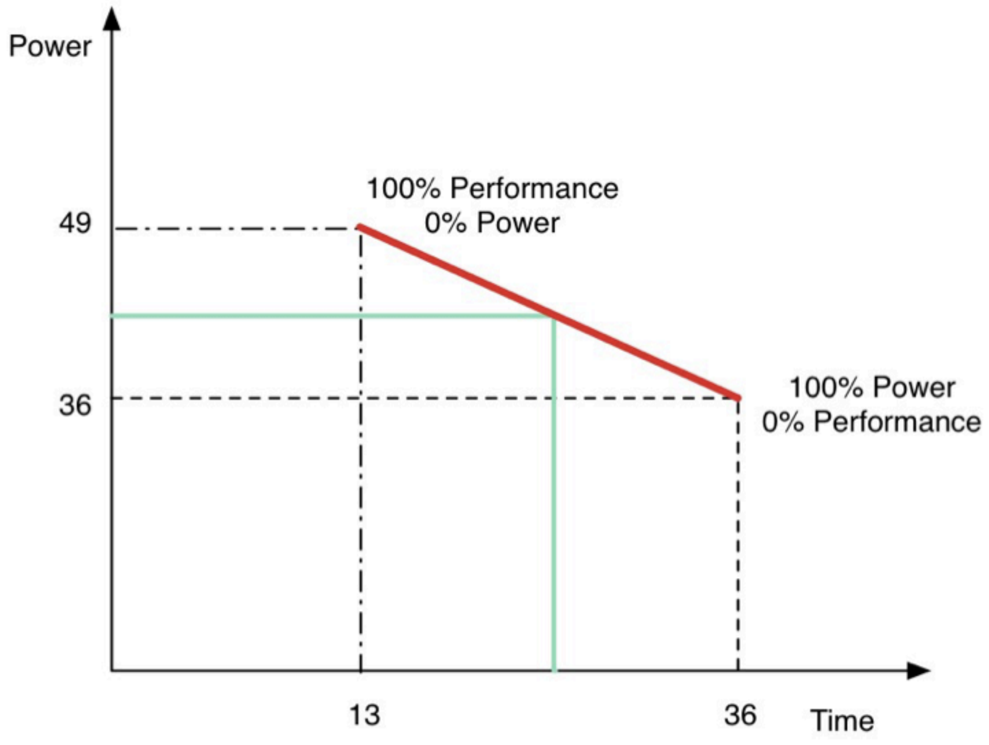

Figure 1: Power vs. Performance [1]

3 Evolution of Asymmetric Scheduling 2015

3.1 Previous Scheduler (Before 2015)

Linux previously had three parts that all worked side-by-side in order to try to improve energy efficiency without compromising throughput.
The three parts are as follows:

1. Linux scheduler (Completely Fair Scheduler - CFS)
2. Linux cpuidle
3. Linux cpufreq

3.1.1 CFS

The Linux Completely Fair Scheduler is very good at optimizing throughput by throwing processes onto open cores. However, by default it does not have any knowledge of energy usage or efficiency. When implementing Energy Aware Scheduling, the CFS is given the ability to access and read the power usage values of every process, as well as previous power usage, in addition to every core's idling statistics. [3]

3.1.2 cpu-idle

The Linux cpuidle system attempts to use heuristics to save energy by putting the processor into an idle state without affecting performance. The cpuidle system has absolutely no control over the processes that are put onto specific cores or when, and is only capable of reacting to what the CFS throws at it. [3]

3.1.3 cpu-freq

The cpufreq (cpu frequency) subsystem uses heuristics to throttle up or down the frequency of a core depending on the load in order to save on power - again without affecting performance. This system is also entirely separate from the CFS *and* cpuidle. [3]

3.1.4 Previous Scheduler drawbacks

The first issue we see with this three-part system is that the bottom two systems (cpuidle and cpufreq) are both reactive to what the CFS does, and since the CFS is entirely unaware of power management it frequently breaks their predictions and results in both power efficiency loss *and* speed loss. In addition, the fact that they are reactive means they have to be run frequently themselves in order to keep up with what is happening in the processor in real time which creates more overhead management. [3]

3.2 Current Scheduler (After 2015)

The current asymmetric multicore processor scheduler implementation is known as Energy Aware Scheduler or EAS. As the name implies, it takes into account the energy usage of each process and allocates the process to a core for the duration of the process. It should also be noted that EAS for Linux is being developed jointly by Arm and Linaro, it still has much room for improvement, specifically at full load. [4]

3.2.1 Energy Aware CFS

Under EAS the Completely Fair Scheduler has undergone many changes, it still performs the same function and prioritizes throughput, but now takes into consideration both idle state and frequency. Effectively both cpuidle and cpufreq were reworked and their abilities given to the CFS. [4]

3.2.2 sched-idle

The new system makes the scheduler aware of the idle states of every core and cluster. It prioritizes placing new processes onto the core with the shallowest idle state - that is to say - the one using the most power while idle. In so doing we lose less power savings than waking up one of the other cores which is more deeply in the idle state, and it responds more quickly as well. Therefore being both faster and more energy efficient. [4]

3.2.3 sched-freq

Mainline Linux kernel is capable of storing information about processes and their load times. CPU Operating Performance Points then change almost immediately to accommodate the new process. Whereas before, cpufreq's main downfall was that since it was a run process it would have to wait until it's turn came to be run. Due to this it would often tune the frequency too late, making the process suffer under lower frequency (speed loss) or have the core at too high of a frequency (energy loss). The results were significant inefficiencies. [4]

3.3 Impacts

It is clear that taking energy into account when scheduling is a vast improvement over blind scheduling with heuristics thrown on top, however, there are still issues that can arise from this scheduling process. Namely, the EAS works very well when the cpu load is interspersed, but when the load is frequently capped, and all cores are running at full capacity, I expect the scheduler will be unable to act on its priorities and begin acting as a completely fair scheduler once again. This does have limited impact on most user-side machines, however, there are certainly cases where having an EAS for full capacity would be beneficial. It should be noted that in order to do balance both speed and power consumption, the scheduler may have to hold process back from running for a time, meaning it would no longer be a Completely Fair Scheduler.

4 WASH

While the schedulers we have examined already certainly have large improvements over existing schedulers, we could certainly do better. On asymmetric processors a lot of assumptions that are made about how we schedule multithreaded programs are broken. In particular we can look at the threads of a single program and find threads which have a higher priority to finish quickly and actually do something about them finishing more quickly. The WASH scheduler keeps track of how many other threads are waiting on each thread to release its locks [2].

4.1 Implementation

WASH is implemented as an extension to the JikesVM Java Virtual Machine. This allows the scheduler much further insight into the operation of each thread than a kernal space scheduler has. The drawback of this implementation is that WASH can only set the core priorities of each virtual machine thread, it cannot influence the threads of other processes on the system. If the application that WASH is scheduling is the primary load on the system this is not an issue, but it shows the difficulty in generalizing this solution [2].

4.1.1 Psudocode

Algorithm 1 WASH

```
function WASH($T_A, T_V, C_B, C_S, t$)
    $T_A$: Set of application threads
    $T_V$: Set of VM service threads, $T_A \cap T_V = \emptyset$
    $C_B$: Set of big cores
    $C_S$: Set of small cores, $C_B \cap C_S = \emptyset$
    $t$: Thread to schedule where $t \in T_A \cup T_V$
    if $|T_A| \leq |C_B|$ then
        if $t \in T_A$ then Set Affinity of t to $C_B$
        else Set Affinity of $t$ to $C_B \cup C_S$
        end if
    else if $t \in T_A$ then
        if $\forall \tau \in T_A(\text{Lock\%}(\tau) \leq \text{Lock}_{\text{Thresh}})$ then
            Set Affinity of $t$ to $C_B \cup C_S$
        end if
    end if
end function
```

[2]

4.2 Performance

WASH is not unique in it's maximal utilization of the most powerful core, where it gains its performance advantages is being able to keep more threads running on the less powerful cores at the same time. As can be seen in figure 2; WASH not only has better completion time than competing schedulers, but also better total energy usage. This is a result of it's more effective utilization of all the cores, not just the high power cores.

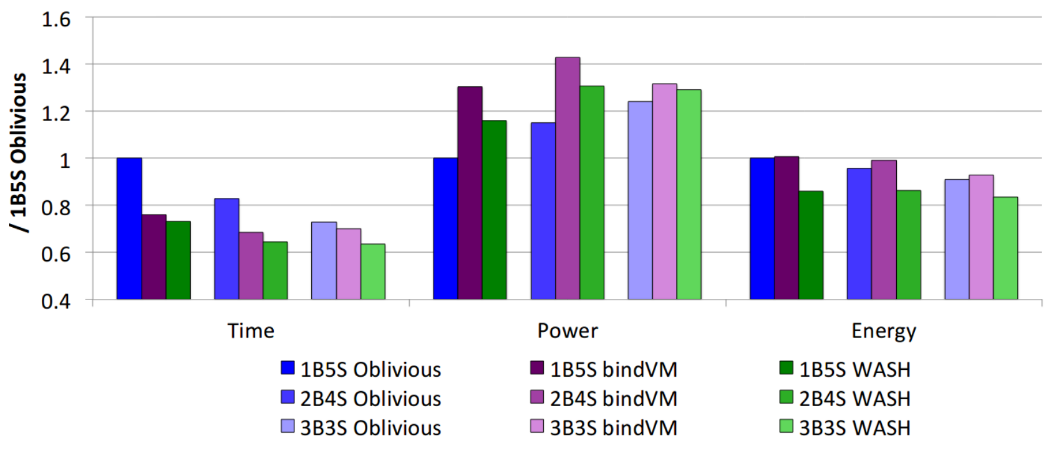

Figure 2: WASH compared to two other schedulers, default CFS (Oblivious), and helper threads manually bound to low power cores (bindVM), on three hardware configurations, 1B5S (1Big core and 5 Small cores), 2B4S and 3B3S [2].

References

[1] Pantelis Antoniou. Adventures in (simulated) asymmetric processing, 2013.

[2] 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 '16, pages 24–35, New York, NY, USA, 2016. ACM.

[3] Amit Kucheria. Energy aware scheduling (eas) project, 2015.

[4] Linaro. Energy aware scheduling (eas) progress update, 2015.