Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016

DOI: xxx/xxxx

ARTICLE TYPE

An efficient model for quantifying the interaction between structural properties of software and hardware in the ARM big.LITTLE architecture

Srboljub Stepanovic* | **Georgios Georgakarakos** | **Simon Holmbacka** | **Johan Lilius**

Faculty of Science and Engineering, Åbo Akademi University, Turku, Finland

Correspondence
*Srboljub Stepanovic, Agora, Vattenborgsvägen 5, FI-20500 Åbo. Email: srboljub.stepanovic@abo.fi

Present Address
Agora, Vattenborgsvägen 5, FI-20500 Åbo

Abstract

Heterogeneous architectures offer the opportunity to achieve high performance and energy efficiency by selecting appropriate cores for execution of ever changing software applications. Appropriate core selection depends on the interaction between the structural properties of the software and the hardware that influences performance of the software. We propose a model for efficient core selection when executing software on ARM's big.LITTLE heterogeneous architecture. It features a metric based on the correlation between the performance and the number of last level data cache (LLC) misses on a big and a LITTLE core. Additionally, our model defines a soft threshold in terms of the number of LLC misses that determines efficient core selection. We verify the model using stress and variable workload benchmarks as well as two popular high throughput applications for mutlicore targets namely HEVC and LDPC decoders profiled with XMEM, Linux perf and PMCTrack dynamic tools. Results show that our model can be used for efficient core selection with a relatively small error probability.

KEYWORDS:
big.LITTLE, heterogeneous architecture, cycles per instruction, last level data cache misses

1 | INTRODUCTION

ARM's big.LITTLE is an asymmetric multi-core architecture where cores share the same instruction set but feature significantly different microarchitectures. The Exynos 5422 chip available in the odroid-xu4 board[1] is an instantiation of the big.LITTLE architecture with two sets of cores organized in clusters: four Cortex A7 cores with simple microarchitecture and short pipelines optimized for energy efficiency and four Cortex A15 with complex microarchitecture and deep pipelines optimized for high performance. However, mapping an application on big.LITTLE is not trivial. In order to efficiently use these resources, we need to understand how to optimally map an application on the suitable core to minimize energy consumption while maintaining performance.

In this paper we examine the interaction between structural properties of hardware and software, and evaluate its impact on the achieved performance on both A7 and A15 cores on selected benchmarks. In particular we study how the microarchitecture can exploit the structural properties of the software.

We measure the interaction between structural properties of software and hardware in terms of the number of last level cache (LLC) misses that appear during execution of a program. LLC misses dominantly affect the execution time because memory access time is an order of magnitude larger than latencies caused by other miss events like instruction cache misses, L1 data cache misses and branch mispredictions. LLC latency will cause stalls in execution of instructions that are dependent on the LLC miss instruction. This in turn prevents a big core to benefit from instruction level parallelism using out-of-order execution of independent instructions. This insight can be exploited in order to optimally utilize heterogeneous

big.LITTLE architectures by scheduling applications with a large number of LLC misses on a LITTLE core and the ones with a small number of LLC misses on a big core.

The main contribution of this paper is a model to predict which processor core will achieve better cycles per instruction (CPI) when executing particular application. The model establishes a linear correlation between CPI and the number of LLC misses of an application. Each benchmark from the stress-ng benchmark suite [2] is evaluated for the number of LLC misses and CPIs on both cores. The model is then used to predict CPI for an unknown program after profiling it to get the number of LLC misses. Evaluated CPIs on both cores determine on which core a certain benchmark will be mapped. Furthermore, we establish a soft threshold in terms of LLC misses that divides all benchmarks in the two regions. The first one contains benchmarks that have the number of LLC misses below the threshold, so they will be scheduled on a big core. The second one contains benchmarks that have the number of LLC misses above the threshold, so they will be scheduled on a LITTLE core. We also evaluate LLC misses and CPIs for Multibench benchmarks and for two popular multimedia and communication applications suited for multicore environments i.e. HEVC decoding and LDPC decoding.

The rest of this paper is organized as follows: Section 2 presents related work done in performance estimation and latency characterization. Section 3 analyses the features of A7 and A15 cores of big.LITTLE. Section 4 describes impact of miss events on performance. Section 5 describes the details of our performed measurements regarding cache and memory latencies, number of cache misses and overall performance. Also in section 5 the results of our measurements are presented and discussed. Section 6 concludes the paper.

2 | RELATED WORK

The work presented in [3] shows that there is a correlation between a small and a big core slowdown and memory level parallelism (MLP) ratio as well instruction level parallelism (ILP) ratio between the cores but it does not quantify this correlation. This work also does not specify what the threshold is in terms of memory intensity that would indicate when either MLP- or ILP-ratio should be applied to predict CPI. The work compares the following scheduling policies in terms of performance on a two-core heterogeneous multi-core: their proposed performance impact estimation including both MLP- and ILP-ratio, MLP-ratio, memory-dominance, random and optimal scheduling. We extend this work by creating a unified linear model for predicting CPI based on the number of LLC misses for all benchmarks. We also propose a threshold in terms of LLC misses that determines on which core an application will be scheduled.

The work presented in [4] introduces X-MEM, a cross-platform and extensible memory characterization tool for the cloud. This tool can be used to characterize the memory hierarchy of a specific platform. The tool is also able to measure cache and main memory unloading latency, main memory load latency and read/write behavior. We enhanced this tool to statistically determine each cache and memory component separately, based on the number of cache and main memory hits.

The design of new performance counters for measuring CPI stacks based on interval analysis is presented in [5]. The work explains the impact of long latency instructions and all relevant miss events such as L1 and LLC cache misses, and branch mispredictions on execution time of a program that is divided into intervals between these miss events. We show that LLC miss latency prevails in relation to latencies of other miss events. We take advantage of this fact to construct a model for CPI prediction based only on the number of LLC misses.

The work presented in [6] proposes a symbiotic core execution mechanism for detecting regions of code with high ILP or MLP and shows how to exploit such features on a heterogeneous architecture. The authors propose coarse-grained scheduling to determine appropriate cores outside of the regions with high MLP and also fine-grained scheduling to determine appropriate cores inside such regions. Contrary to their mechanism that is applied on chunks of instructions, our approach is applied on application level. It enables to determine CPI for an unknown benchmark after only profiling benchmarks from representative testing set such as the stress-ng benchmark suite.

Characterization of branch misprediction penalty is explained in [7] through interval analysis. In this interval analysis, superscalar processor performance is viewed as a sequence of inter-miss intervals. The misses that define the intervals are branch mispredictions, (L1 and L2) i-cache misses and long (L2) d-cache misses. They show how the penalty for a particular branch misprediction depends on a preceding miss event and hence how this event affects overall performance. In our work we focus on L2 (LLC) d-cache misses as they most dominantly affect performance.

3 | ARM BIG.LITTLE ARCHITECTURE

The platform chosen for this study is the ARM big.LITTLE multi-core system. We use the Exynos 5422 implementation with four Cortex-A7 [8] cores and four Cortex-A15 [9] cores. The LITTLE A7 is a 32 bit core using a partial dual-issue in-order microarchitecture with an 8-10 stage pipeline as shown in the upper part of Fig. 1. This means that it enables dual-issue instruction execution only for some integer operations; for all other operations A7 is a single-issue machine. This core does not have reorder buffer. The argument for using this kind of microarchitecture is to improve energy efficiency because of its simple construction with few speculative execution units such as branch prediction, Translation Lookaside Buffer

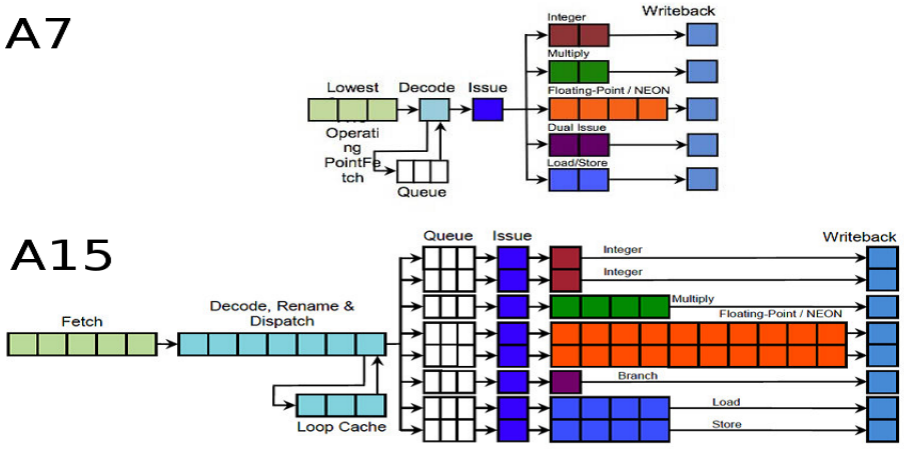

FIGURE 1 Microarchitecture of the A7 and the A15 [10].

(TLB) lookups and no instruction scheduling. The memory system consists of separate L1 instruction (i-cache) and data (d-cache) caches and shared L2 cache as shown in Table 1.

The big A15 cores consist of a more complex 32 bit architecture with a 15-25 stage pipeline with two separate integer and Neon units as shown in the lower part of Fig. 1. The core is also using separate load/store units and a separate unit for branch instructions. The A15 core executes instructions out-of-order to gain high throughput that enables more concurrency in the hardware. Three instructions can be processed per clock cycle. Out-of-order execution requires register renaming to dynamically alter register names in order to resolve false dependences and enable parallel hardware to operate on data without dependences. A reorder buffer is used to commit the instructions in correct order after out-of-order execution. It also makes possible that the correct process state can be restored in case of a branch misprediction [7]. Reservation stations store instructions which await operands to be broadcasted without first written back to memory. The memory system consists of separate L1 i-cache and d-cache and shared L2 cache as shown in Table 1. The L1 i-cache and d-cache have Least Recently Used (LRU) cache replacement policy.

4 | IMPACT OF MISS EVENTS ON PERFORMANCE

In this section we will show how a particular miss event affects performance and what the parameters are that characterize the impact of each miss event on performance. Our approach is based on an observation from [5] that pipeline miss events can be split into frontend and backend pipeline miss events.

Indeed, miss events depend on the interaction between structural properties of software and hardware. Structural properties of software related to cache miss events are the frequency and the order of accessing particular memory locations by a program. Structural properties of hardware related to cache miss events are cache size, associativity, fixed line length and replacement policy. L1 cache may contain a subset of the data in L2 (LLC) cache. L2 cache contains a subset of the data in main memory. This means that more than one of the memory blocks can be mapped to the same cache line. Therefore, every access to a location of the memory blocks mapped to the same cache line as the memory block currently stored into this cache line will cause a cache miss event. Also, the cache is too small and cannot hold all of the memory blocks referenced in the program. A cache replacement policy defines how old data in the cache is replaced with new data. In this way two different cache replacement polices applied to the same program will cause different number of cache miss events. The parameters that characterize each miss event are penalty, rate and

TABLE 1 LITTLE/big core cache structure.

Parameter	L1 i-cache	L1 d-cache	L2 cache
Size[KB]	32	32	512/2048
Associativity[way]	2	4/2	8/16
Fixed line length[B]	32/64	64	64
Cache replacement policy	pseudo random/LRU	pseudo random/LRU	pseudo

overlapping effect. Penalty is the time needed for particular miss event to be resolved. Rate is the number of particular miss events relative to the total number of instructions. An overlapping effect appears when two miss events happen at the same time.

The frontend pipeline miss events are L1 instruction cache misses and branch mispredictions. Penalty of these miss events consists of two components on a big core. The first one is caused by discharging instructions that follow a miss event from the reorder buffer. The second one is caused by refilling the pipeline with new instructions. On a LITTLE core refilling the pipeline is the only penalty. Because the frontend pipeline length of a big core is larger than the one of a LITTLE core and because of the size of a big core reorder buffer, the frontend pipeline miss event penalty is much larger on a big core. Instructions are executed only serially in the frontend pipeline. Therefore, the frontend pipeline miss events do not overlap at all.

The backend pipeline miss events are L1 and LLC data cache misses as well as long latency instructions. L1 data cache misses and long latency instructions are hidden through out-of-order execution on a big core. When they occur, some other instructions present in the reorder buffer will be executed until data accessed from LLC data cache become available or the long latency instruction is finished. The compulsory condition that must be fulfilled in order to avoid stalls due to these miss events is that the LLC access time and the long latency instruction execution time are smaller than the execution time of instructions present in the reorder buffer. We show that this condition is always fulfilled in section 5. This shows clear advantage of a big core when considering these events.

LLC misses are events that cannot be hidden through out-of-order execution because the memory access time is always larger than the time needed to execute all instructions present in the reorder buffer. This means that all LLC misses that are handled in parallel can be considered as one miss. But because instructions may be executed in parallel in the backend pipeline also, LLC misses might overlap if they are present in the reorder buffer at the same time as was observed in [3]. When a LLC cache miss occurs instruction level parallelism cannot be exploited by out-of-order execution of a big core and therefore it is meaningful to schedule benchmarks with high number of LLC misses on a LITTLE core. The main claim of our work is summarized in Table 2.

5 | EXPERIMENT AND ANALYSIS

We use the X-Mem tool [4] to measure latency of the ARM big.LITTLE memory system. X-Mem measures the average aggregate L1 cache, L2 cache and main memory latency (L). We measure this latency using one worker thread across working sets from 4KB to 1GB. A working set is a private region of memory the worker thread operates on. X-Mem is doing this in such a way that for working sets up to L1 cache size it measures L1 cache hit latency (L1L), up to L2 cache size it measures average aggregate L1 and L2 hit latency and for working sets over L2 size it measures average aggregate L1, L2 and main memory hit latency. We modify X-Mem to measure the number of L1 cache (L1H), L2 cache (L2H) and main memory (MEMH) hits to be able to statistically determine L2 cache (L2L) and main memory (MEML) latencies. We use the equation shown below to calculate L2L and MEML:

$$L = L1H \times L1L + L2H \times L2L + MEMH \times MEML \quad (1)$$

At the end, we average out latency across all working set sizes belonging to corresponding L1 cache, L2 cache or main memory region respectively. The corresponding latencies for LITTLE core are shown in Fig. 2. We measure them at the highest frequency for a LITTLE core of 1.4GHz. The same set of cache and memory latencies is applied to different applications on a particular core type. The X-Mem tool introduces only negligible overhead. The tool does not specify logical/physical CPU core affinity. It is also not possible to turn off the first LITTLE core on the odroid-XU4 board and measure latencies of a big core separately. But because the same technology process is applied in implementing memory system on both cores, we assume that a big core has the same latencies.

We measure the following latencies for L1 data cache, LLC data cache and memory (Fig. 2): 2.49ns (4.98 cycles), 16.46ns (32.92 cycles) and 223.29ns (446.58 cycles) respectively where the number in brackets is the number of cycles relative to 2GHz. When we compare LLC latency to the size of reorder buffer (128) multiplied by only one cycle needed for an instruction execution, we can notice that L1 data cache misses cannot cause stalls as stated in section 4. When we compare memory latency with size of reorder buffer multiplied by an average latency of 3 cycles, there will be stalls even if the reorder buffer is almost full and all present instructions in the reorder buffer are dependent from each other and from the LLC miss instruction.

TABLE 2 The main claim of our work.

The number of LLC misses of a program	The core type a program to be scheduled on
Low number of LLC misses (below the threshold)	big core (A15)
High number of LLC misses (above the threshold)	LITTLE core (A7)

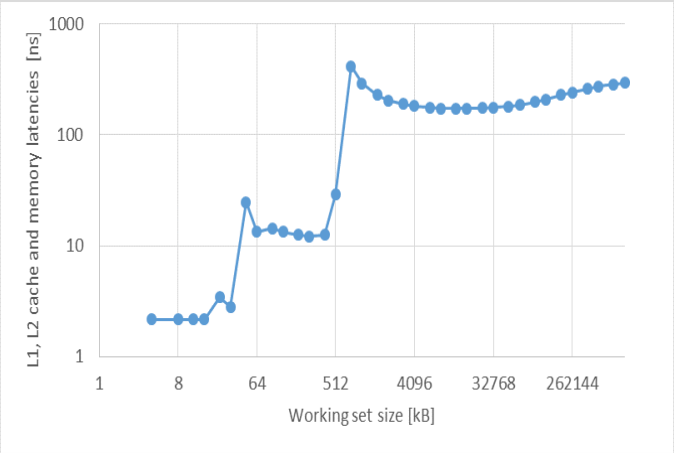

FIGURE 2 Determining cache and memory latencies.

We conduct miss event and performance measurements for all benchmarks on odroid-XU4 board with big.LITTLE processor architecture described in section 3. We run each benchmark either on an A15 core or an A7 core. We use an A15 core with performance governor on the highest frequency of 2GHz and an A7 core with powersave governor on the lowest frequency of 200MHz. We use perf tool from user space via command line to measure the number of instructions, the number of clock cycles and the number of different miss events.

We also use PMCTrack tool [11]. PMCTrack is an open-source performance monitoring tool for Linux that allows gathering performance monitoring counters' (PMC) values from user space via libpmctrack library. This library enables to characterize performance of specific code phases via PMCs in any program written in C. Libpmctrack's API makes it possible to specify the desired PMC configuration to the PMCTrack's kernel module at any point in the application's code. The corresponding event counts for any code phase are then obtained by enclosing the code between invocations to the pmctrack_start_counters() and pmctrack_stop_counters() functions.

5.1 | Stress-ng benchmark suite

We use benchmarks from stress-ng benchmark suite in order to establish the correlation between the number of LLC misses and CPI. They contain different instruction types such as ALU, memory, branch, integer multiply and SIMD instructions to stress corresponding execution units separately and some of them together. Because one LITTLE core is always turned on, we use taskset option of the benchmark suite to specify only one big core. All benchmarks are executed 10 bogo operations to provide the same number of instructions executed on both cores for a particular benchmark. Bogo operations are iterations of the stressor i.e. benchmark during the run. This is metric of how much overall "work" has been achieved in bogo operations.

We show that the number of LLC cache misses per 10K instructions correlates well with CPI on both cores, by evaluating the accuracy of the CPI predictor model. Because the model is a linear function, model accuracy is a good indicator for correlation: if the model is accurate, CPI correlates well with the number of LLC misses per 10K instructions, and vice versa.

The model that transforms numbers of LLC misses per 10K instructions into CPI is shown in Fig. 3. We build this model using the numbers of LLC misses and CPI values from the benchmarks. We find that a linear relationship between the numbers of LLC misses and CPI fits best. We use the following equation to calculate the CPI from the number of LLC misses:

$$CPI(MPI) = a \times MPI + b \tag{2}$$

where MPI is the number of LLC misses per 10K instructions. Parameters *a* and *b* are determined using a least-squares fit. Every point represents the CPI (y-axis) and the corresponding number of LLC misses per 10K instructions (x-axis). There are 58 benchmarks per core. The fitted model is indicated by the lines. It is clear that a linear model is appropriate for this data, and that fit is relatively good. Fig. 4 shows the relative error for the proposed model. The average absolute relative error between real measurements and the model for A15 is 9.71% while for A7 is 21.67%. We can notice that the error continuously decreases as the number of LLC misses increases for almost all benchmarks because the memory component dominantly affects CPI in relation to other miss events for the large number of LLC misses per 10K instructions.

We also run the stress-ng benchmarks on 600MHz and 1GHz either on A15 or A7 core with userspace governor. Parameter *a* in (2) increases while parameter *b* is approximately constant as frequency increases on both cores. This means that the highest observed frequencies 1GHz on A7 core and 2GHz on A15 core cause most stalls in execution i.e. the highest CPI for the same benchmark. These frequencies also cause the highest

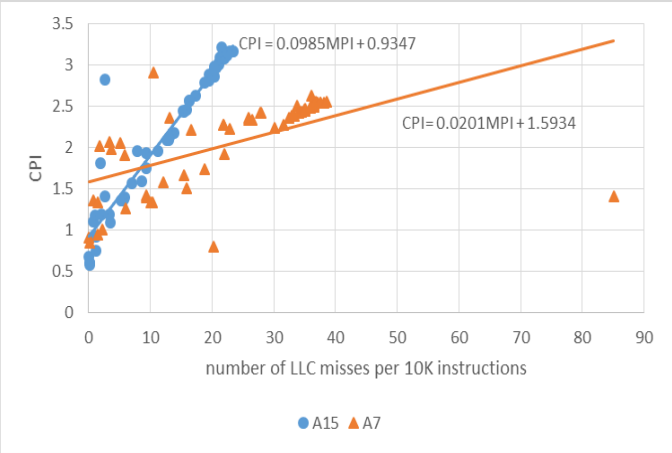

FIGURE 3 The CPI model for A15 and A7 cores.

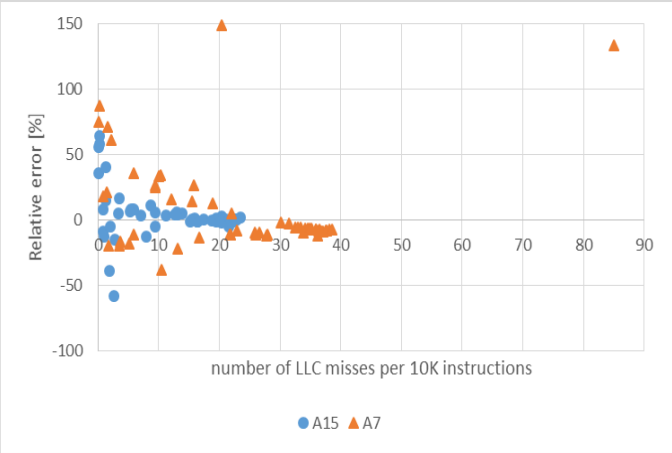

FIGURE 4 The relative error of CPI model.

core temperature. Stalls happen in order to decrease the core temperature because userspace governor keeps fixed operating frequency i.e. there is no automatic frequency scaling to scale frequency down and decrease the temperature. We obtain that CPI on A15 is better than CPI on A7 on the same frequency (600MHz or 1GHz) for every benchmark except ackermann.

In the next two subsections we present two different ways to select an appropriate core:

- using equations in (2).
- using thresholds derived from the same equations.

5.1.1 | Predicted core selection using equations

In this paper, we assume that the big and LITTLE cores have different cache hierarchy, i.e. the same number of cache levels but different cache sizes and/or structures at each level. In other words, we assume that the number of misses is not constant across core types. To illustrate this we show measured and predicted CPI values for all stress-ng benchmarks sorted according to the number of LLC misses per 10K instructions on both core types in Figures 5 and 6 respectively. CPI values are predicted using equations in (2). We can notice that order of sorted benchmarks is not the same on big and LITTLE cores. For example, sieve benchmark is the third on a big core with 0.19 misses while it is the last on a LITTLE core with 85.08 misses per 10K instructions.

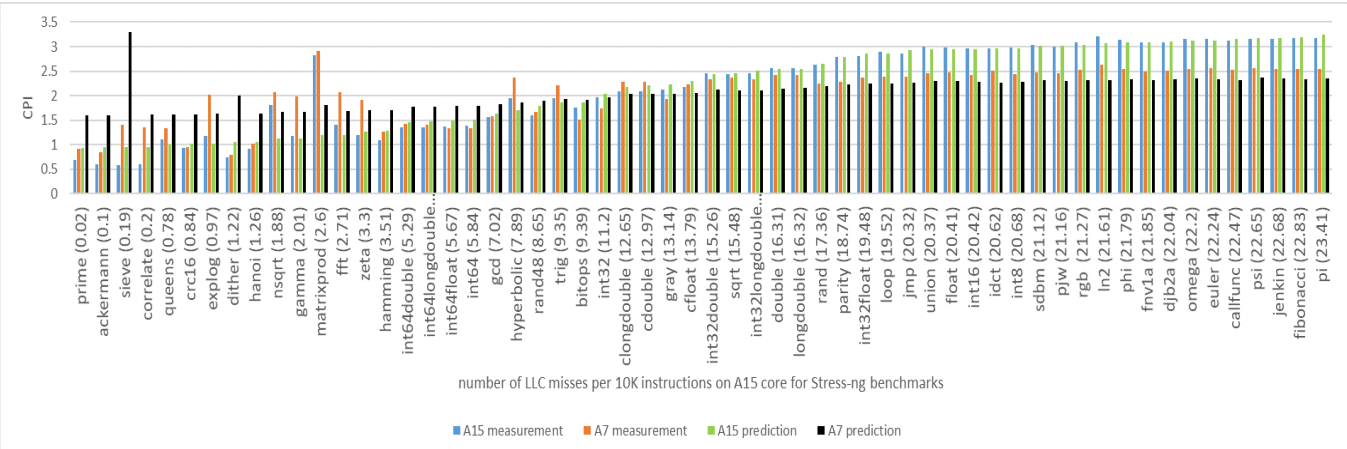

FIGURE 5 The measured and predicted CPI values for Stress-ng benchmarks executed either on an A15 or an A7 core and sorted according to the number of LLC misses per 10K instructions on an A15 core.

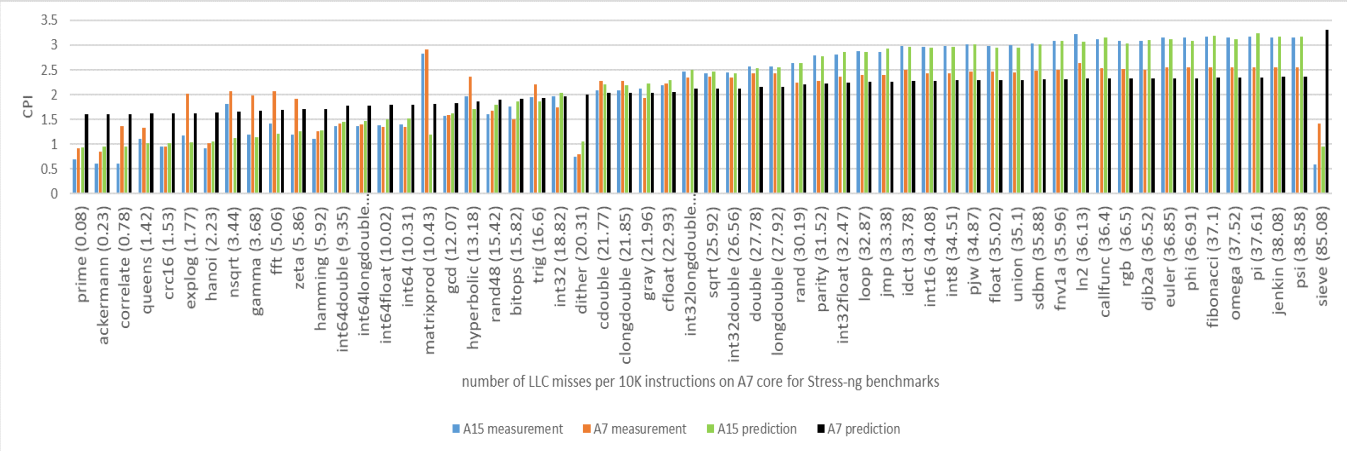

FIGURE 6 The measured and predicted CPI values for Stress-ng benchmarks executed either on an A15 or an A7 core and sorted according to the number of LLC misses per 10K instructions on an A7 core.

We can also notice a phenomenon when considering measured CPI on big and LITTLE cores that benchmarks with low number of LLC misses on a big core from prime to int64longdouble should be executed on a big core because of lower CPI on that core and benchmarks with high number of LLC misses from int32double to pi should be executed on a LITTLE core. The benchmarks in the middle range from int64float to cfloat should be executed alternatively on a big or a LITTLE core. This phenomenon is even more expressed when considering predicted CPI. Then, all benchmarks with low number of LLC misses from prime to bitops should be executed on a big core and all benchmarks with high number of LLC misses from int32 to pi on a LITTLE core. Thus, we can notice that benchmarks having 9.39 or less LLC misses per 10K instructions on a big core should be executed on a big core and benchmarks having 11.2 or higher should be executed on a LITTLE one according to the predicted CPI. Further, for 6 out of 58 benchmarks predicted core selection does not match measured core selection. All these benchmarks belong to the middle range: int64float, int64, bitops, clongdouble, cdouble and cfloat. Thus, for all other benchmarks that are outside the middle range, predicted core selection is 100% accurate. This implicitly indicates that there is an initial probability of around 10.3% that an unknown benchmark will be executed on a wrong core and that this wrongly mapped benchmark most probably belongs to the middle range.

In the similar way, for benchmarks sorted according to the LLC misses per 10K instructions on a LITTLE core we can notice again that benchmarks with low number of LLC misses on a LITTLE core from prime to int64longdouble should be executed on a big core while benchmarks with high number of LLC misses from int32longdouble to psi should be executed on a LITTLE core according to the measured CPI. An exception is only sieve benchmark with the highest number of LLC misses (85.08) that should be executed on a big core. There are also benchmarks in the middle range from int64float to cfloat where some benchmarks should be executed on a big and some benchmarks on a LITTLE core. When considering predicted CPI, all benchmarks except one (int32) with low number of LLC misses should be executed on a big core and all benchmarks with high number of LLC misses except one (dither) should be executed on a LITTLE core. Thus, we can notice that benchmarks having 16.6 or less LLC misses per 10K instructions on a LITTLE core should be executed on a big core and benchmarks having 21.77 or higher should be executed on a LITTLE one according to the predicted CPI. Also, for 6 out of 58 benchmarks (10.3%) predicted core selection does not match measured core selection. These benchmarks belong to the middle range and are the same like in the previous case. Again, for all other benchmarks that are outside the middle range, predicted core selection is 100% accurate.

5.1.2 | Predicted core selection using thresholds

From the previous analysis it is clear that each core type has its own threshold in terms of the number of LLC misses per 10K instructions that can indicate appropriate core selection for each benchmark and only one can be applied at a time for that purpose. We first decide which core type threshold to use and then profile each benchmark only on a core type the threshold belong to, compare the number of LLC misses of each benchmark with the threshold and select a core type on which each benchmark will be executed. We propose a simple algorithm to determine the thresholds on both core types:

1. Sort benchmarks according to the number of LLC misses per 10K instructions on a big/LITTLE core.

2. Find the last benchmark that should be executed on a big core before the first benchmark that should be executed on a LITTLE core according to the predicted CPI values on both cores (lower CPI is better).

3. Find the first benchmark that should be executed on a LITTLE core after the last benchmark that should be executed on a big core according to the predicted CPI values on both cores.

4. The range that includes all the benchmarks between these two benchmarks except these two is called the middle range.

5. The proposed threshold on a big/LITTLE core is the average of the numbers of LLC misses per 10K instructions for these two benchmarks.

According to the last two columns in Fig.5, the first benchmark that should be executed on a LITTLE core is int32. Hence, last benchmark that should be executed on a big core before int32 is bitops. And vice versa, the last benchmark that should be executed on a big core is bitops while the first benchmark that should be executed on a LITTLE core after bitops is int32. The proposed threshold on a big core is the average of LLC misses of bitops and int32, that is 10.29. Thus, all benchmarks with LLC misses below the threshold should be executed on a big core, otherwise on a LITTLE core. Because this threshold is between bitops and int32 benchmarks, the predicted core selection using the threshold on a big core is the same as in the case of using the equations in (2) with probability of wrong core selection around 10.3%. According to the last two columns in Fig.6, int32 is the first benchmark that should be executed on a LITTLE core and trig is the last benchmark that should be executed on a big core before int32. dither is the last benchmark that should be executed on a big core and cdouble is the first benchmark that should be executed on a LITTLE core after dither. The proposed threshold on a LITTLE core is the average of LLC misses of trig and cdouble that is 19.18. This means that the threshold is between int32 and dither benchmarks. Thus, int32 should be executed on a big and dither and sieve should be executed on a LITTLE core that contradicts predicted core selection using equations and selection using measured CPI values. This results in wrong predicted core selection using the threshold on a LITTLE core for 3 more benchmarks that is in total 9 out of 58 benchmarks (15.52%).

5.2 | Multibench benchmark suite

We validate equations in (2) and proposed thresholds in terms of optimal core selection using Multibench benchmark suite[12]. The suite targets multicore processor performance using a wide variety of application-specific workloads. It contains both data processing and computationally intensive tasks. Each workload consists of several work items. Work items are from multiple segments such as networking and consumer. For example, rotate-4Ms1 is simple algorithm for image rotation with very little computation. Depending on image size can exercise the system in interesting ways. Many other applications use similar data movement patterns.

We only show measured and predicted CPI values for all Multibench benchmarks sorted according to the number of LLC misses per 10K instructions on a big core in Fig. 7. CPI values are predicted using equations in (2). When considering predicted CPI (two last columns in Fig. 7), all benchmarks with low number of LLC misses on a big core from ippktcheck-4M to 4M-tcp-mixed should be executed on a big core and all benchmarks with high number of LLC misses from md5-4M to ipres-4M on a LITTLE core. We can notice that for 5 out of 18 (27.78%) benchmarks predicted core selection does not match measured core selection. 4 of these benchmarks belong to the middle range: empty-wld, md5-4M, rgbcmyk-4M and 4M-cmykw2 and one benchmark (ippktcheck-4M) is with the lowest number of LLC misses on a big core. Thus, for all other benchmarks that are outside the middle range and except ippktcheck-4M, predicted core selection is 100% accurate. Because the proposed threshold (10.29) is between 4M-tcp-mixed and md5-4M benchmarks, the predicted core selection using the threshold on a big core is the same as in the case of using the equations with probability of wrong core selection around 27.78%.

When considering predicted CPI, benchmarks with low number of LLC misses on a LITTLE core from idct-4M to 4M-tcp-mixed should be executed on a big core and benchmarks with high number of LLC misses from 4M-check-reassembly-tcp-cmykw2-rotatew2 to 4M-reassembly on a LITTLE core. The first 3 benchmarks from the middle range md5-4M, 4M-cmykw2 and rgbcmyk-4M should be executed on a LITTLE core while the last two empty-wld and ippktcheck-4M should be executed on a big core. We can notice that for 5 out of 18 (27.78%) benchmarks predicted core selection does not match measured core selection. All these benchmarks belong to the middle range. Thus, for all other benchmarks that are outside the middle range, predicted core selection is 100% accurate. Because the proposed threshold (19.18) is between 4M-tcp-mixed and md5-4M benchmarks, the predicted core selection using the threshold on a LITTLE core is wrong for 2 less benchmarks (empty-wld and ippktcheck-4M) than in the case of using the equations, that is in total 3 out of 18 benchmarks with probability of wrong core selection around 16.67%.

These results show that predicted core selection using either the equations in (2) or the proposed threshold is correct for all benchmarks from this testing set except one (ippktcheck-4M) outside the middle range.

5.3 | HEVC decoding application

In order to evaluate performance behavior of big.LITTLE cores we also use High Efficiency Video Coding (HEVC)[13] as our target test application. The reason to use HEVC comes from the growing adaptation of big.LITTLE in SoCs targeting mobile devices, where video based media applications such as content streaming is increasingly popular[14]. HEVC algorithm also manifests significant complexity as well as diversity of operations ranging

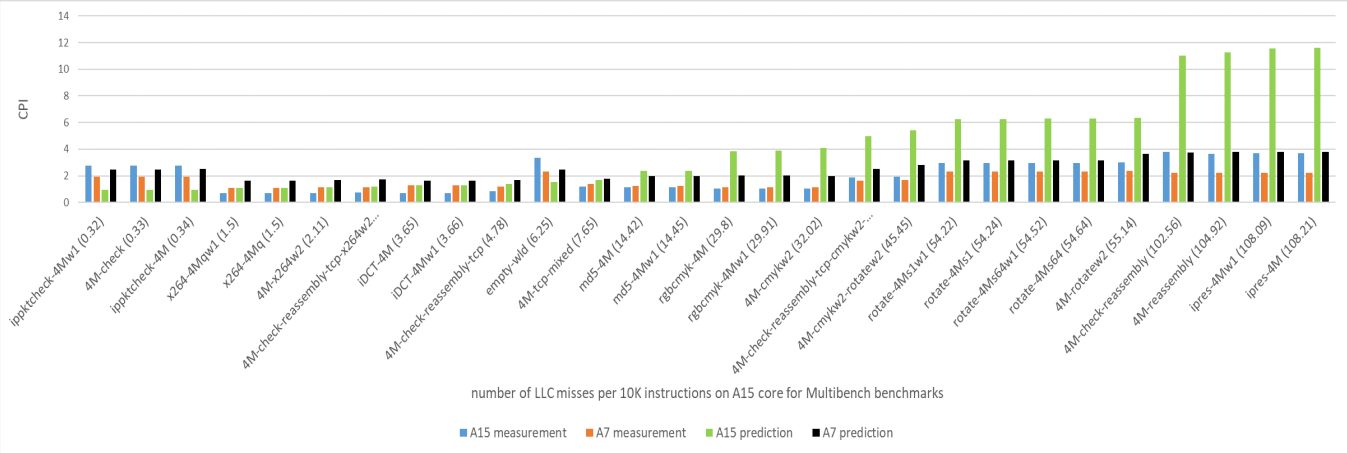

FIGURE 7 The measured and predicted CPI values for Multibench benchmarks executed either on an A15 or an A7 core and sorted according to the number of LLC misses per 10K instructions on an A15 core.

from memory intensive operations (motion estimation), compute intensive operations (prediction, rate-distortion optimization, transformation) and hybrid combinations. We benchmark the A7 and A15 cores of the big.LITTLE architecture using an open source implementation of the HEVC decoder. For the decoding measurements, we create input streams with two different structures using the x265 implementation of the HEVC encoder [15] in order to examine performance differentiation with respect to different workloads. We use streams with two different Group of Picture (GOP) structures: one with frames encoded using only Intra prediction (I frames) and one with frames encoded using motion estimation as well (I, P and B frames). The difference is that P and B frames require memory accesses in order to perform motion compensation during decoding. This creates a more memory-intensive workload compared to an all-Intra frame GOP where prediction is solely compute intensive and no memory accesses are needed for prediction except when accessing cached neighbor pixels of each block. We are also varying input streams resolution from the lowest (QCIF), middle (CIF) to the highest (full HD). We used nine QCIF, nine CIF [16] and seven full HD [17] input streams. The rest of the parameters for our streams are kept in typical values. Table 3 summarizes the input streams configuration.

We obtain that CPI value is approximately constant for almost all input stream configurations of HEVC decoding benchmark running on a particular core type. We also obtain that CPI value is always lower on a big core (lower is better). The last observation is in line with the proposed threshold (10.29) on a big core in the subsection 5.1. We get for 49 out of 50 input stream configurations executed on a big core that the number of LLC misses per 10K instructions is below the threshold and also that CPI on a big core is lower than CPI on a LITTLE core. Therefore, these configurations will be correctly scheduled on a big core. We get for only one input stream with full HD resolution and IBBBP frame GOP structure that the number of LLC misses (10.55) is above the threshold and CPI on a big core is better. Therefore, only this input stream configuration will be wrongly scheduled on a LITTLE core. Also, only this input stream configuration will be wrongly scheduled on a LITTLE core when using equations in (2).

We notice small difference in performance behavior for input streams that have either all Intra or IBBBP frame GOP structure when considering a particular core type (Fig. 8). Decoding B and P frames from IBBBP frame GOP requires large amount of memory accesses when fetching pixels of blocks within previously decoded frames while decoding I frames from all Intra frame GOP requires small amount of memory accesses because it fetches pixels of blocks only from the current frame. This causes that all input streams with all-Intra frame GOP will have lower CPI than the same input streams with IBBBP frame GOP on a big core and around 68% of them on a LITTLE core.

We also notice small difference in performance behavior for input streams that have either QCIF, CIF or full HD resolution when considering a particular core type (Fig. 9). The highest resolution (full HD) causes the most memory accesses on both core types while the middle resolution (CIF) causes the least memory accesses on a big core and the smallest resolution causes the least memory accesses on a LITTLE core in average for all input streams. When considering CPI, the lowest resolution causes highest CPI on both core types while the middle resolution causes the lowest CPI on a big core and the highest resolution causes the lowest CPI on a LITTLE core also in average for all input streams.

5.4 | LDPC decoding application

More optimal scheduling would be achieved by dividing an application into phases and re-mapping these phases dynamically to different cores according to the number of LLC misses during them [6]. In order to prove this concept we analyze Low-Density Parity-Check (LDPC) decoding iterative algorithm known as Minimum Sum Algorithm (MSA) [18]. LDPC codes have gained popularity for their near channel limit performance

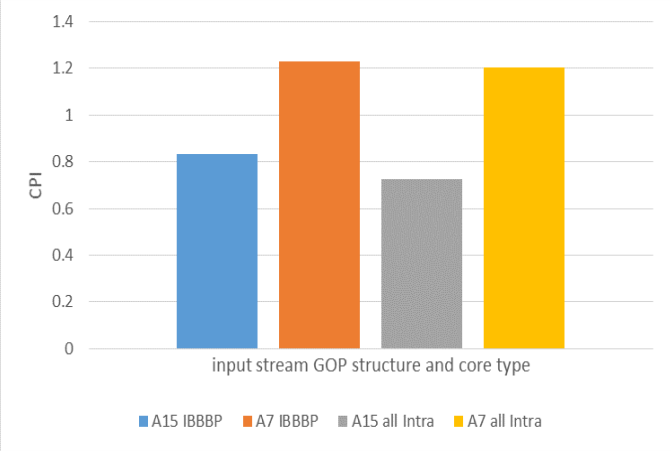

FIGURE 8 The average CPI values for HEVC decoding input streams with IBBBP and all Intra GOP structures executed either on an A15 or an A7 core.

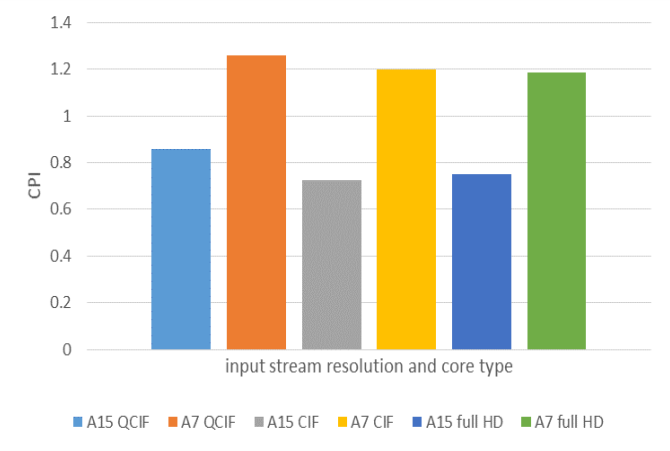

FIGURE 9 The average CPI values for HEVC decoding input streams with QCIF, CIF and full HD resolutions executed either on an A15 or an A7 core.

and have been adapted in many applications ranging from deep space missions to terrestrial broadcasting for DVB-S2, DVB-T2 and DVB-C2 standards[19]. We use an open source implementation of MSA found in[20]. First, we identify 8 phases of MSA implementation that are shown in Table 4. This implementation clearly separates "checknode update" and "variablenode update" phases that are compute intensive and "checknode to variablenode transition" and "variablenode to checknode transition" phases that are memory intensive and only shuffle memory locations. We verify this observation by profiling all phases on both core types using PMCTrack tool. Measured LLC misses per 10K instructions, predicted CPI values using equations in (2) and predicted core selection based on these values for each phase are also shown in Table 4. The predicted core selection will be the same if we use the proposed threshold on a big/LITTLE core because only the first two phases have the number of LLC misses below the threshold.

Second, we map each phase either on a big or a LITTLE core at a time using a pthread mapping mechanism. The pthread mechanism for setting CPU affinity takes a parameter that represents core on which a phase is executed. Mapping each code phase is then achieved by preceding the code with an invocation of the mechanism . In total, for 8 phases executed on 2 cores there are 256 different scheduling combinations. We measured CPI for all combinations and best, worst, median and predicted cases are shown in Fig.10. The best case (bbLLbLbL) indicates that 4 phases (1st, 2nd, 5th and 7th) with the least LLC misses should be executed on a big core while 4 other phases with the most LLC misses should be executed on a LITTLE core. This proves our concept that compute intensive phases should be executed on a big core and memory intensive phases should be executed on a LITTLE core. Our predicted combination (bbLLLLLL) is generally correct being placed between the median and the best case. The difference exists only for 5th and 7th phase. Although these phases are memory intensive with the number of LLC misses above the threshold, they are also highly compute intensive. The reason for not matching the best case is that impact of compute operations significantly prevail memory operations and these phases should be executed on a big core. The worst case is just opposite to the best case. The case when all phases are executed on a big core is slightly better than the median cases and vice versa, the case when all phases are executed on a LITTLE core is slightly worse than the

TABLE 3 INPUT STREAMS CONFIGURATION.

Parameter	Configuration
Encoder	x265
Profile, Level	Main
Resolution	QCIF, CIF, full HD
Throughput	25 fps, 100 frames
GOP	IBBBP or full Intra
Motion search method	Exhaustive search
Motion search range	57
Filters	Deblocking and Sample Adaptive Offset

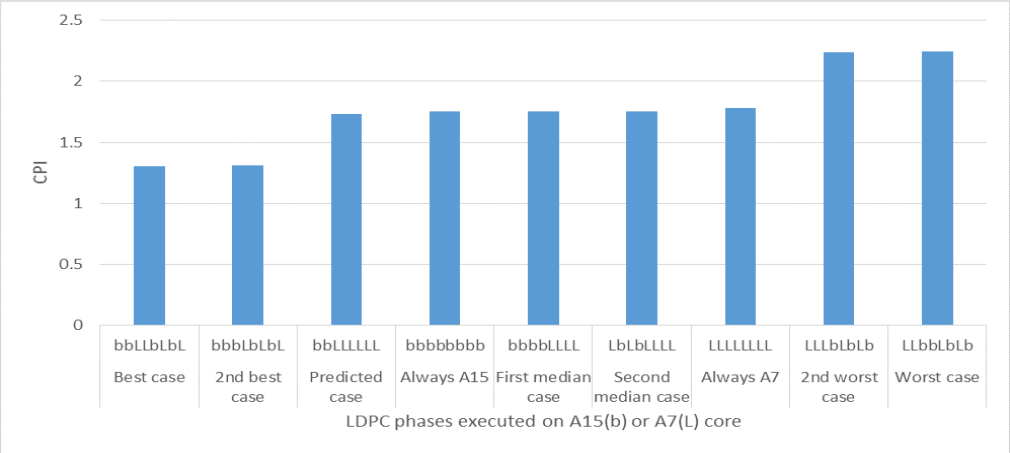

FIGURE 10 CPI for LDPC when its phases executed either on an A15 or an A7 core at a time.

median cases. The most benefit of around 71.6% that could be achieved with this memory access aware scheduler is when comparing the best and worst cases. When comparing the best case to the more real "always A15" and "always A7" cases benefits are around 33.99% and 36.37% respectively. This shows that is worthwhile to take into account memory accesses of an application and its phases when designing a scheduler.

6 | CONCLUSIONS

In this paper we investigated how miss events affect overall performance of the ARM big.LITTLE architecture. We explained what performance impacts of particular miss events are relative to each other. We concluded that LLC misses have the greatest impact on performance. Thus, we constructed the model that establishes correlation between the number of LLC misses and CPI representing overall performance. This model is represented by two linear equations, each one related to specific core type, big or LITTLE. These equations are used as metrics to predict appropriate core selection based on the number of LLC misses per 10K instructions on both core types. This model also enables us to propose soft threshold in terms of the number of LLC misses as an alternative to determine on which core to schedule a certain application in order to maximize performance. All applications with the number of LLC misses above this threshold will be scheduled on a LITTLE core because large number of LLC misses does not allow a big core to exploit ILP through out-of-order execution. We validated our equations and the proposed threshold with the Multibench benchmarks and the popular HEVC decoding application. We obtained that the predicted core selection is very similar when using the equations or the threshold and it is almost 100% accurate for benchmarks that do not belong to the middle range, i.e. that are not too close to the threshold. Finally, we showed that the best option is to divide an application into phases and then to map each phase to appropriate core based on the number of memory accesses of each phase instead to map the application as a whole. This approach could be completely applied on the other hardware platforms supporting performance monitoring counters.

TABLE 4 LLC misses and predicted CPI on A15 and A7 cores for LDPC phases .

Phase	Executed	LLC misses per 10K ins. on A15/A7	Predicted CPI on A15/A7	Predicted core
1. Initialize all message structures	1	5.82 / 7.18	1.51 / 1.74	A15 (b)
2. Log Likelihood Ratios (LLRs)	22	0.56 / 0.66	0.99 / 1.61	A15 (b)
3. LLR to checknode	22	76.91 / 95.59	8.51 / 3.51	A7 (L)
4. Iterator	22	618.09 / 1080.25	61.82 / 23.31	A7 (L)
5. Checknode update	3080	19.83 / 19.9	2.89 / 1.99	A7 (L)
6. Checknode to variablenode transition	440	148.36 / 206.97	15.55 / 5.75	A7 (L)
7. Variablenode update	3520	41.23 / 41.4	5.0 / 2.43	A7 (L)
8. Variablenode to checknode transition	440	210.3 / 273.31	21.65 / 7.09	A7 (L)

In future work we plan to explore an efficient way to identify memory and compute intensive workloads so that we can refine the accuracy of core mapping prediction. We also plan to explore memory contention on the big.LITTLE architecture when several applications compete for the same memory resources. We want to design contention-aware scheduling algorithm based only on memory accesses of applications executed in isolation. We want to quantify this algorithm relative to the optimal schedule. Furthermore, we want to quantify performance degradation of each application scheduled using this algorithm when co-executing with other applications.

References

1. Odroid-xu4 board. http://www.hardkernel.com/main/products/prdt_info.php [06 September 2016]; 2013.
2. Stress-ng Benchmark Suite. http://kernel.ubuntu.com/~{}cking/stress-ng/ [17 November 2016]; 2016.
3. Van Craeynest K, Jaleel A, Eeckhout L, Narvaez P, Emer J. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: :213–224IEEE Computer Society; 2012.
4. Gottscho M, Govindan S, Sharma B, Shoaib M, Gupta P. X-Mem: A cross-platform and extensible memory characterization tool for the cloud. In: :263–273IEEE; 2016.
5. Eyerman S, Eeckhout L, Karkhanis T, Smith JE. A performance counter architecture for computing accurate CPI components. In: :175–184ACM; 2006.
6. Patsilaras G, Choudhary NK, Tuck J. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization (TACO)*. 2012;8(4):28.
7. Eyerman S, Smith JE, Eeckhout L. Characterizing the branch misprediction penalty. In: :48–58IEEE; 2006.
8. Cortex-A7 processor. https://developer.arm.com/products/processors/cortex-a/cortex-a7 [09 March 2017]; 2011.
9. Cortex-A15 processor. https://developer.arm.com/products/processors/cortex-a/cortex-a15 [09 March 2017]; 2011.
10. Myslewski R. Deep inside ARM's new Intel killer. http://www.theregister.co.uk/2011/10/20/-details_on_big_little_processing/ [20 October 2016]; 2011.
11. Alcaide JCS, Hernán JC, Juste AS, Setoain J. PMCTrack. https://pmctrack.dacya.ucm.es/ [15 January 2018]; 2018.
12. MultiBench - Performance Suite for Multicore Architectures. http://eembc.org/multibench/index.php [30 November 2017]; 2017.
13. Sullivan GJ, Ohm J, Han WJ, Wiegand T. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*. 2012;22(12):1649–1668.
14. Rodríguez-Sánchez R, Quintana-Ortí ES. Architecture-Aware Optimization of an HEVC decoder on Asymmetric Multicore Processors. *CoRR*. 2016;abs/1601.05313.
15. x265 HEVC Encoder. https://bitbucket.org/multicoreware/x265/wiki/home [16 August 2017]; 2013.
16. YUV Video Sequences. http://trace.eas.asu.edu/yuv/ [5 september 2017]; 2017.
17. Ultra Video Group Test Sequences. http://ultravideo.cs.tut.fi/#testsequences [1 September 2017]; 2013.
18. Anastasopoulos A. A comparison between the sum-product and the min-sum iterative detection algorithms based on density evolution. In: :1021–1025IEEE; 2001.
19. MacKay DJC, Neal RM. Near Shannon limit performance of low density parity check codes. *Electronics letters*. 1996;32(18):1645.
20. Kanur S. LDPC Decoding Application. https://gitlab.abo.fi/aaust/stream/org.abo.preesm.ldpc/tree/master/Code [29 January 2018]; 2018.

How to cite this article: S. Stepanovic, G. Georgakarakos, S. Holmbacka, J. Lilius (2018), Quantifying the Interaction Between Structural Properties of Software and Hardware in the ARM big.LITTLE Architecture applied to evaluate HadGEM3-GC2, *Concurrency and Computation: Practice and Experience*, *2019;00:1–6*.