Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach

Abstract

The energy demands of modern mobile devices have driven a trend towards *heterogeneous multi-core systems* which include various types of core tuned for performance or energy efficiency, offering a rich optimization space for software. On such systems, data coherency between cores is automatically ensured by an interconnect between processors. On some chip designs the performance of this interconnect, and by extension of the entire CPU cluster, is highly dependent on the software's memory access characteristics *and* on the set of frequencies of each CPU core. Existing frequency scaling mechanisms in operating systems use a simple load-based heuristic to tune CPU frequencies, and so fail to achieve a holistically good configuration across such diverse clusters. We propose a new adaptive governor to solve this problem, which uses a simple trained hardware model of cache interconnect characteristics, along with real-time hardware monitors, to continually adjust core frequencies to maximize system performance. We evaluate our governor on the Exynos5422 SoC, as used in the Samsung Galaxy S5, across a range of standard benchmarks. This shows that our approach achieves a speedup of up to 40%, and a 70% energy saving, including a 30% speedup in common mobile applications such as video decoding and web browsing.

ACM Reference Format:

. 2020. Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach. In *Proceedings of LCTES*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern embedded platforms, such as mobile and tablet devices, have become a ubiquitous part of the modern computing ecosystem. Their battery-powered design has driven a new wave of hardware research, including the *asymmetric multiprocessor* (AMP). This is a System-on-Chip which offers one cluster of CPU cores designed to be energy efficient, and another designed to offer high performance. This concept has been implemented in the ARM *big.LITTLE* architecture which is widely adopted in mobile platforms, including the Samsung Galaxy range. The big.LITTLE design has an energy efficient processor (named LITTLE) with a performant but more power-hungry processor (named big), where each processor also offers a range of frequency settings. This design exposes a large optimization space for software to trade performance against reduced energy consumption by choosing a processor depending on energy and time constraints.

In order to simplify software development for the platform, the hardware offers transparent data coherency between its processor clusters. On many big.LITTLE platforms, this is implemented via the ARM CoreLink CCI-400 interconnect [7] which uses a bus-snooping protocol: when a data access is issued by a processor, the interconnect will broadcast a message to all processors to check whether the data is present in their local cache before accessing RAM. Because the interconnect communicates with the processor, extra latency can be introduced in this procedure if the processor's clock frequency is low. Our experiments show that `gcc`, for example, can suffer an 80% slowdown due to this mechanism. While newer big.LITTLE platforms include a hardware snoop filter to mitigate these effects, the popular CCI-400 interconnect remains in wide use across the world – a recent study by Facebook reports that **75% of smartphones** using their platform have CPU designs released before 2013 [28], before any big.LITTLE hardware snoop filters were designed.

In this paper we propose a software solution to this problem with a novel `ondemand-anti-snoop` governor, a new DVFS governor which enhances the standard Linux `ondemand` governor to consider the memory traffic between processor units and main memory at a hardware-level. Our approach is highly generalized, working transparently across all software, and requires only a simple, generic, train-once model of real-time system activity to learn snoop effects in a range of scenarios. Our evaluation shows that performance improvements of up to 40% can be achieved with our new dynamic frequency governor on real-world software.

The main contributions of our work are:

- A methodology to characterize snooping latency effects on the bus-snoop protocol interconnect fabric.
- A simple but effective model of snoop latency using a microbenchmark; our model is trained once on this benchmark and then applies generically to all software.
- A new DVFS governor which uses our model together with hardware-level information to mitigate snooping latency in real-time by locating the ideal holistic frequency configurations on the SoC.

The remainder of this paper is organized as follows. Section 2 first presents the hardware architecture in detail on which our study is conducted. Section 3 then presents a model of snooping effects, considering hardware-level data, and a new frequency scaling governor using this model to mitigate snooping latency. In section 5 we evaluate our new governor on a set of real-world workloads. In section 6 we discuss related work, and conclude in section 7.

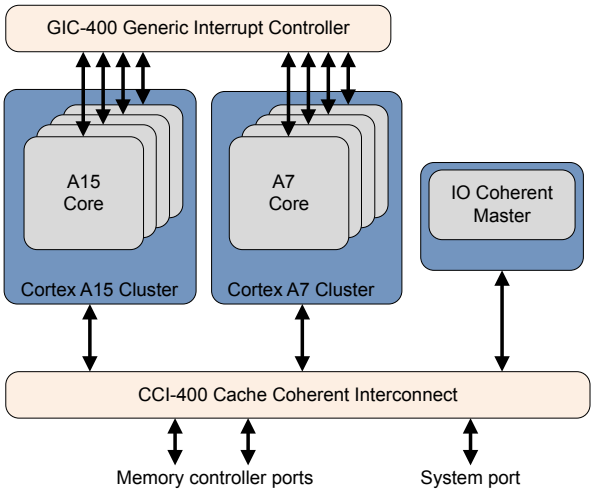


Figure 1. big.LITTLE architecture implementation on our platform. Full data coherency is assured by a bus-snooping protocol. (Courtesy of Arm Limited.)

2 Memory architecture background

In order to understand the relative difference in execution time for different clock frequencies, we first explain in detail how the memory works on the platform we use.

We use an Odroid-XU3 from HardKernel [17], which implements the Exynos 5422 System-on-Chip (SoC) from Samsung [6]. The SoC itself is identical to that used in the Samsung Galaxy S5 smartphone, though the host board is slightly different. Figure 1 shows a block diagram of the SoC architecture, which implements two CPU clusters of 4 cores each, for a total of 8 cores. One cluster targets energy efficiency (named LITTLE) and uses a Cortex-A7 [5], while the other cluster targets performance (named big) and uses a Cortex-A15 [4]. Each CPU core has its own private L1 cache, and shares one L2 cache within the same cluster. Both CPUs have 32 Kb instruction cache and 32 Kb for data in L1 cache. The LITTLE cluster has a unified 512 Kb L2 last level cache, while the big cluster has 2 Mb. Finally, CPU core clock frequency is shared between cores within the same cluster.

In this SoC, data cache coherency between the two clusters is managed by the ARM CoreLink CCI-400 Cache Coherent Interconnect [3]. Using this interconnect fabric, when a CPU core in a cluster performs a read/write memory operation for data that is not contained in its internal cache, the interconnect checks if the data is present in the cache of the other cluster, and if not, it then performs an access to off-chip main memory (RAM). This check is performed by an operation called snooping, for which further specification details can be found in the relevant ARM white paper [26].

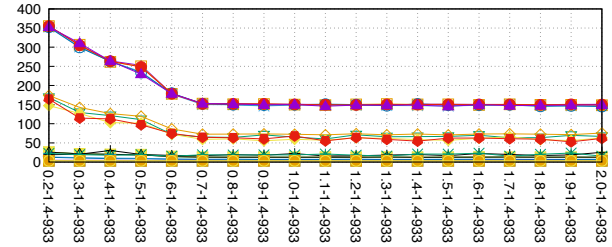
The effect of snooping on this hardware is that extra latency can be introduced when a process performs several memory accesses and when the frequency of the idle cluster is low. This is because the interconnect fabric communicates with the CPU cluster to check its cache status, and the cache status check is performed at a speed relative to the current clock frequency of the cluster. Effectively, therefore, a process running on one CPU cluster can stall on memory accesses because another cluster has a low clock frequency.

This effect interacts with the common dynamic CPU frequency scaling policy (DVFS) used in Linux, and present on the majority of Android smartphones (the popular Energy Aware Scheduler (EAS) for Android uses a frequency governor with a very similar design to that used in standard Linux). DVFS by default attempts to reduce the clock frequency of a CPU whenever it is idle, to correspondingly reduce the amount of energy consumed by that CPU. Whenever one particular CPU cluster on a big.LITTLE chip has a lower workload, therefore, the clock frequency of that cluster is reduced, which can in turn cause snoop-induced stalling on memory accesses across other clusters. Our benchmarks show that this stalling can cause a slowdown of up to 80% for the most memory-intensive applications – which is a very significant impact for the wide range of smartphone models using this hardware architecture.

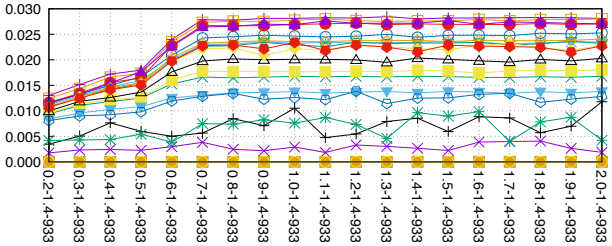
Obvious solutions to this problem are (i) to always run all CPUs at their highest clock frequency, or (ii) to try to completely power down CPUs that aren't being heavily used (as powering down a CPU cluster also removes snooping effects). Both approaches are problematic: running CPU clusters at high frequencies incurs significant energy penalties and so will reduce battery lifetime, while powering down a CPU cluster takes a significant amount of time to migrate tasks and deactivate cluster-bound kernel services (our measurements with `hotplug` show that Linux takes 300ms to power down a low-workload cluster). A third approach for an idle CPU with no workload is to rely on periodically putting that CPU into sleep mode via the `cpuidle` framework (rather than forcing a full power down), and periodically waking the CPU to accommodate kernel maintenance routines. However, during all of the periods in which the CPU is awake this incurs snooping latency.

Neither of these approaches are therefore attractive by themselves. Instead, we develop a trained model of the hardware's snoop behavior and its interaction with clock frequencies; this model is trained just once on a memory-based microbenchmark to discover how the snoop architecture behaves. Using this trained model we develop a new dynamic frequency governor which monitors the system in real-time to continually find a balance between raising clock frequencies to avoid snoop-induced stalling when needed while still keeping them as low as possible to conserve energy. Our governor is most effective when multiple clusters are awake with at least some workload, but is also complementary to periodic CPU sleep protocols used on completely idle clusters during the times when that CPU wakes for maintenance procedures.

(a) CPU cycles spent to perform a memory access.



(b) CPU PMCs `memory_bus_access` for the process.



(c) CCI PMCs `stall cycles per read` on the big port.

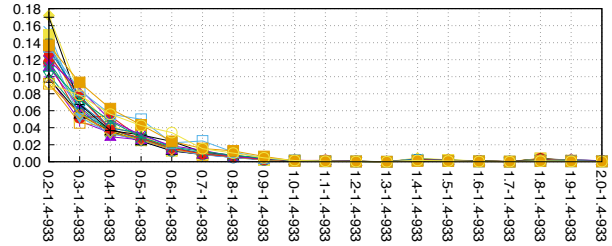
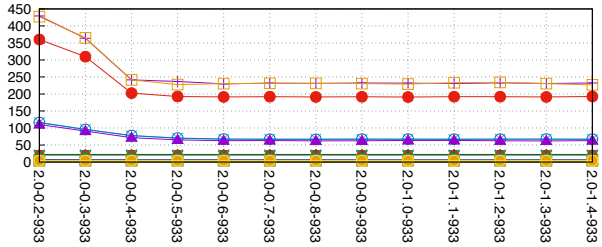
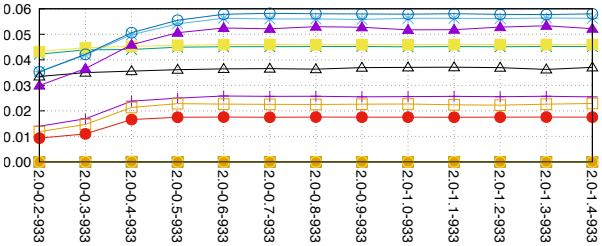


Figure 2. Benchmark on the LITTLE cluster, big cluster idle

(a) CPU cycles spent to perform a memory access.



(b) CPU PMCs `memory_bus_access` for the process.



(c) CCI PMC `stall cycles per read` on the LITTLE port.

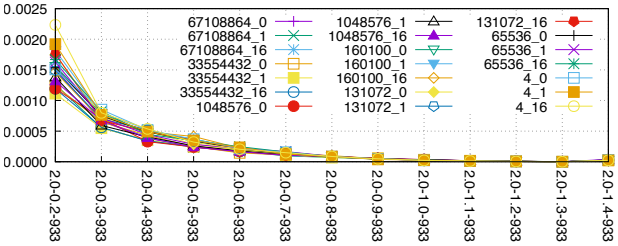


Figure 3. Benchmark on the big cluster, LITTLE cluster idle

3 Modeling snooping latency

Our first step is to develop a detailed model of the conditions under which snooping latency occurs, and to understand which hardware monitors we can measure to enable detection of this effect in real-time and correlate it with our model. We do this using microbenchmarks that target a range of different memory access patterns to study how they cause associated memory characteristics in the hardware.

The following sub-sections first present a study of snoop latency effects in general using our benchmark, then discuss how we can detect these effects in real-time using performance monitoring counters on both the CPU and CCI.

3.1 Memory latency exploration

To study snooping latency on memory operation, we use the `ccbench:cache` [11] microbenchmark. This was initially written to discover a CPU's internal memory hierarchy, and runs a pointer chasing loop over an array of a given size. Values in the array represent the next index in this same array to follow for the next iteration. The benchmark can be configured to access the array using a range of different patterns, including a linear access of a unit stride, a stride of a cache line size, or a random pattern which prevents smart CPU memory prefetchers from being effective.

We run this microbenchmark on one active cluster, keeping the other cluster idle, over each possible set of cluster frequencies and with different parameterization to cover a large range of use cases. The parameterization comprises all three memory pattern accesses, and multiple different array sizes. The array size itself is chosen to either fit, or not fit, in the different CPU internal cache sizes of both clusters, the latter case forcing off-chip memory access.

Figure 2 presents the overall results of running the benchmark on the LITTLE cluster and keeping the big cluster idle,

while Figure 3 presents the reverse scenario. Each graph shows all three memory access patterns with different array sizes. The graph legend shows the size (in number of index in the array) and pattern access for each series – where an access pattern 0 is the random pattern, 1 a strictly sequential pattern, and 16 is a sequential pattern with a stride of the size of the last level cache line of the CPU (both CPU clusters use a 64-byte cache line and array indexing via 4-byte integers).

We begin by measuring the overall total number of CPU clock cycles spent performing a single memory access in the benchmark (which reads the index of the next memory access in the array); the results of this are shown in Figure 2a and Figure 3a for the benchmarks running on the LITTLE and big cluster respectively. The x-axes of all graphs show the CPU and main memory frequency configuration and are present at the bottom figures, in the format {big_freq}-{LITTLE_freq}-{mem_freq}[1], and the y axes show CPU cycles. Figure 2a and Figure 3a show that there is an increase in the number of cycles for any benchmark with an array of more than 131,072 elements[2], or when the memory access pattern is not strictly sequential. In both of these cases we see additional increases in cycles when the idle cluster has its frequency set below a certain level; in Figure 2a this occurs when the idle big cluster drops below 0.7Ghz, while in Figure 3a it occurs when the idle LITTLE cluster drops below 0.4Ghz.

The graphs in Figure 2b-c and Figure 3b-c then show information from the PMCs that we use to detect snooping latency cases in real-time. These PMC readings are taken across an identical set of frequencies and benchmarks as those used in Figure 2a and Figure 3a. In detail, Figure 2c shows CCI stalling cycles per read request coming from the LITTLE cluster, while Figure 3c shows CCI stalling cycles per read request coming from the big cluster. In both cases we can clearly see that this PMC correlates with the snoop filter effect: when clock frequencies are too low under certain memory access patterns, we get higher stalling cycles recorded on this PMC. While this PMC therefore shows part of the picture, it is only able to report all events from an entire cluster, involving all of its cores, and all of its processes.

It also useful to be able to understand the memory profile of a particular process for which we would like to optimize, to understand if the snoop latency measurements are actually affecting this process. Because we want to avoid any application-specific knowledge or static memory profiling, we use a CPU PMC providing real-time memory bus usage for each core. Figure 2b (and 3b) shows the `mem_bus_access` PMC for the microbenchmark running on the LITTLE and big cluster respectively which offers exactly this information in real-time. Here we see that the number of memory accesses per cycle appears to *reduce* across our benchmark configurations as the CPU frequency of the other cluster is lowered. This decrease is caused by the stalling effect itself, such that a process has an apparently lower amount of memory access as recorded by this PMC if that process is being affected by snoop-based stalling. This effectively reduces the speed with which that process executes and so reduces its apparent memory access volume per cycle.

The combined data from these PMCs indicates that we can detect stalling for a particular process of interest by reading the CCI PMC of its non-resident cluster, and the memory bus access profile CPU PMC of the process on its resident cluster. When the memory access profile exceeds a certain memory size or has a non-sequential pattern and begins to show lower memory accesses per cycle, *and* the CCI PMC of the non-resident cluster indicates stalling, we can assume that the clock frequency of the other cluster must be increased to reduce stalling effects for this particular process.

In the next section we propose a formal way to use these PMCs to dynamically determine when the snooping mechanism could affect application performance.

3.2 Detection of snooping latency

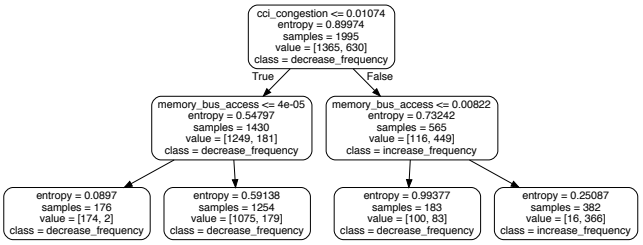
As shown in section 3.1, we see a clear trend of increase in latency for certain combinations of stalling cycles on the CCI PMC and memory bus access via the CPU's PMC. However, these increases show a non-linear relationship between the PMC values (specifically the values reported in Figure 2b and Figure 2c relative to processes running on the LITTLE cluster, and the values in Figure 3b and Figure 3c for processes running on the big cluster). These non-linearities are difficult to capture heuristically to determine the ideal levels to configure the respective CPU frequencies; instead we therefore develop an offline automatic modeling process.

There are two key questions to consider in solving this task: 1) which CPU frequency configuration has performance loss? and 2) what values do the relative PMCs report when there is known snooping latency?

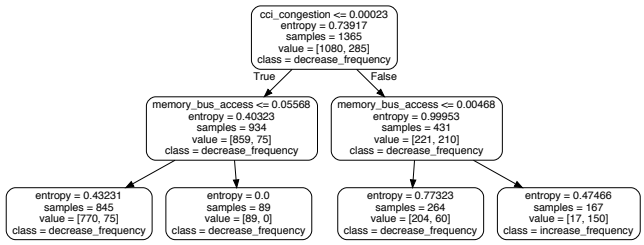
A rigorous test for the first question would be to detect when the performance variation is *statistically significant* and not inherent noise due to the operating system's process management. To determine whether there is a statistically significant variation, we perform a Student's t-test on each execution trace (scanning each CPU configuration) against the highest possible frequency of both clusters; this is a standard statistical method to establish whether or not a difference between two data sets is significant. In our case we assume that there is a problematic snooping latency when the p-value of this statistical test is higher than 0.05.

[1]CPU frequencies are shown in GHz, and memory in MHz. Dynamic adjustment of CPU frequency is available by default on the Linux kernel, and memory bus frequency can be adjusted statically at boot time. Although our experiments here all use the same memory bus frequency of 933MHz, we include the actual values used in graphs for completeness.

[2]This threshold is a result of the L2 cache size on the LITTLE cluster, which is 512KB ($131,072 \times 4bytes$).



(a) Decision tree used to manage frequency of the big cluster.



(b) Decision tree used to manage frequency of the LITTLE cluster.

Figure 4. Decision trees used to find CPU frequencies that limit snooping latency.

Once we have isolated which execution has snooping latency issues, we can answer the second question using a machine learning model. The goal of this step is to find thresholds at which there is problematic snooping latency relative to the hardware-level monitoring points of the CPU PMC `memory_bus_access` and CCI PMC `stall cycles`. Since our objective is to find a way to make a quick decision at runtime to detect and mitigate snooping latency at any given time, we use a decision tree to model the situation. Decision trees are relatively simple models which are trained on a set of example data, for which we use our benchmark examples, to determine which input values (PMC levels, in our case) should imply which output values (increase or decrease clock frequency). Once trained they can be automatically converted to simple C programs of if-else statements for rapid runtime decision-making.

As we manage both cluster frequencies independently, we build two decision trees that consider CCI PMCs of the other cluster. In other words, when we manage the frequency of the LITTLE cluster, we consider `stalling cycles` per `read request` on the channel of the big cluster, and vice versa.

Figure 4a and 4b show the trained decision trees, based on our benchmark data, to consider while managing the frequency of the big and LITTLE cluster respectively. In the trees, the variable *cci_congestion* corresponds to the quotient of CCI PMCs, while *memory_access* corresponds to the sum of processes CPU PMC `memory_bus_access` of applications to optimize for. At runtime, if we detect PMC values over these thresholds, we are in problematic snooping latency territory and may increase the frequency of the other CPU to limit performance loss.

We next present the way in which we use our model at runtime as part of the OS frequency management process.

4 A snoop-aware frequency governor

In this section we present our runtime algorithm for frequency scaling, which uses our decision trees to make almost instant decisions on which clock frequencies to use across both CPU clusters based on the current memory access characteristics of a process of interest, and the stalling behavior of the inter-cluster cache interconnect.

We first present the detail of how the Linux kernel manages CPU frequencies by default, then propose a refined version which leverages runtime hardware-level information to alleviate snooping latency.

4.1 Linux DVFS governor

Algorithm 1: Linux DVFS ondemand governor.

```
1 if load > up_threshold then
2 |   cpu_freq = max_cpu_freq
3 else
4 |   cpu_freq = min_cpu_freq + load *
  |     (max_cpu_freq − min_cpu_freq)/100
5 end
```

In order to limit energy consumption of a device when there is no CPU activity, Linux dynamically adjusts the power and speed setting of all CPUs using Dynamic Voltage Frequency Scaling (DVFS). This is implemented in the logic of a module termed a governor.

The default governor used in many flavors of Linux is the `ondemand` governor [21] which works simply by adjusting the CPU frequency in direct proportion to the current load of the CPU, where the load level is defined as the amount of time for which the CPU is non-idle during the last sampling period. Algorithm 1 shows the main body of this standard governor [3], which is also very similar to that used with the EAS scheduling framework (`schedutil` governor) on Android.

This governor does not rely on any in-depth hardware-level information, basing its decision-making only on process activity levels over time. As such, the governor would choose the lowest frequency for a cluster when it is idle. However, as shown in the previous section, this decision may induce snooping latency combined with certain memory behaviour for a process running on another cluster.

[3]The full source code of the governor can be found at https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/cpufreq/cpufreq_ondemand.c

To avoid this issue on AMP architectures with bus-snoop cache coherency, we design a more advanced governor which uses our trained model with real-time hardware-level data.

4.2 DVFS ondemand-anti-snoop governor

Algorithm 2: Enhanced DVFS `ondemand-anti-snoop` governor.

```
1  if load > up_threshold then
2  |   cpu_freq = max_cpu_freq
3  else
4  |   update_pmc_cci_congestion()
5  |   update_pmc_memory_access()
6  |   if cci_congestion > cci_congestion_threshold
7  |       and memory_access > memory_threshold
   |    then
8  |   |   stall_cpu_freq = cpu_freq + cpu_freq_step
9  |   else
10 |   |   stall_cpu_freq = cpu_freq − cpu_freq_step
11 |   end
12 |   cpu_freq = min_cpu_freq + load ∗
   |    (max_cpu_freq − min_cpu_freq)/100
13 |   cpu_freq =
   |    clamp(max(cpu_freq, stall_cpu_freq))
14 end
```

Our approach to avoiding snooping latency is based on progressively finding the right frequency where we do not detect any latency caused by the snooping mechanism. Using our model presented in section 3, we refine the `ondemand` governor by integrating information from both CCI PMCs and CPU PMCs of processes to avoid snooping latency when our model indicates that it is occurring. Algorithm 2 shows our enhanced `ondemand-anti-snoop` governor, which integrates our decision tree compiled out to C code as if-statements.

This enhanced DVFS governor works as follows. We first update the relevant PMC metrics by reading from hardware (line 4 and 5), then consult our trained decision tree (compiled out to C code, on lines 6 and 7) to determine whether our model suggests increasing the CPU frequency or reducing it for a given cluster. Following this, we calculate both (a) the suggested CPU frequency based on the current system load, and (b) the recommended frequency based on our model, and set the actual frequency to whichever of (a) and (b) is higher (line 13)[4]. This comparison is necessary because the load-based frequency scaling approach may recommend a higher frequency than our model for scenarios in which CPU-intensive processes are running that incur no snooping latency. Likewise, the load-based approach may suggest a lower frequency for a less busy CPU cluster, when another cluster is actually stalling due to cache snooping, in which case our model will suggest the higher frequency.

[4]The `clamp` notation on line 13 includes a calculation of both the valid frequency range of the CPU according to its design specifications, and its maximum actual range advised by the thermal driver, taking into account automated thermal protection.

4.3 Implementation details

We have implemented our approach directly in the Linux kernel by modifying the `task_struct` to allow per-thread CPU PMCs readings. Also, as our `ondemand-anti-snoop` reads CPU PMCs for each thread that has been active over the last period, and CCI PMCs[5] from the interconnect. Because of this additional periodic monitoring of PMCs we note that our approach does incur a small continuous overhead; we cover this in detail in section 5.3.

For the purposes of replication, we note that our implementation has been developed using the Linux kernel v5.3.11; all software and benchmarks have been compiled using GCC v8.3 and run on a Debian 10 Linux distribution. To enable our specific experiments and results to be repeated, all source code and the kernel patch of our implementation is made available as open-source software[6].

5 Evaluation

We evaluate the performance of our approach against three alternatives, using a set of benchmarks discussed in the following section. The first alternative uses the default CPU DVFS `ondemand` governor, and serves as a reference for other comparators. The second one sets the active cluster to its highest frequency, while forcing the idle cluster to its lowest frequency. This setting is meant to determine the worst case when we do not take into account snooping latency but exhibits low energy usage. The third one sets both clusters at their highest frequency, achieving the minimal time of the workload at the expense of higher energy costs overall.

We measure both *execution time* and *energy consumption* of each benchmark using our approach versus these comparison points; each benchmark is executed five times and an average of execution time and energy is taken to normalize noise. Using onboard energy sensors that measure power consumption we take a reading every 263808 μs and accumulate a variable representing the energy consumed during the last sampling period. This accumulator is reset before running a benchmark, and the first reading after the benchmark has finished is reported as energy.

[5]Support for reading CCI-400 PMCs is not currently available in the mainline kernel source code; we have added support for these PMCs to our Linux kernel and have submitted a patch to the Linux mainline maintainer which is currently under review.

[6]URL:Hidden for review.

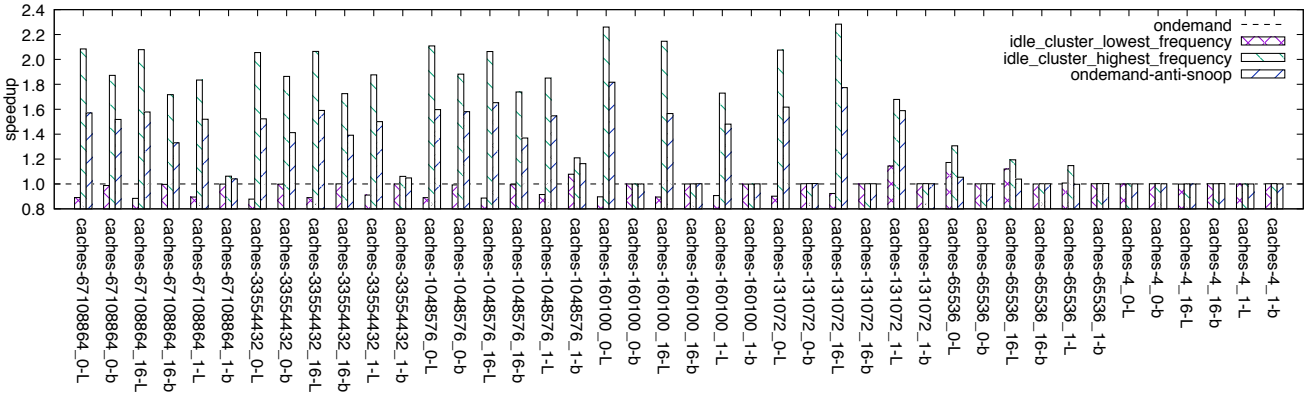


Figure 5. Time results on ccbench::caches, the microbenchmark used to train the decision tree. The Linux ondemand governor as baseline.

5.1 Benchmark selection

We evaluate our approach to performance optimization with a set of 15 different benchmarks. This includes popular benchmarks for CPU profiling chosen to demonstrate performance in the best, worst, and average case for our approach; and also benchmarks which are representative of the general use cases of mobile devices using this CPU architecture.

For our profiling benchmarks, we select particular points from the standard `SPEC2006` benchmark suite for hardware experimentation [18]. Our selection of specific tests is based on existing research by Jaleel [19] which studies this benchmark suite in detail to characterize it in terms of CPU cache memory misses per 1,000 instructions, a metric termed 'MPKI' which interacts with the hardware features our approach is designed to optimize. We specifically select the `gcc` benchmark with `g23` input and `bwaves` with `test` input as these benchmarks face high MPKI for both clusters. For comparison, we also selected `povray` with `train` input as it face very little MPKI, and we use `h264ref` with `train` input as its MPKI appears below 2 Mb, causing the LITTLE cluster to face high snoop latency while the big cluster should not suffer much. These benchmarks are chosen to provide a clear theoretical picture of the characteristics of our approach in best, worst, and middle-ground scenarios. We also re-use the `ccbench:cache` benchmark here, as it was used to train our decision tree as discussed in Section 3.1, to show how it performs at runtime when using our approach.

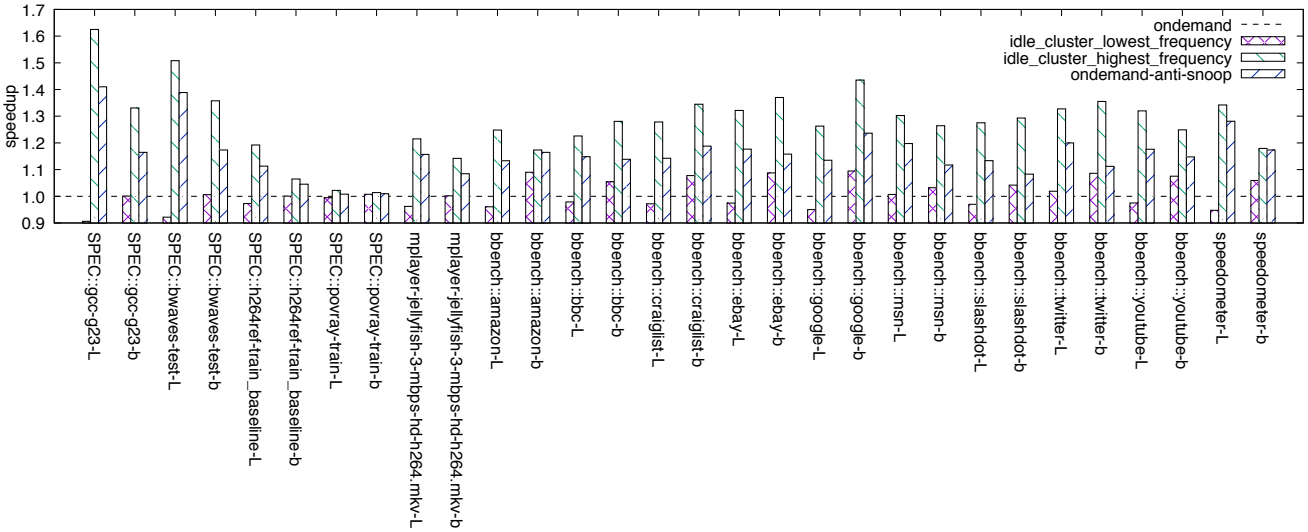
The particular SoC present in the platform we use, the Exynos 5422, is mostly used on mobile devices including smartphones and tablet computers. To reflect two of the dominant end-user applications for these devices we use a set of standard web-browsing benchmark suites and a video decoding benchmark, demonstrating the effects of our approach in a realistic end-user setting. In detail, we use `BBench` [16] and `Speedometer 2.0` [2] for web-browsing benchmarks. `BBench` is used to test general web browsing, which performs automatic browsing by loading and scrolling a selected web page. `Speedometer 2.0` is used to specifically test JavaScript performance in the browser to model highly interactive websites. Both aspects of this web browser benchmarking are performed using the `Chromium` browser controlled by puppeteer [12]. All of the performance measurements that we report include the launch of `Chromium`, page loading, full JavaScript execution and taking a screenshot of the full rendered final web page. The server-side elements of the `Speedometer 2.0` benchmark are hosted on an isolated local Apache web server serviced by a 1 Gb Ethernet connection. For video decoding benchmarks we use the standard Linux `mplayer` application with the command-line parameters `-nosound -vo null -benchmark` options, using a specific video stored locally on the device and publicly available for replication [1].

5.2 Results

The execution time results of our experiments are shown in Figure 5 for the `ccbench::caches` benchmark used to train our approach and Figure 6a for real-world benchmarks. Figure 6b shows the energy results for real-world benchmarks. On both graphs the x-axis labels have the format {name}-{input}-{cluster} and show which benchmark name is being used, its input type, and the cluster (b/L) on which the benchmark is executed (the other cluster is kept idle). On both graphs we report data in terms of how many times better or worse it is than the default Linux `ondemand` governor (thereby using this governor as our consistent baseline). All results involving our dynamic governor are achieved using real-time monitoring of each process, along with stalling effects on the cache interconnect, to dynamically scan for the ideal clock frequencies of both clusters.

We first consider how our new governor behaves against `ccbench::caches`, the benchmark used to train our model as described in section 3. On average, in Figure 5, we see

(a) Speedup.



(b) Energy saving.

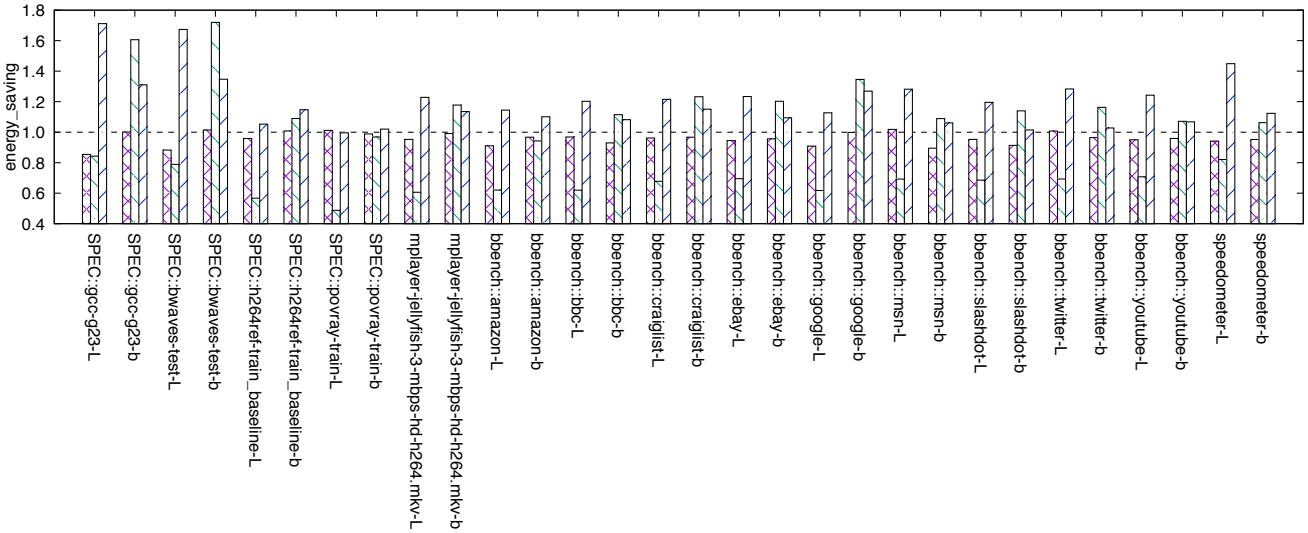


Figure 6. Results of experiments on real-world benchmarks, using the Linux ondemand governor as a baseline.

that our new governor outperforms the ondemand governor by 1.4x. Our governor is also always at least as good as the ondemand governor across all benchmarks, indicating that the inherent overhead of using PMCs at runtime has limited impact on the system compared to the benefits. The varying performance of our governor across specific configurations of this benchmark is simply down to the relative memory intensity and access pattern of each particular configuration – those which incur more L2 cache misses see a higher benefit using our approach.

We next explore realistic benchmarks, including those designed to test specific aspects of our approach and those representative of common end-user activities. Considering execution time first, on Figure 6a, across all results we see that the configuration that keeps the idle cluster at its highest frequency gains the highest performance (at the cost of high

energy consumption as we discuss next). Our approach consistently comes second, followed by the ondemand governor. In some cases this difference is very significant – against the SPEC2006 benchmark our approach delivers 1.4x speedup for the gcc test compared to the ondemand governor. The exact level of speedup is highly dependent on the memory usage characteristics of the benchmark, with the povray test yeilding a very minor speedup due to its low level of main memory usage. Examining real-world applications, we use our web browsing benchmark against a series of different popular web pages, gaining between 1.1x and 1.25x speedup for page loading, while Speedometer JavaScript tests yield up to 1.3x speedup under our approach.

Considering energy, show in Figure 6b, this graph demonstrates the benefit of our approach to the overall performance/energy profile of the device. As an example, we see that SPEC2006 benchmark's gcc test under our approach saves 1.7x the amount of energy compared to the default ondemand governor while also (from the previous graph) completing the benchmark 1.4x faster. This is also useful to compare against the configuration which runs the idle core at its highest frequency: although this configuration completes the benchmark faster than our approach, it also uses far more energy. Our approach therefore finds a useful balance between performance and energy. This is demonstrated throughout almost all of the benchmarks, where our approach offers a significant energy saving over the ondemand governor while also yielding higher performance.

The reason behind this result is that energy expenditure is not linearly related to clock frequency and therefore execution time. The hardware specification of the CPU is such that the clock frequencies of a CPU depend on voltage domains which are in bands. Each voltage domain supports a range of clock frequencies, before reaching a threshold at which a new voltage level is needed for the next set of frequencies. All frequencies within a given voltage level are therefore equal in energy expenditure[7]; in some cases this allows our approach to set a higher clock frequency, resulting in faster execution time, while also gaining a low energy profile.

Finally, we note that all of these benchmarks are subject of different dynamic activity phases over their execution lifetime and none have been seen before by our trained model. This indicates that our approach deals well with fluctuations of memory usage over time by dynamically adjusting frequencies on a continuous basis and also that its training on a single set of focused benchmarks around energy characteristics generalizes very well to good performance on a broad range of new benchmarks.

[7] In reality this is not quite true, as thermal increases vary the precise amount of energy consumed [20], but is sufficiently accurate to explain these results.

5.3 Discussion

Our ondemand-anti-snoop governor aims to avoid snooping latency using dynamic hardware-level information from PMCs, along with a simple trained model of how the CPU clusters behave at different frequencies and with different memory access scenarios.

We next consider the overhead of our approach, and its broader implications. The runtime overhead of our governor comes from the fact that we read PMCs for every individual process (thread) to understand in real-time how the memory accesses of a process of interest interacts with CPU performance. From the SPEC2006 benchmark suite, which are all single-threaded applications, our experiments report that it took 41.82 μs on average to update CPU PMCs. For multi-threaded applications like Chromium, the Speedometer 2.0 benchmark uses 71 threads during peak activity in our experiments, and took 415.18 μs on average to update PMCs. This shows that the number of threads in the system has an impact on how long it takes to read process-level PMC information. Reading the CCI PMCs, meanwhile, took 0.07 μs on average, and is not affected by an increase in the number of processes. This very low update time, coupled with the fact that PMC updates and decision-making do not require any application to stop their execution, even briefly, indicates that the overhead of our approach is far outweighed by the benefits it brings in overall performance and energy usage.

Our new DVFS governor succeeds in limiting snoop latency with a pure-software solution; this approach is valuable in any heterogeneous multi-core design in which cache coherency checks are dependent on the relative clock frequencies of each different cluster. In these chip designs, we are able to train a very simple model on memory access and frequency interaction, and combine this with real-time monitoring to configure the clock frequencies of all clusters in a CPU to an ideal system-wide setting. While some newer CPU designs include extra hardware support to aid with cache coherency, in which the interconnect maintains a list of which memory is currently in the cache of each cluster, a large number of existing devices do not have this capability and so will benefit from our approach. A recent study [28] suggests that as much as 75% of today's smartphone population use CPU designs that was released before 2013, and rely on a cache coherence interconnect with no hardware snoop filter support, making our approach very widely applicable across popular end-user devices today.

6 Related work

There are a large number of works targeting scheduling and DVFS settings for big.LITTLE architectures. In this section we discuss the most closely related research which uses the same hardware platform as that used in this paper.

In a general sense, there are several approaches which use machine learning techniques to combine software and

hardware feature models in order to optimize for performance, energy, or a mixture between the two. Each of these approaches is specific to a particular software package, however, using domain-specific features extracted from that software. Proteus [24], for example, uses a browser extension in Chromium to attempt to predict which core to assign the rendering of each web page to achieve a maximum of performance, based on an offline trained model of page features and their correlation to the characteristics of each core. Similarly, a range of research has explored optimization of OpenCL-specific applications which extract features from OpenCL code and data to predict which core to execute an application at runtime [9, 25, 27]; similar approaches have also been explored for OpenMP [13]. In comparison to this kind of work, our approach operates at a lower level and as such is far more general, directly using real-time hardware-level information with the use of PMCs for CCI activity and memory access characteristics, without the need for any particular source code information or modification.

The closest research works to ours for big.LITTLE architectures are those by Donyanavard *et al.* [14], and Reddy *et al.* [22], both of which consider PMC-based metrics at runtime. In SPARTA [14], the authors demonstrate an approach to task mapping and DVFS; however, the metrics considered in this case only relate to CPU load, ignoring memory access characteristics and snooping latency effects as considered by our approach. Research by Reddy *et al.*, meanwhile, considers real-time MRPI metrics to adjust the clock frequencies of both clusters [8, 22]. MRPI is defined as Memory Reads Per Instruction, which uses PMCs as real-time input and is defined as $(L2\_refill/Instruction\_retired)/CPU\_cycles$. The approach attempts to construct a predictive model of MRPI across both clusters and adjusts frequencies to reduce contention and gain performance. Our research, by comparison, pre-models the specific effects of snooping latency using a single representative benchmark, and uses that model to inform clock frequency decisions based on instantaneous PMC readings without the need for online prediction.

Finally, there are multiple efforts [10, 15, 23] which propose hardware-level modeling for the simulation of the same board used in this paper within the `gem5` simulator. The goal of these works is to support accurate power modeling of workloads, using correlations with the real onboard hardware sensors to validate the behavior of the simulator. This principle supports the development of energy management approaches in simulation which can then be applied to the real-world hardware. However, we note that these simulation models do not consider snooping latency problems relative to the interconnect fabric. One particular simulation model in this category [10] does provide error ranges within the simulator for uncertainty over memory access characteristics, and suggests that one of the sources of this inaccuracy lies in the lack of a complete CCI simulation model; the research reported in our paper confirms this theory and provides a compete, generalized solution to avoiding memory latency issues in real-time on the actual hardware.

7 Conclusions

The increasing demand for performance and energy efficiency has led embedded systems such as mobile and tablet devices to employ heterogeneous multiprocessor system-on-chips. The combination of different kinds of core types and frequency configurations helps to fine-tune energy efficiency and/or performance at runtime. Thanks to full data coherency managed in hardware through an interconnect fabric, the software developer can ignore data cache management as threads spread across processors. However, the interconnect fabric on some SoC can cause significant performance drops if processors are poorly configured. As this we have shown in this paper, these performance drops can be attributed to snooping latency which can occur when the software has large amounts of memory traffic *and* CPU frequencies are set too low aiming to save energy.

We have presented an automated characterization of this snooping latency for any SoC that implements ARM CCI-400 as its interconnect. We build a simple model that takes into account hardware-level information in accordance with software memory usage to detect when snooping latency occurs and its extent, and we use this simple model to develop a new `ondemand-anti-snoop` dynamic frequency governor to manage CPU cluster frequencies and avoid snooping latency.

Evaluation of this governor shows that a speedup of more than 40%, with a 70% energy saving, can be achieved versus the default Linux `ondemand` governor on a real-world application. Our new governor, based on hardware-level use of PMCs, does not depend on any particular software knowledge or modification to operate and resides directly in the operating system, fully transparent to application software.

References

[1] Jellyfish video. http://jell.yfish.us/media/jellyfish-3-mbps-hd-h264.mkv.

[2] Apple WebKit Team. Speedometer2.0. https://browserbench.org/Speedometer2.0/.

[3] ARM. CCI-400. https://www.arm.com/products/silicon-ip-system/corelink-interconnect/cci-400.

[4] ARM. Cortex-A15. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a15.

[5] ARM. Cortex-A7. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a7.

[6] ARM. Exynos 5 Octa (5422). https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/.

[7] ARM. White paper: big.little technology: The future of mobile, 2013.

[8] K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett. AdaMD: Adaptive Mapping and DVFS for Energy-efficient Heterogeneous Multi-cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP(X):1–1, 2019.

[9] C. Bolchini, S. Cherubin, G. C. Durelli, S. Libutti, A. Miele, and M. D. Santambrogio. A runtime controller for OpenCL applications on

heterogeneous system architectures. *CEUR Workshop Proceedings*, 1697(February):29–35, 2016.

[10] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, and M. Robert. Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration. *Proceedings - IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2016*, (5422):201–208, 2016.

[11] C. Celio. Characterizing multi-core processors using micro-benchmarks. https://github.com/ucb-bar/ccbench/wiki, 2009.

[12] Chrome DevTools Team. puppeteer. https://pptr.dev/.

[13] E. Del Sozzo, G. C. Durelli, A. Miele, E. M. G. Trainiti, M. D. Santambrogio, and C. Bolchini. Workload-aware Power Optimization Strategy for Asymmetric Multiprocessors. *Date '16*, pages 531–534, 2016.

[14] B. Donyanavard, T. Muck, S. Sarma, and N. Dutt. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. *2016 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2016*, pages 0–9, 2016.

[15] F. A. Endo, D. Couroussé, and H.-p. Charles. Micro-architectural simulation of embedded core heterogeneity with gem5 and McPAT. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools - RAPIDO '15*, pages 1–6, New York, New York, USA, 2015. ACM Press.

[16] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*, pages 81–90, 2011.

[17] HardKernel. Odroid-XU3. http://www.hardkernel.com/.

[18] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, sep 2006.

[19] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[20] P. Kocanda and A. Kos. Static and dynamic energy losses vs. temperature in different cmos technologies. In *2015 22nd International Conference Mixed Design of Integrated Circuits Systems (MIXDES)*, pages 446–449, June 2015.

[21] A. Pallipadi and A. Starikovskiy. The ondemand governor: past, present and future. *Proceedings of the Linux Symposium*, pages 215–230, 2006.

[22] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh. Online concurrent workload classification for multi-core energy management. *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, 2018*, 2018-January:621–624, 2018.

[23] B. K. Reddy, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett. Empirical CPU power modelling and estimation in the gem5 simulator. *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017*, 2017-Janua:1–8, 2017.

[24] J. Ren, X. Wang, J. Fang, Y. Feng, D. Zhu, Z. Luo, J. Zheng, and Z. Wang. Proteus: Network-aware web browsing on heterogeneous mobile systems. *CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 379–392, 2018.

[25] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi. Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. *ACM Transactions on Embedded Computing Systems*, 16(5s), 2017.

[26] A. Stevens. Introduction to AMBA® 4 ACE™ and big.LITTLE™ Processing Technology. https://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf, 2013.

[27] B. Taylor, V. S. Marco, and Z. Wang. Adaptive optimization for opencl programs on embedded heterogeneous systems. *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Part F128681:11–20, 2017.

[28] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.