Elsevier Editorial System(tm) for Journal of Parallel and Distributed Computing
Manuscript Draft

Manuscript Number: JPDC-18-112R2

Title: On the Maturity of Parallel Applications for Asymmetric Multi-Core Processors

Article Type: VSI: HeterogeneousSystems

Keywords: parallel programming;
scheduling;
runtime systems;
asymmetric multi-cores;
hpc

Corresponding Author: Miss kallia chronaki,

Corresponding Author's Institution: Barcelona Supercomputing Center

First Author: kallia chronaki

Order of Authors: kallia chronaki; Miquel Moreto, PhD; Marc Casas, PhD; Alejandro Rico, PhD; Rosa M Badia, PhD; Eduard Ayguade, PhD; Mateo Valero, PhD

Abstract: Asymmetric multi-cores (AMCs) are a successful architectural solution for both mobile devices and supercomputers.
By maintaining two types of cores (fast and slow) AMCs are able to provide high performance under the facility power budget.
This paper performs the first extensive evaluation of how portable are the current HPC applications for such supercomputing systems.
Specifically we evaluate several execution models on an ARM big.LITTLE AMC using the PARSEC benchnark suite that includes representative highly parallel applications.
We compare schedulers at the user, OS and runtime levels, using both static and dynamic options and multiple configurations, and assess the impact of these options on the well-known problem of balancing the load across AMCs.
Our results demonstrate that scheduling is more effective when it takes place in the runtime system level as it improves the baseline by 23%, while the heterogeneous-aware OS scheduling solution improves the baseline by 10%.

***Cover Letter**

Dear Editor(s),

We are delighted to submit our manuscript in the Special Issue of JPDC on “Trends on Heterogeneous and Innovative Hardware and Software Systems”. Our work entitled “On the Maturity of Parallel Applications for Asymmetric Multi-Core Processors” is a thorough evaluation study about high performance applications and asymmetric multi-core systems.

Our work examines if current high performance applications are portable and efficient when moving to asymmetric mutli-core systems.
We then perform an extensive evaluation of different scheduling models such as application level scheduling, runtime level scheduling and OS level scheduling and find out what is the most appropriate way in order to utilize such heterogeneous systems.
Our findings show that the runtime system offers the most flexible and efficient environment to handle resource allocation or parallel applications.

We hope that you will find our research and contributions interesting and valuable.

Yours sincerely,
The Authors

We would like to thank the Reviewers for their valuable comments to improve the quality of this work. Below we answer the specific comments of each reviewer and we also point out the changes that we performed in our paper following their suggestions. We have updated the Figures 3, 4, 5, 6 to increase readability.

Reviewer #1:

1. *Based on the description of the A-7 and the A-15 core in Section 2 it seems like the smaller A-7 core has a larger L1 associativity (4 way) than the bigger A-15 core core (2 way) although both have the same cahe capacity. Is this the case?*
    - Yes, the L1 associativity of the A7 core is 4-way associative and the associativity of the A15 core is 2-way. This information can be found in the technical reference manuals of these architectures in the following links: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464d/DDI0464D_cortex_a7_mpcore_r0p3_trm.pdf http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438c/DDI0438C_cortex_a15_r2p0_trm.pdf
2. *Although Section 3 introduces three different OS-level scheduling strategies the evaluation only compares against GTS scheduling. It would be helpful if the static threading approach presented in the evaluation is also introduced/discussed in this section. In static threading are the threads pinned to the cores? How does static threading work in the case of applications that utilize custom thread pool implementation?*
    - We have updated the text and have added subsection 3.4 within Section 3 to describe static threading in more detail.
3. *It would be interesting if the perf ratio (in Table 1) is also computed assuming the same frequency for the big and the little core. It would help attribute the improvement achieved big core over the little core to the clock frequency and to the core micro-architecture.*
    - This is indeed an interesting experiment. The table below shows the obtained ratios with frequency=1200MHz on both big and little cores. As shown on this table, the performance ratio is indeed affected by the frequency, but there is still performance difference even when they run on the same frequency.

App	Perf-paper	Perf same freq
Blackscholes	2,18	1,44
Bodytrack	4,16	2,12
Canneal	1,73	1,13
Dedup	2,67	1,46
Facesim	3,40	1,9
Ferret	3,59	1,84
Fluidanimate	3,32	1,63
Streamcluster	3,48	2,11
Swaptions	2,78	1,78

4. *In Figure 3, 4+0 and 0+4 configurations on average provide similar speedups irrespective of the scheduling strategy. However 4+0 configuration provides improvement with task based scheduling specifically for bodytrack and fluidanimate. Why is this the case?*
    - The task-based implementation of bodytrack is highly optimized to perform better than the pthreads implementation of the benchmark by overlapping communication with computation. This is because the task-based approach allows more programming flexibility by introducing tasks and task dependencies. The details of this implementation can be found in the paper by Chasapis et.al. "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite". We have updated the text in paragraph 7 of Section 5.1 to explain this.
    - In the case of fluidanimate there was a typo in the data producing the charts which is now corrected. We have updated the chart on Figure 3 with the correct values. As a data-parallel application, fluidanimate does not improve much with the task based model for the 0+4 configuration.
5. *Although the performance of the three scheduling strategies are similar for many applications assuming a homogeneous 4+0 configuration, the average power consumption for task-based is comparatively much higher (blackscholes, bodytrack, dedup, facesim) . Is it because task-based scheduling is not as efficient as other static threading for symmetric multicore configurations? If so, why?*
    - The reason that for some applications the power consumption with the task-based solution is higher than the other approaches is that the task-based approach utilizes the big cores more effectively. The runtime system scheduling in the task-based approach leads to faster scheduling decisions, thus less idle time of the big cores. For that reason the power dissipation of the big cores with the task-based approach is observed to be higher. Less idle time of the cores means faster processing, thus increase in performance.
6. *The static threading results are discussed in detail in Section 5.1. However the results for GTS and task-based scheduling strategies are not discussed to the same extent. For instance, why is GTS successful in exploiting 2+2 configuration only for facesim, fluidanimate, streamcluster and swaptions and not for others?*
    - We agree that we should add more explanation regarding the GTS results. Specifically the reason that GTS is successful in exploiting the 2+2 configuration for these applications is that GTS is dynamically moving the threads around the cores depending on the CPU utilization. From this, it is expected that GTS will generally perform better than static threading for the asymmetric configurations (this is the reason it is designed for). However, for ferret that is an application with highly sophisticated parallelization static threading can achieve equally good results for the asymmetric configuration, avoiding the overhead of thread switching among the CPUs. On the other hand, canneal is an application that is by default memory intensive and with a low performance ratio, thus it is by definition hard for an OS scheduler that performs context switching to increase performance. The rest of the applications have the expected behavior for GTS which is to increase performance for asymmetric systems. We have added paragraph 7 in Section 5.1 to explain these scenarios.

7. *The last paragraph in Section 5.1 makes an observation about static threading and its limitation in the context of asymmetric configurations. IMHO based on the results this summary could also mention that static threading is the best approach in case of homogeneous configurations.*
    - We agree that the average performance results of the applications indicate that all approaches have similar performance and static threading achieves the lowest energy consumption, which makes it the most successful approach for symmetric configurations. However, there are applications like bodytrack, dedup and fluidanimate where more sophisticated solutions like GTS and task based perform better even for the homogeneous configurations. In addition, taking into account the energy and EDP results, we can see that on average static threading achieves similar results to the other approaches for the homogeneous configurations, thus it would be unfair to state that it is the best approach for such cases.
8. *In Section 5.2 it is mentioned that the average improvement is 15% over the symmetric configuration when 4 extra cores are added. This improvement however seems much smaller than what is indicated in Figure 2. Why is there such a considerable gap between the ideal and the actual improvement?*
    - The ideal speedup reported in Figure 2 is an ideal and practically unachievable speedup from all the applications. This speedup is theoretical and assumes that there are no inter-task dependencies neither synchronization points or memory overheads. Thus it is expected that the real evaluation results cannot reach the performance reported in Figure 2. The reason for adding this chart in the paper is so that we can give to the reader a theoretical ideal performance for each application on this platform, irrespective of implementation or scheduling mechanism used. We have added a brief explanation in order to demonstrate the usefulness of this chart in the second to last paragraph of Section 4.2.
9. *The energy results seem to indicate that it is inherently energy inefficient to keep the big cores busy and it is always better from the energy standpoint to carry out as much work as possible on the small cores. Why is this the case?*
    - This is correct. The design and layout of the little cores has been optimized for power efficiency while the design of the big cores targets higher performance levels at the cost of less energy-efficiency. Experimentally, the power dissipation of little cores is as much as 13 times less than the power dissipation of big cores (0.1W for 1 little vs 1.3W for 1 big, for bodytrack). As a consequence, approaches that lead to higher utilization of the big cores achieve better performance at the cost of less energy efficiency. We have added a comment in the last paragraph of Section 5.2 to highlight this insight.
10. *Although streamcluster and swaptions are grouped in a similar bin the results presented in Figure 7 indicate that 4+4 configuration provides improvement compared to 4+0 configuration only for swaptions and not for streamcluster. Moreover the improvement over static for the two other scheduling strategies is considerable for swaptions than for streamcuster. What can thi be attributed to?*
    - Swaptions shows an increased improvement with GTS and task-based solutions compared to streamcluster. This is because of the implementation and the nature of the two

applications. The task graph of streamcluster presents multiple parallel regions that are spawned and synchronized. Due to the multiple synchronization points, GTS and task-based cannot increase performance of streamcluster as much. Swaptions on the other hand, that is also a data-parallel application has less synchronization points, thing that allows GTS and task-based to exploit asymmetry. We have updated the text and have added this information on the 7th paragraph of Section 5.2, page 17.

11. *In general it would help the reader to follow the discussions in detail in Section 5.1 and 5.2 if there were individual subsections discussing the results for each scheduling strategy.*
    - Indeed his suggestion would help the reader, so we added one paragraph after section 5.2 titled as *Discussion* that summarizes the findings for each scheduling approach.
12. *To my understanding static-threading and loop-static both divide work statically between the threads without taking the heterogeneity of the system into account. What is it that causes Loop-static to perform considerably better than static-threading for 4+x configurations (when x>=1).*
    - It is true that loop-static theoretically performs the same scheduling as static threading. However there is an important difference. Loop-static does not pin the threads to cores but allows the GTS scheduler to migrate threads among cores. This is the default option for the OpenMP programming model. As we used the default version of it, the programming model's execution leaves the freedom to the OS scheduler to move threads to different types of cores. We have updated the text of the paper in the 5th paragraph of Section 5.3 to explain this.
13. *It is indicated in the discussion that loop-dynamic is more efficient on coarse grained parallel applications than task-based scheduling. Why is this the case?*
    - We agree that the text is making a generalization about the behavior of task-based and loop-dynamic approaches depending on the task granularity, without enough evidence to support this statement. This motivated us to research on the performance of swaptions and investigate more thoroughly why the behavior indicates loop-dynamic to perform more efficiently than task-based. During this process, we had to re-evaluate the performance of this application after regenerating the binaries. The results with the updated binary files showed that loop-dynamic and task-based have the same performance, thus we updated the data on the chart of Figure 9. The latest results make sense, since the implementation of the two approaches practically does not differ, as they both generate the same number of tasks and assign them dynamically to the available threads. We have updated the text according to the new findings in Section 5.3.

Minor Comments

1. *It would be good to have the average speedup numbers in Figure 2.*
    - We have added the average speedup numbers in Figure 2.
2. *It would help the reader of figure 3,4 and 5 are placed on the same page.*
    - We have updated the figures' placing in the current draft.

Reviewer #3:

1. *Findings are not very insightful. „It is fairly obvious that an application that is optimized for running on a homogeneous multi-core will run poorly on an asymmetric multi-core. Also, it is expected that a task based implementation „ which automatically schedules new tasks when others complete „will be a better fit for such architectures. The paper needs to add insight beyond this observation. For example, how do different task-based approaches perform? This is a much more interesting question as you would then compare approaches which one would expect to perform well.*
    - One very important insight of this paper is that even though the state-of-the-art solutions for asymmetric systems suggest scheduling in the OS level (GTS), this is not the optimal. This paper provides the very important insight that scheduling should take place in the runtime system. Comparing different scheduling policies within the runtime system is also very interesting and our paper introduces a related study on section 5.3, but further research on runtime level scheduling is out of the scope of this specific paper.
      In addition, even if it is expected for a task based implementation to be a better fit for these architectures, there is currently no study quantifying it on a real system, which makes our work novel. Quantifying such results requires a big effort in characterizing all parts of the evaluation including the applications as well as the scheduling approaches.
      We have added some of these insights in the conclusions section of the paper.
2. *The energy/power/EDP analysis is confusing. It seems that when performance goes up, power consumption goes up. When performance goes down, power consumption goes down. This makes sense as higher performance means the cores are working harder which results in more switching and high power. Energy is generally proportional to the amount of work to be done (i.e., instructions in the program). Is this intuition supported by your results? Please explain.*
    - The intuition of the reviewer is correct. The design and layout of the little cores has been optimized for power efficiency while the design of the big cores targets higher performance levels at the cost of less energy-efficiency. Experimentally, the power dissipation of little cores is as much as 13 times less than the power dissipation of big cores (0.1W for 1 little vs 1.3W for 1 big, for bodytrack). As a consequence, approaches that lead to higher utilization of the big cores achieve better performance at the cost of less energy efficiency. We have added a comment in the last paragraph of Section 5.2 to highlight this insight.
3. *The authors do not state clearly what were the main results of the experiments. Currently, they present a lot of numbers, but it is unclear what the key findings are and how the numbers back up these findings.*
    - As suggested by the reviewer, we have summarized the main insights of this study in the conclusions of the paper.
4. *The introduction does not indicate what are the root causes of the poor performance of current OS and runtime schedulers and how the runtime system approaches can overcome these issues (second to last paragraph). Foreshadowing the main findings in the introduction would make the paper much easier to read*

- We have updated the introduction in paragraphs 4 and 5 to indicate the causes of poor performance.

5. *The authors go into too much detail on the platform in Section 2. This section should only include the details that are needed to understand the results, and the authors can refer to the technical documentation of the platform for further details.*
    - We have shortened this section to include only the most important details of the platform and we provide the appropriate design document references.
6. *The authors fix the frequency of the cores to avoid overheating. Did you consider mounting a heatsink and possibly a fan? Static power depends on temperature so controlling temperature is critical to get consistent power measurements. Also, I'm concerned that (arbitrarily) fixing the frequencies of the big and small cores may affect the performance of different scheduling approaches. Intuitively, I would expect that the bigger the performance difference between the cores, the better the TBP approach will perform compared to the other approaches. Some sensitivity analysis on this issue should be added.*
    - Our systems are using heatsinks and fans in order to maintain their temperature. Still though, performing real experiments for a long time can cause overheating. Setting the core frequency is essential for our study not only to avoid overheating, but also to make sure that the reported performance is not affected by changes in the CPU frequency due to the DVFS governor. If we do not fix the cpu frequency, the running DVFS governor would change the frequency of each core dynamically and this is out of the runtime system or the GTS control. We have modified the text in the first paragraph of Section 4.1 to explain this decision.

      Performing experiments to address the impact of the governor is out of the scope of this paper. This sensitivity analysis is indeed interesting. We have added it as future work in the last paragraph of the conclusions of this paper.
7. *The authors state that they report the average over five runs, but they don't report the average variability (e.g., standard deviation). Please report this.*
    - Due to the large amount of data reported in this paper, adding the stdev or min and max values on the charts for each bar significantly reduces the visibility of the charts.
    - This table shows the max stdev for the speedup of task-based, GTS and static threading among all applications:

-	- MAX	- MEDIAN
- Task based	- 0,75	- 0,03
- GTS	- 1,11	- 0,05
- Static threading	- 0,53	- 0,02

    - We have modified the text to report the stdev for our results in the second paragraph of Section 4.1.
8. *Power measurements are collected online and may interfere with the running process. Does this affect all techniques equally? Please report how performance differs when power measurement is enabled vs. disabled.*
    - Our experimental methodology for the energy is verified to be adding negligible overhead in the execution of the application. We have verified that the additional overhead of the

measuring daemon was always less than 3%. More specifically, in most cases performance was not affected and for some cases there was an overhead of around 1-2%. These negligible values are not enough to alternate the outcomes of this study. We have clarified these overheads in the third paragraph of Section 4.1.

9. *Why do you normalise to four cores and static threading? Is this the configuration that consumes most energy or is there some other reason? Please explain.*
    - This normalization is just to help the reader to have a performance reference. We use the specific reference as it is the less aggressive one. Using this reference we assess the impact on the energy consumption by the increase of the big cores. Looking at the increase of energy on top of the configuration using four little cores we can easily see how much energy is increased by adding big cores. We have updated our explanation for this choice in the last paragraph of Section 4.1.
10. *The labelling of the figures is confusing (e.g., Figure 2). I would prefer to have the number of little cores on one line and big cores on the other line -- all clearly labeled. Another option is to consistently use the B+L labelling the authors introduce. The key issue is that it should be possible to understand the figure without reading the explanation in the text.*
    - We have updated Figure 2 in the draft to show the number of big cores on the top line and the number of little cores on the bottom line. We have also included the average results, as was suggested by Reviewer 1.
11. *The authors use a lot of space for introducing PARSEC, but most readers will be familiar with it. This discussion should be shortened.*
    - We have updated the description of the applications to take into account the comment of the reviewer. Due to repetition with other publications, we eliminated the description of each application on Table 1.
12. *Figure placement needs to be improved. A lot of figures are placed quite far away from where they are discussed which reduces readability.*
    - We have updated our figure placement in the current draft so that the figures and the text describing them are closer.
13. *The argument for choosing GTS over CS and IKS (footnote on page 12) is weak. Sometimes, less advanced techniques are better than more advanced techniques for non-intuitive reasons. It would have liked to see an experiment which shows that you are in fact comparing to the best performing Linux scheduler.*
    - This is the currently supported scheduler in our board so it was not feasible to use another approach. However, it is considered to be the most efficient for asymmetric systems and it is the only one with which we can have a fair comparison against task-based. For example, cluster switching allows only up to 4 cores to be in use and the resulting system is always homogeneous. This would not allow us to evaluate the scheduling approaches on an asymmetry-based environment. IKS is also limited, as it allows only up to 4 cores to execute simultaneously. We (will) add this in the text to explain these reasons in addition to the fact that GTS is the most efficient.
14. *It sounds a bit strange that the SMC with the four small cores is the most energy efficient configuration (final paragraph, Sec. 5.2), given that some applications get a significant speed-up*

when moving to the big cores. Does the actually energy consumption increase faster than the speed-up? Please explain.

- In the case of little cores, their design and layout has been optimized for power efficiency. Opposed to this, big cores are designed to target higher performance levels at the cost of higher energy consumption. As a consequence, approaches that lead to higher utilization of the big cores achieve better performance at the cost of less energy efficiency. The comment that we have added in the last paragraph of Section 5.2 to highlights this insight.

***Highlights (for review)**

- We evaluate scheduling of high performance applications on asymmetric multi-cores
- We compare three scheduling approaches taking place on different levels of the software stack
- We analyze the results in terms of: performance, energy consumption and power
- We find that the runtime system is the most appropriate level for scheduling on asymmetric multicores
- We investigate different runtime scheduling approaches (e.g. loop vs task-based)

On the Maturity of Parallel Applications for Asymmetric Multi-Core Processors

Kallia Chronaki*[1], Miquel Moreto*, Marc Casas*, Alejandro Rico†, Rosa M. Badia*‡, Eduard Ayguadé*, Mateo Valero*

*Barcelona Supercomputing Center,
†ARM,
‡Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC)

Abstract

Asymmetric multi-cores (AMCs) are a successful architectural solution for both mobile devices and supercomputers. By maintaining two types of cores (fast and slow) AMCs are able to provide high performance under the facility power budget. This paper performs the first extensive evaluation of how portable are the current HPC applications for such supercomputing systems. Specifically we evaluate several execution models on an ARM big.LITTLE AMC using the PARSEC benchnark suite that includes representative highly parallel applications. We compare schedulers at the user, OS and runtime levels, using both static and dynamic options and multiple configurations, and assess the impact of these options on the well-known problem of balancing the load across AMCs. Our results demonstrate that scheduling is more effective when it takes place in the runtime system level as it improves the baseline by 23%, while the heterogeneous-aware OS scheduling solution improves the baseline by 10%.

Keywords: parallel programming, scheduling, runtime systems, asymmetric multi-cores, hpc

[1]**Corresponding authors:**
kallia.chronaki@bsc.es (Kallia Chronaki), miquel.moreto@bsc.es (Miquel Moreto), marc.casas@bsc.es (Marc Casas), alejandro.rico@arm.com (Alejandro Rico)

1. Introduction

The future of parallel computing is highly restricted by energy efficiency [1]. Energy efficiency has become the main challenge for future processor designs, motivating prolific research to face the *power wall*. Using heterogeneous processing elements is one of the approaches to increase energy efficiency [2, 3]. Asymmetric multi-core (AMC) systems is an interesting case of heterogeneous systems to utilize for energy efficiency. These systems maintain different types of cores that support the same instruction-set architecture. The different core types are designed to target different (performance or power) optimization points [4, 5, 6].

AMCs have been mainly deployed for the mobile market. Mobile processors are also utilized in HPC platforms aiming to energy savings [7]. Asymmetric mobile SoCs combine low-power simple cores (*little*) with fast out-of-order cores (*big*) to achieve high performance while keeping power dissipation low. Another area where AMCs have been successful is the supercomputing market. The Sunway TaihuLight supercomputer topped the Top500 list in 2016 using AMCs. In this setup, big cores, that offer support for speculation to exploit Instruction-Level Parallelism (ILP), run the master tasks such as the OS and runtime system. Little cores are equipped with wide Single Instruction Multiple Data (SIMD) units and lean pipeline structures for energy efficient execution of compute-intensive code.

Like in other heterogeneous systems, load balancing and scheduling are fundamental challenges that must be addressed to effectively exploit all the resources in AMC platforms [8, 9, 10, 11, 12, 13]. Mobile applications rely on multi-programmed workloads to balance the load in the system, while supercomputer applications rely on hand-tuned code to extract maximum performance. However, these approaches are not always suitable for general-purpose parallel applications.

In this paper, we evaluate several execution models on an AMC using the PARSEC benchmark suite [14]. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We quantify the performance loss of executing the applications *as-is* on all cores in the system. **R3Q4:** The main challenge on executing these applications on an AMC is to maintain load balance. These applications were originally developed on homogeneous platforms and typically operate by dividing the workload on even units. Executing these equal work units on an asymmetric system is expected to suffer due to load

imbalance.

To overcome this matter, we consider two possible solutions at the OS and runtime levels to exploit AMCs effectively. The first solution delegates scheduling to the OS. We evaluate the built-in heterogeneity-aware OS scheduler currently used in existing mobile platforms that automatically assigns threads to different core types based on CPU utilization. **R3Q4:** The main drawback of this approach is the overhead introduced by the thread migration, thus resulting in limited performance no matter the potential of the underlying system.

The second solution is to transfer the responsibility to the runtime system so it dynamically schedules work to different core types based on work progress and core availability. We evaluate the impact of using an inherently load-balanced execution model such that of task-based programming models. Recent examples [15, 16, 17, 18, 19, 20, 21, 22, 23] include clauses to specify inter-task dependencies and remove most barriers which are the major source of load imbalance on AMCs. Another approach of scheduling in the runtime system is to change the existing statically-scheduled work-sharing constructs for the applications implemented in OpenMP to use dynamic scheduling.

This paper provides the first to our knowledge comprehensive evaluation of representative parallel applications on a real AMC platform: the Odroid-XU3 development board with ARM big.LITTLE architecture. We analyze the effectiveness of the aforementioned scheduling solutions in terms of performance, power and energy. We show why parallel applications are not ready to run on AMCs and how OS and runtime schedulers can overcome these issues depending on the application characteristics. Further we point out in which aspects the built-in OS scheduler falls short to effectively utilize the AMC. Finally, we show how the runtime system approach overcomes these issues, and improves the OS and static threading approaches by 13% and 23% respectively.

The rest of this document is organized as follows: Section 2 describes the evaluated AMC processor, while Section 3 provides information on scheduling at the OS and runtime system levels. Section 4 describes the experimental framework. Section 5 shows the performance and energy results and associated insights.Finally, Section 6 discusses related work and Section 7 concludes this work.

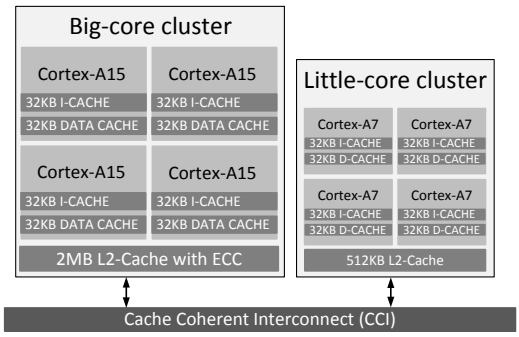

Figure 1: Samsung Exynos 5422 processor with ARM big.LITTLE architecture.

2. The ARM big.LITTLE Architecture

R3Q5: We have shortened this section to include the most important details of the platform and providing the appropriate design references.

The ARM big.LITTLE [24, 10] is a state-of-the-art AMC architecture that has been successfully deployed in the mobile market. ARM big.LITTLE combines simple in-order cores (little) with aggressive out-of-order cores (big) in the same System-on-Chip (SoC) to provide high performance and low power. *Big* and *little* cores support the same instruction set architecture (ISA) so they can run the same binaries and therefore are easily combined within the same system. The little cores in a big.LITTLE system are designed targeting energy efficiency, while big cores are designed for high performance. Current cores implementing the ARMv7-A and ARMv8-A ISA support big.LITTLE configurations.

In this work, we use of one of the commercially available development boards featuring a big.LITTLE architecture: the Hardkernel Odroid-XU3 development board. As shown in Figure 1, the Odroid-XU3 includes an 8-core Samsung Exynos 5422 chip with four ARM Cortex-A15 cores and four Cortex-A7 cores. In this SoC, there are two core clusters with caches that are shared among the cores of each cluster [25]. The four Cortex-A15 share a 2 MB 16-way 64-byte-cache-line L2 cache, while the Cortex-A7 cores share a 512 KB L2 cache. A single cache coherent memory controller (CCI) provides access to RAM to both clusters. The reason we use this platform instead of the more up-to-date Juno platform [26] is that even if the latter features the more advanced Cortex A53 [27] and Cortex A57 [28] cores, it is limited to six cores instead of the 8 cores in Odroid-XU3.

The Cortex-A7 cores in this SoC support dual-issue of instructions and

their pipeline length is between 8 and 10 stages. The L1 instruction cache is 32KB two-way set associative, with virtually indexed and physically tagged cache-lines that can hold up to 8 instructions. The L1 data cache is four-way set associative with physically-indexed and physically-tagged cache lines and uses a pseudo-random replacement policy [29]. Dynamic Voltage and Frequency Scaling (DVFS) techniques adjust the frequency of the little cores from 200MHz up to 1.4GHz.

The Cortex-A15 cores in this SoC support triple-issue of instructions and their pipeline length is between 15 and 24 stages [30]. The L1 instruction and data caches of the Cortex-A15 are both 32 KB and 2-way set-associative with 64 byte cache lines [31]. DVFS techniques vary the frequency of the big cores from 200 MHz up to 2 GHz. For the rest of the paper, we refer to Cortex-A15 cores as *big* and to Cortex-A7 cores as *little*.

3. Scheduling in Asymmetric Multi-Cores

Scheduling a set of processes on an AMC system is more challenging than the traditional process scheduling on SMCs. An efficient OS scheduler has to take into account the different characteristics of the cores and act accordingly [32]. There have been three mainstream OS schedulers for ARM big.LITTLE systems: *cluster switching*, *in-kernel switch* and *global task scheduling*, described in the next sections. In the case of parallel applications, *dynamic scheduling at the runtime system level* can be exploited to balance the workload among the different cores and is described in section 3.3. **R1Q2:** Finally, when all the scheduling mechanisms are disabled, the thread scheduling decisions are based on the *static application-level scheduling*.

3.1. Cluster Switching and In-Kernel Switch

In the Cluster Switching (CS) approach [24], only one of the clusters is active at any given time: either the cluster with little cores or the cluster with big cores executes. Thus, the OS scheduler operates on a *de-facto* symmetric multi-core with only four cores, namely the cores of the current active cluster. The policy to change the operating cluster is based on CPU utilization. When idle, background processes are executed on the little cores. When CPU utilization surpasses a threshold, all processes (foreground and background) are migrated to the big cluster. When running on the big cluster, if CPU utilization decreases below a given lower threshold, the entire workload is moved to the little cluster.

In the In-Kernel Switch (IKS) approach [33], each little core is paired with a big core and it is seen as a single core. On idle, background processes are run on little cores. When the CPU utilization on a given little core surpasses a threshold, the execution on that core is migrated to the big core. When the CPU utilization decreases on that big core below a given threshold, the execution migrates to the associated little core. Thus, at the same time, little and big cores may co-execute, but only one of each pair is active at a given point in time, effectively exploiting just half of the cores concurrently. For both CS and IKS, an enhanced `cpufreq` driver manages the switching within each core pair.

3.2. Global Task Scheduling

The Global Task Scheduling (GTS) [24] allows running applications on all cores in the asymmetric multi-core. In GTS, all cores are available and visible to the OS scheduler, and this scheduler is aware of the characteristics of the core types. Each process is assigned to a core type depending on its CPU utilization: high CPU utilization processes are scheduled to big cores and low CPU utilization processes to little cores. GTS also migrates processes between big and little cores when their CPU utilization changes. As a result, cores are active depending on the characteristics of the workload.

The key benefit of GTS is that it can use all the cores simultaneously, providing higher peak performance and more flexibility to manage the workload. In GTS tasks are directly migrated to cores without needing the intervention of the `cpufreq` daemon, reducing response time and minimizing the overhead of context switches. As a consequence, Samsung reported 20% improvement in performance over CS for mobile benchmarks [24]. Also, GTS supports clusters with different number of cores (e.g. with 2 big cores and 4 little cores), while IKS requires to have the same number of cores per cluster.

3.3. Dynamic Scheduling in the Runtime

Current programming models for shared memory systems such as OpenMP rely on a runtime system to manage the execution of the parallel application. In this work, we make use of two types of programming models: loop- and task-based. Loop-based scheduling distributes the iterations of a loop among the threads available in the system, following a traditional *fork-join* model. OpenMP supports loop-based scheduling through its *parallel for* directives.

This clause implies a barrier synchronization at the end of the loop[2], and supports either static or dynamic loop scheduling.

With static loop scheduling, the iterations of a loop are divided to as many chunks as the number of cores. Then, every core executes the assigned chunk, leading to a low-overhead static scheduling. In addition, OpenMP supports dynamic loop scheduling. It generates more chunks than cores, and assigns them to the available cores at runtime. This is more suitable to asymmetric multi-core systems where the cores are not similar and a static iteration assignment would cause load imbalance.

Recent advances in programming models recover the use of task-based programming models to simplify parallel programming of multi-cores [16, 17, 34, 19, 21]. In these models the programmer splits the code in sequential pieces of work (tasks) and specifies the data dependencies among them. With this information the runtime system schedules tasks and manages synchronization. These models ease programmability [16, 17, 34, 19, 21, 22], and also increase performance by avoiding global synchronization points.

To evaluate this approach we make use of OpenMP tasking support [16]. OpenMP allows expressing tasks and data dependences between them using equivalent code annotations. It conceives the parallel execution as a *task dependence graph* (TDG), where nodes are sequential pieces of code (tasks) and the edges are control or data dependences between them. The runtime system builds this TDG at execution time and dynamically schedules tasks to the available cores. Tasks become ready as soon as their input dependencies are satisfied. The scheduling of the ready tasks is done in a first-come-first-served manner, using a FIFO scheduler. Even though this scheduler is not aware of the task computational requirements or the core type and its performance and power characteristics, it can balance the load as long as there are ready tasks available thanks to the lack of global synchronization.

3.4. Static Scheduling in the Application Level

R1Q2: When the OS and runtime schedulers are disabled scheduling relies on the application. Current parallel applications generate software threads and rely on the operating system for the efficient mapping of these threads on the available cores. By disabling the operating system scheduler,

[2]unless specified otherwise with the `nowait` clause

Table 1: Benchmarks used from the PARSEC benchmark suite and their measured performance ratio between big and little cores

Benchmark	Input	Parallelization	Perf ratio
blackscholes	native	data-parallel	2.18
bodytrack	native	pipeline	4.16
canneal	native	unstructured	1.73
dedup	351 MB data	pipeline	2.67
facesim	native	data-parallel	3.40
ferret	native	pipeline	3.59
fluidanimate	native	data-parallel	3.32
streamcluster	native	data-parallel	3.48
swaptions	native	data-parallel	2.78

each created thread is pinned on one of the cores and it is not allowed to migrate to another hardware component. Each pinned thread executes the work that the application is assigning to it statically. In this scenario, the application is responsible for the efficient parallelization, as it only depends on how the application is statically assigning the work on each of the software threads.

To evaluate this approach, we make use of applications that are implemented using the pthreads model. We then modify the code of the application so that each created thread is pinned to the next available processor. This way, threads are statically assigned to processors and the operating system is not allowed to modify this.

4. Experimental Methodology

4.1. Metrics

All the experiments in this paper are performed on the Hardkernel Odroid XU3 described in Section 2. **R3Q6:** In our experiments we keep the frequencies of the cores static. This is in order to first, avoid machine overheating, and second to prevent the DVFS governor to dynamically modify the frequency of the cores during runtime, thing that would affect the reliability of the results. We make use of the `cpufreq` driver to set big cores at 1.6GHz and little cores at 800MHz.

We evaluate seven configurations with different numbers of *little* (L) and *big* (B) cores, denoted `L+B`. For each configuration and benchmark, we report the average performance of five executions in the application parallel region. Then, we report the application speedup over its execution time on one little core. **R3Q7:** The variability of the results among the five runs is very small

and the stdev of the speedup obtained ranges from 0 to 1.1, averaging at 0.85. Equation 1 shows the formula to compute this speedup.

$$\text{Speedup(L, B, method)} = \frac{\text{Exec. time(1, 0, method)}}{\text{Exec. time(L, B, method)}} \quad (1)$$

In this platform, there are four separated current sensors to measure, in real time, the power consumption of the A15 cluster, the A7 cluster, the GPU and DRAM. To gather power and energy measurements, a background daemon reads the machine power sensors periodically during the application execution with negligible overhead. Sensors are read at their refresh rate, every 270ms, and the values of A7 and A15 clusters' sensors are collected. With the help of timestamps, we correlate the power measurements with the application parallel region in a *post-mortem* process. **R3Q8:** The execution time overhead of the running daemon is measured and verified to be less than 3%. [3] The reported power consumption is the average power tracked during five executions of each configuration, considering the application parallel region only. We then report average power in Watts along the execution.

Finally, in terms of energy and Energy Delay Product (EDP), we report the total energy and EDP of the benchmarks region of interest. **R3Q9:** To facilitate the explanation of these results and isolate the impact of the different system configurations on the energy consumption we normalize these results to the run on four little cores with static threading. Equations 2 and 3 show the formulas for these calculations.

$$\text{Normalized Energy(L, B, method)} = \frac{\text{Energy(L, B, method)}}{\text{Energy(4, 0, static-threading)}} \quad (2)$$

$$\text{Normalized EDP(L, B, method)} = \frac{\text{EDP(L, B, method)}}{\text{EDP(4, 0, static-threading)}} \quad (3)$$

4.2. Applications

With the prevalence of many-core processors and the increasing relevance of application domains that do not belong to the traditional HPC field, comes the need for programs representative of current and future parallel workloads. The PARSEC benchmark suite [14, 35] features state-of-the-art, computationally intensive algorithms and very diverse workloads from different areas of computing. In our experiments, we make use of the original PARSEC

[3] In most cases this overhead is 1% with some applications reaching 2%.

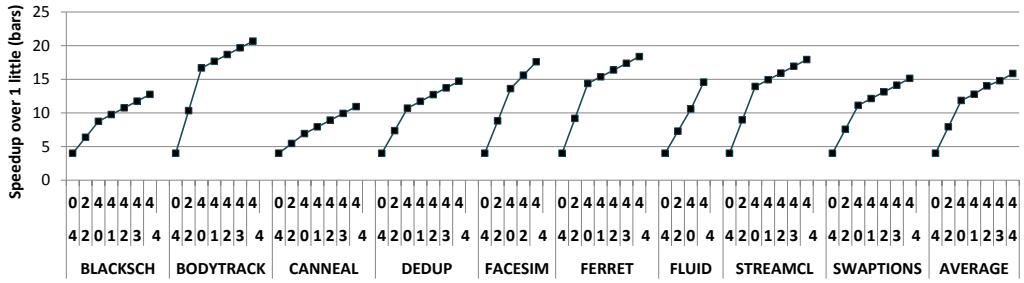

Figure 2: Ideal speedup over 1 little core according to Equation 4. Numbers at the bottom of x axis show the number of little cores, numbers above it show the number of big cores

codes together with a task-based implementation of nine benchmarks of the suite [36].

Table 1 shows the benchmarks included in the study along with their respective inputs, parallelization strategy and performance ratio between big and little cores per application. We are using native inputs, which are real input sets for native execution, except for `dedup`, as the entire native input file of 672 MB and the intermediate data structures do not fit in the memory system of our platform. Instead, we reduce the size of the input file to 351 MB.

The original codes make use of the `pthreads` parallelization model for all the selected benchmarks. The taskified applications follow the same parallelization strategy implemented with OpenMP 4.0 task annotations. The task-based implementation is done following two basic ideas: i) remove barriers where possible, by adding explicit data-dependencies; and ii) remove application-specific load balancing mechanisms, such as application-specific pools of threads implemented in `pthreads` and delegate this responsibility to the runtime.

When running on the big.LITTLE processor, each benchmark exhibits different performance ratios between big and little cores. These ratios tell us how many times faster a big core is compared to a little core. We measure the performance ratio of each application by executing it first on one big core and then on one little core, which corresponds to Speedup(0, 1, task-based) in Equation 1. Table 1 also includes the observed performance ratio for each application. Bodytrack is the application that benefits the most from running on the big core with a performance ratio of 4.16×. The out-of-order execution of the big core together with an increased number of in-flight

instructions significantly improves the performance of this application. In contrast, canneal is the benchmark with the lowest performance ratio, 1.73×, as this is a memory-intensive benchmark that does not benefit as much from the extra computation power of the big core. In general, performance ratios are above 2.5× for seven out of nine benchmarks, reaching 3.03× on average.

Taking into account these performance ratios, we can estimate the ideal speedup of the platform for each workload assuming a perfect parallelization strategy. **R1Q8:** This metric is useful for understanding the potential of each application irrespective of parallelization strategy and scheduling approach. It isolates the computations of each application and shows its ideal performance for each possible confuguration of the AMC. Equation 4 shows the equation for the ideal speedup over 1 little core computation according to the number of big (B) and little (L) cores.

$$\text{Ideal speedup(workload, B, L)} = \text{B} \times \text{Perf\_ratio(workload)} + \text{L} \tag{4}$$

Figure 2 shows the ideal speedup of the system for each application for the varying numbers of cores. This speedup assumes that the applications are fully parallel with no barriers or other synchronization points. Thus, these speedups are an upper bound of the achievable application performance.

5. Evaluation

We measure execution time, power, energy and EDP of nine applications from the PARSEC benchmark suite [35]. We compare these metrics for three different scheduling approaches:

- *Static threading*: scheduling decisions are made at the application level. The OS is not allowed to migrate threads between the clusters of big and little cores.
- *GTS*[4]: dynamic coarse-grained OS scheduling using the GTS scheduler integrated in the Linux kernel [24, 37] using the default PARSEC benchmarks.

[4]We choose to evaluate GTS instead of CS and IKS because it is the most advanced scheduling approach supported in the Linux kernel.

- *Task-based*: dynamic fine-grained scheduling at the runtime level with the task-based implementations of the benchmarks provided in PARSECSs [36].

5.1. Exploiting Parallelism in AMCs

This section examines the opportunities and challenges that current AMCs offer to emerging parallel applications. With this objective, we first evaluate a system with a constant number of four cores, changing the level of asymmetry to evaluate the characteristics of each configuration. In these experiments, all applications run with the original parallelization strategy that relies on the user to balance the application (*Static threading*). We also evaluate the OS-based dynamic scheduling (*GTS*) and the task-based runtime dynamic scheduling (*Task-based*) for the same applications. The system configurations evaluated in this section are: i) Four little cores (`0+4`); ii) Two big and two little cores (`2+2`); and iii) Four big cores (`4+0`)

For these configurations, Figure 3 shows the speedup of the PARSEC benchmarks with respect to running on a single little core. Figure 4 reports the average power dissipated on the evaluated platform. Finally, Figure 5 shows the total energy consumed per application for the same configurations. Energy results are normalized to the energy measured with four little cores (higher values imply higher energy consumptions). Average EDP results are also included in this figure.

Focusing on the average performance results, we notice that all approaches perform similarly for the homogeneous configurations. Specifically, applications obtain the best performance on the configuration *4+0*, with an average speedup of 9.5× over one little core. When using four little cores, an average speedup of 3.8× is reached for all approaches. This shows that all the approaches are effective for this core count. In the configuration *2+2*, *Static threading* slightly improves performance (5.0× speedup), while *GTS* and *Task-based* reach significantly higher speedups: 5.9× and 6.8×, respectively.

Contrarily, in terms of power and energy, the most efficient configuration is running with four little cores, as the performance ratio between the different cores is inversely proportional to the power ratio [10]. On average, all the approaches reach a power dissipation of 0.75W for the `0+4` configuration, while *Task-based* reaches 3.5W for the `4+0` configuration which is the one with the highest average power dissipation. In configuration `2+2`, average energy values for *Static threading* and *Task-based* are nearly the same, as

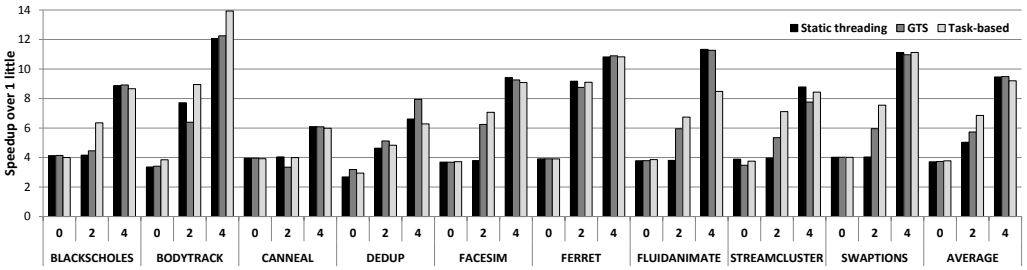

Figure 3: Execution time speedup over 1 little core for systems that consist of 4 cores in total with 0, 2 and 4 big cores. Different schedulers at the application (*static threading*), OS (*GTS*) and runtime (*task-based*) levels are considered.

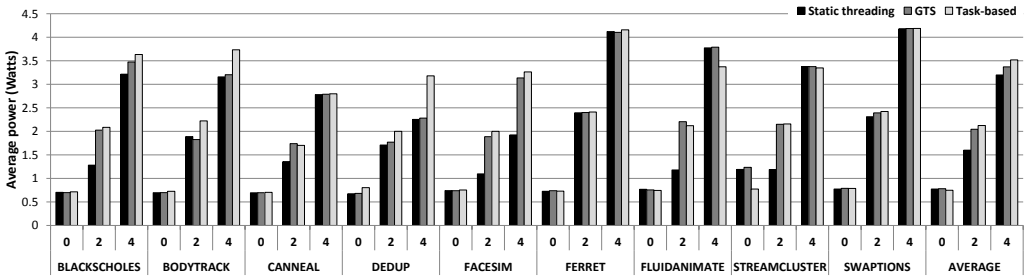

Figure 4: Average power measurements on a 4-core system with 0, 2, and 4 big cores.

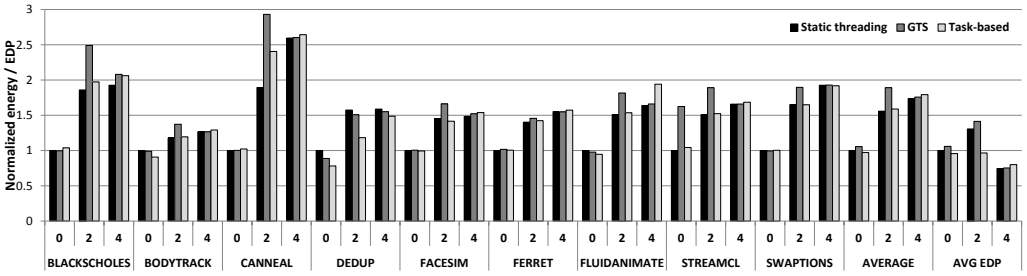

Figure 5: Normalized energy consumption and average EDP on a 4-core system with 0, 2, and 4 big cores. Static threading on 4 little cores is the baseline in both cases.

the increase in power from 1.6W to 2.1W is compensated by a significant improvement in performance of 30%.

Finally, in terms of EDP using the four big cores is the optimal, as the performance improvements compensate the increase in total energy. In con-

figuration 2+2, *Task-based* achieves the same EDP results as in 0+4, but with 81% better performance. For the asymmetric configuration, *Task-based* achieves the best performance-energy combination since its dynamic scheduling is effectively utilizing the little cores.

Next, we focus on the obtained results per benchmark. For applications with an extensive use of barriers (blackscholes, facesim, fluidanimate, streamcluster and swaptions) or with a memory intensive pattern (canneal), the extra computational power offered by the big cores in configuration 2+2 is not exploited. As a result with *Static threading* performance is only slightly improved by 1% on average when moving from 0+4 to the 2+2 configuration. This slight improvement comes at the cost of much more power and energy consumption (79% and 77% respectively). These results are explained three-fold: i) load is distributed homogeneously among threads in some applications; ii) extensive usage of barriers force big cores to wait until little cores reach the barrier; and iii) high miss rates in the last-level cache cause frequent pipeline stalls and prevent to fully exploit the computational power of big cores. To alleviate these problems, the programmer should develop more advanced parallelization strategies that could benefit from AMCs, as performed in the remaining applications, or rely on dynamic scheduling at OS or runtime levels.

R1Q6: GTS is a suitable alternative for barrier-synchronized applications (blackscholes, facesim, fluidanimate, streamcluster and swaptions) when asymmetry is introduced. GTS enhances performance as it is dynamically migrating the threads around the cores depending on the CPU utilization. Thus it is expected that performance will increase compared to static threading for the asymmetric configuration. For these applications, the *task-based* approach further improves GTS for the asymmetric configuration. This is because *task-based* schedules tasks among threads which is much more efficient than scheduling threads among cores.

The three remaining applications are parallelized using a pipeline model (bodytrack, dedup, and ferret) with queues for the data-exchange between pipeline stages and application-specific dynamic load balancing mechanisms designed by the programmer. As a result, *Static threading* with these applications benefits from the extra computational power of the big cores in the configuration 2+2. **R1Q6:** Since *Static threading* can already maintain load balance for these applications due to their implementation, there is no need for dynamic thread migration that is introduced by GTS. **R1Q4:** Using the *task based* approach, the code of the application is simplified allowing

the application to express even more parallelism as the runtime system automatically allows the overlapping of the different pipeline stages. This can be verified by the fact that bodytrack obtains higher performance with the *task based* approach even for the symmetric configurations. On the asymmetric configuration, *Task-based* further improves the obtained performance, reaching a 13% average improvement over *GTS*. Clearly, these applications benefit in performance by the increased number of big cores, while power and energy are increasing since the big cores are effectively utilized.

Generally, relying on the programmer to statically schedule asymmetric configurations does not report good results, as it is very hard to predict the system's behavior at application-level. Only applications that implement advanced features with user-level schedulers and load balancing techniques, can benefit from asymmetry, at the cost of programmability effort. Relying on the OS scheduler is a suitable alternative without code modifications, but relying on the runtime to dynamically schedule tasks on the asymmetric processor achieves much better performance, power and energy results.

5.2. Adding Little Cores to an SMC

In the following experiments, we explore if an application running on a symmetric multi-core (SMC) with big cores can benefit from adding small cores that help in its execution. Having more computational resources increases the ideal speedup a parallel application can reach, but it also introduces challenges at application, runtime and OS level. Thus, we examine how many small cores have to be added to the system to compensate the cons of having to deal with AMCs.

To evaluate this scenario, we explore configurations `4+0`, `4+1`, `4+2`, `4+3` and `4+4`. In these experiments, the number of big cores remains constant (four), while the number of little cores increases from 0 to 4. First we focus on the average results of speedup, power, energy and EDP, shown in Figure 6.

The speedup chart of Figure 6 shows that *Static threading* does not benefit from adding little cores to the system. In fact, this approach brings an average 6% slowdown when adding four little cores for execution (`4+4`). This is a result of the static thread scheduling; because the same amount of work is assigned to each core, when the big cores finish the execution of their part, they become idle and under-utilized. GTS achieves a limited speedup of 8% with the addition of four little cores to the `4+0` configuration. The addition of a single little core brings a 22% slowdown (from `4+0` to `4+1`) and requires three additional little cores to reach the performance of

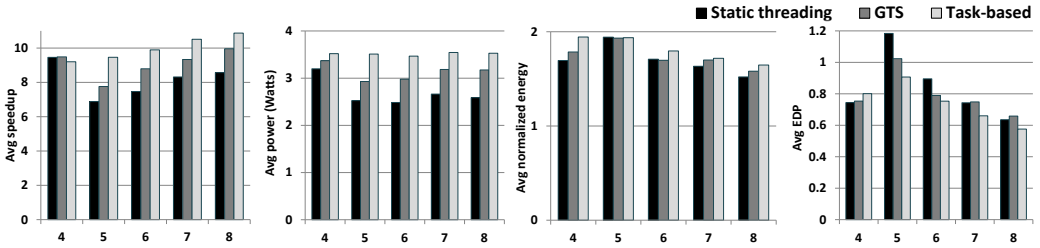

Figure 6: Average results when running on 4 to 8 cores with 4 of them big. Speedup is over 1 little core. Static threading on 4 little cores is the baseline of energy consumption and EDP

the symmetric configuration (`4+3`). Finally, the *Task-based* approach always benefits from the extra computational power as the runtime automatically deals with load imbalance. Performance improvements keep growing with the additional little cores, reaching an average improvement of 15% over the symmetric configuration when 4 extra cores are added.

The power chart of Figure 6 shows oppositional benefits among the three approaches. We can see that *Static threading* and *GTS* benefit from asymmetry, effectively reducing average power consumption. *Static threading* reduces power consumption when moving from the `4+0` to the `4+4` system by 23% while *GTS* does so by 6.2%. On the other hand, the *task-based* approach keeps the big cores busy for most of the time so it maintains the average power nearly constant.

The reduction in power, results to reduced average energy in the case of *Static threading* in configuration `4+4`, as shown on the energy chart of Figure 6. As discussed in Section 5.1, little cores are more energy efficient than big cores, at the cost of reduced performance. In all the approaches, at least two extra little cores are needed to reduce energy. In configuration `4+4`, energy is reduced by 14% for *Static threading*, 15% for *GTS*, and 16% for *Task-based*. Consequently, we can state that asymmetry reduces overall energy consumption.

To see the impact on both performance and energy efficiency we plot the average EDP on the rightmost chart of Figure 6. In this chart the lower values are the better. The *task-based* approach is the one that has the best performance-energy combination for the asymmetric configurations since it maintains the lowest EDP for all cases. *Static threading* manages to reduce the average EDP by 6% while *GTS* and *task based* approaches do so by 24% and 36% respectively.

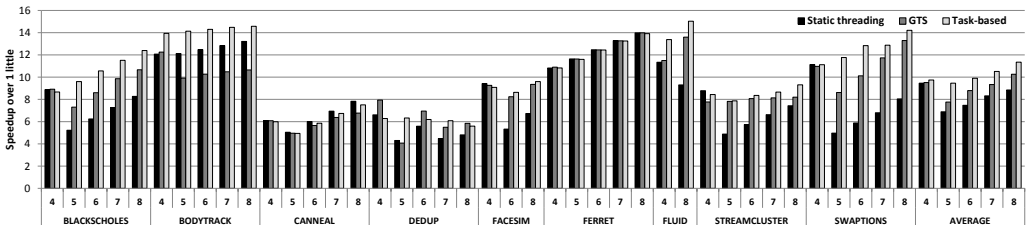

Figure 7: Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big

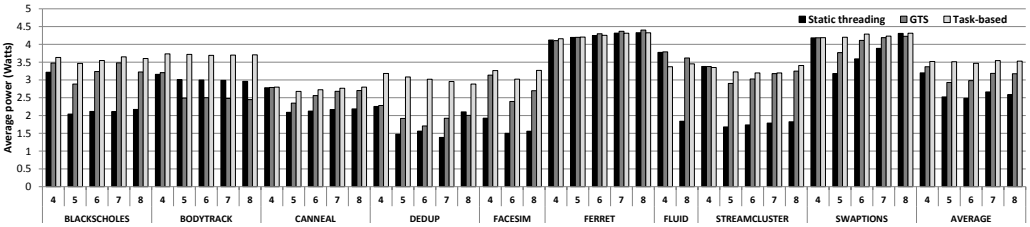

Figure 8: Average power when running on 4 to 8 cores and 4 of them are big

Figure 7 shows a more detailed exploration of the performance results. As Table 1 shows, the applications with barrier synchronization are blackscholes, facesim, fluidanimate, streamcluster and swaptions. For these applications the most efficient system configuration with the *Static threading* approach is the 4+0. Little cores increase execution time due to load imbalance effects. **R1Q10:** *GTS* and *task-based* approaches overcome these issues by scheduling the load to the appropriate resources. The differences in the improvement of the *task based* and GTS solutions for these applications relies on the nature of each application and its parallel implementation. For example, swaptions, benefits more from the *task based* and *GTS* approaches than streacmluster. This is because the task graph of streamcluster presents multiple small parallel regions that are spawned and synchronized. Due to the multiple synchronization points, *GTS* and *task-based* cannot increase performance of streamcluster as much. Contrarily, swaptions has less synchronization points, thing that allows GTS and task-based to exploit asymmetry during its longer parallel regions.

Applications with more advanced load balancing techniques like pipelined parallelism (bodytrack, dedup and ferret), benefit of the asymmetric hardware and balance the load among all the cores. As a result, the performance of *Static threading* approach does not degrade when adding little cores as

in the previous set of applications. In the case of bodytrack, *GTS* reduces performance by 15% when adding four little cores. We attribute this to the cost of the thread migration from one core to the other in contrast to the *Static threading* approach that does not add such overheads. *Task based* approach also avoids these overheads and improves the performance of bodytrack by efficiently scheduling the tasks among threads. In the case of dedup, results show more variability. This benchmark is very I/O intensive and, depending on the type of core that executes these I/O operations, performance drastically changes. In order to deal with this problem, a smarter dynamic scheduling mechanism would be required. Finally, canneal does not scale according to its ideal speedup reported on Figure 2 as it has a memory intensive pattern that limits performance.

Figure 8 shows the average power. The barrier-synchronized applications (blackscholes, facesim, fluidanimate, streamcluster and swaptions) reduce power because of their imbalance; since big cores have long idle times with the *Static threading* approach, they do not dissipate the same power as *GTS* and *Task-based*. For pipeline-parallel applications, both bodytrack and ferret maintain nearly the same power levels among the configurations for each scheduling approach. Dedup is an exception, as the results highly depend on the core that executes the aforementioned I/O operations. Yet, the effect of the lower power for *Static threading* is observed in all the benchmarks and is because the big cores are under-utilized.

Discussion

R1Q11: Sections 5.1 and 5.2 explored the potential of different scheduling approaches when used on various workloads on an AMC. It was proven that current applications are not ready to utilize an AMC and that adding little cores to an SMC with big cores presents significant challenges for the application, OS and runtime developers. Little cores increase load imbalance and can degrade performance as a result.

A dynamic OS scheduler such as *GTS* helps in mitigating load imbalance, providing an average performance increase of 10%. Barrier synchronized applications benefit more from the *GTS* approach as the applications with more sophisticated scheduling techniques can utilize the little cores more efficiently even with *Static threading*. *Task based* approach offers the optimal performance results for all types of workloads. It improves *Static threading*

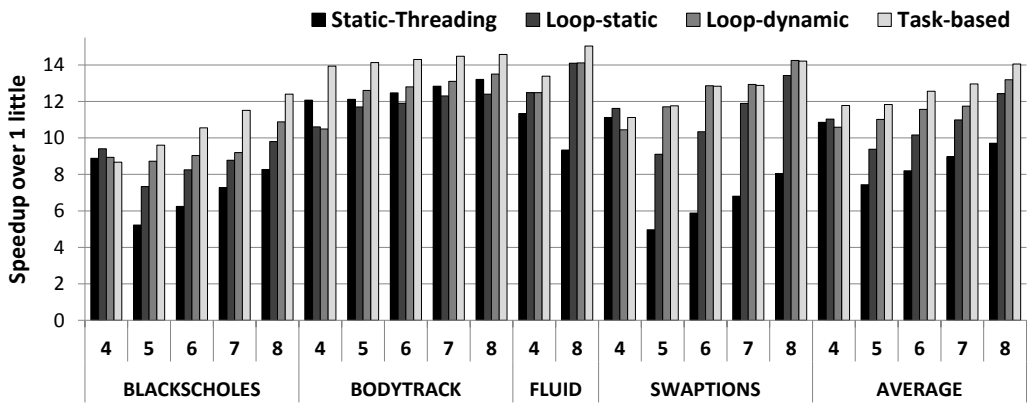

Figure 9: Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big. Four different programming models are considered: Static threading using `pthreads`, parallel loops with static scheduling (loop static), parallel loops with dynamic scheduling (loop dynamic), and a task-based solution with dynamic scheduling (task-based).

by 20% on average by effectively balancing the load among big and little cores.

In terms of power and energy, the AMC provides significant benefits, although the SMC with little cores remains the most energy-efficient configuration. **R1Q9:** This is attributed to the differences of the designs of the big and little cores; little cores have been optimized for power efficiency while the design of the big cores targets higher performance levels at the cost of higher energy consumption. The answer to the question of which system configuration provides the best power-performance balance, can be found on the average EDP chart of Figures 5 and 6, and is the use of the entire 8-core system with the *Task based* approach.

5.3. Programming Models for AMCs

As we saw in the previous section, current implementations of parallel applications are not ready to fully take advantage of an AMC system. Applications that are statically threaded using the low-level `pthreads` library usually suffer from load imbalance since their implementations assume that the work has to be equally distributed among the available cores. Implementing advanced load balancing schemes, such as work pools, in `pthreads` requires a significant development effort.

As an alternative, many parallel applications are implemented using loop-based scheduling with the OpenMP *parallel for* directives. In this case, the runtime library is in charge of scheduling work to the available threads in the system, either statically or dynamically, as described in Section 3.3.

We compare these solutions to the task-based approach evaluated in the previous sections. Figure 9 shows the results obtained from running blackscholes, bodytrack, fluidanimate and swaptions on all the scheduling models: static threading, static loop scheduling, dynamic loop scheduling and task-based scheduling. We chose these applications as they are the only ones implemented using the OpenMP loop directives.

Looking at the average results in Figure 9, we can observe that the task-based solution achieves the best results when the system is asymmetric. Task-based improves the static threading by up to 59% on 5 cores, while dynamic loop scheduling improves by up to 54%. The OpenMP version with static scheduling reaches an average 26% improvement over the static-threading approach with pthreads. **R1Q12:** The main reason for this improvement is that the OpenMP programming model by default allows the OS scheduler to migrate threads to cores. Thus, in this case, GTS is allowed to move threads from little to big cores or vice versa, which differs to the static threading that pins the threads to the cores. Similarly, loop-dynamic allows dynamic iteration scheduling as well as thread scheduling by GTS.

Taking a closer look to the results we observe that for bodytrack, an application with sophisticated parallelization techniques, static-threading achieves better results than loop-static. This is because the static-threading implementation contains specific parallelization techniques that cannot be completely expressed using the loop-static method. The loop-dynamic method improves performance for bodytrack by up to 4% due to the runtime decisions of the iteration execution, but the optimal solution is offered by the task-based approach that achieves up to 16% improvement over static-threading, due to the flexibility in expressing irregular parallelization strategies.

Blackscholes, fluidanimate and swaptions, consist of independent tasks and are a good fit for loop parallelism. The first observation is that all three applications benefit from the loop-static approach on an SMC with 4 big cores. Moreover, the task-based approach is still the optimal for blackscholes and fluidanimate, reaching up to 83% improvement over static threading for 5 cores, while for swaptions both task-based and loop-dynamic are efficient, improving the baseline by up to 2.3×. Finally, fluidanimate, that is also a fine-grained application that consists of 128 500 tasks, also benefits from the

task-based approach. For this benchmark, static and dynamic loop scheduling achieve similar performance; this is due to the limited parallelism per parallel region, as the loop-based implementation consists of multiple barriers between small parallel regions, fact that diminishes the effect of dynamic vs static scheduling.

6. Related Work

There has been a lot of studies on AMC systems. Some works focus on the system design, while other works explore the challenges that appear in efficiently utilizing such a heterogeneous system. Kumar et al [38] present the idea of an AMC system and proposed a feedback-based way to dynamically migrate processes among the different cores. To determine the core that most effectively executed a workload, Kumar et al [4] proposed the use of sampling. This method minimizes the execution time of each single thread and increases performance. Other studies focused on the pipeline design of such AMCs and the area that should be devoted to each component in the system [5, 39]. Other works on AMCs focus on hardware support for critical section detection [8] or bottleneck detection [11, 12]. These approaches are orthogonal to the ones evaluated in this paper and could benefit from them to further improve the final performance of the system.

Process scheduling on AMCs is one of the most challenging topics in this area of study. Bias scheduling [40] is an OS scheduler that characterizes the running threads according to their memory or execution intensity. It then schedules the computation intensive threads to the big cores of the system while the memory intensive threads to the little cores of the system. The experimental evaluation is done on Intel Xeon processors and the heterogeneous system is emulated by changing the configuration of three out of the four cores of the processor. Cong et al propose the Energy-Efficient [41] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [42] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et al. [43] propose the fairness-aware OS scheduler that focuses on AMC architectures. The performance impact estimation (PIE) scheduler [44] is based on the impact of MLP and ILP on the overall CPI and focuses on improving performance. The scheduler predicts the impact of each different core-type of the system on the MLP, ILP and it assumes hardware support for CPI. Rodrigues et al [45] propose a thread scheduling technique that estimates power and performance when deciding to assign a

thread to a specific core of the heterogeneous system. Finally, Energy-Aware Scheduling (EAS) is an on-going effort in the Linux community to introduce the energy factor in the OS scheduler [46, 47]. It is based on performance and power profiling to set performance and power capacities and let the Linux completely fair scheduler assign slots to processes considering the different core capacities. EAS is not yet part of the Linux kernel and, therefore, GTS is the most sophisticated state of the art scheduling method in production on current big.LITTLE processors.

Similar to OS scheduling approaches there have been many task scheduling approaches that are directed for utilizing AMCs. The Levelized Min Time [48] heuristic first clusters the tasks that can execute in parallel (*levels*) and then it assigns priorities to them, according to their execution time. The Dynamic Level Scheduling algorithm [49] assigns the tasks to the processors according to their *dynamic level* (DL). Heterogeneous Economical Duplication (HED) [50] duplicates the tasks in order to be executed on more than one cores but it then removes the redundant duplicates if they do not affect the makespan. CATS scheduler [51] is designed for AMCs like big.LITTLE and dynamically schedules the *critical* tasks to the big cores of the system to increase performance. Topcuoglu et al proposed the Heterogeneous Earliest Finish Time (HEFT) scheduler that statically assigns each task to the processor that will finish it at the earliest possible time. To do so, it keeps records with the task costs for each processor type. They also proposed the Critical Path on a Processor (CPOP) algorithm [52] that maintains a list of tasks and statically identifies and schedules the tasks belonging to the critical path to the processor that minimizes the sum of their execution times. The Longest Dynamic Critical Path (LDCP) algorithm [53] identifies the tasks that belong to the critical path and schedules them with higher priority.

All these works reflect the remarkable research that is taking place on AMCs. However we consider that their experimental evaluation is limited for three main reasons: i) The evaluation is done through a simulator or emulation of an AMC [38, 39, 5, 40, 43, 44, 45, 49, 48, 50, 8, 11, 12]; ii) The evaluated applications are either random task dependency graph generators or scientific kernels and micro-benchmarks [49, 52, 53]. iii) Their evaluation does not focus on power and energy consumption [4, 43, 44, 48, 51].

This paper includes a unique evaluation of performance, power and energy on a real AMC of real parallel applications. This paper also reflects the impact of using different big and little core counts which is not present in previous works [41].

7. Conclusions

R3Q1, R3Q3: This paper presented the first to tour knowledge extensive evaluation of highly parallel applications on an ARM big.LITTLE AMC system. The goal of this study was to identify whether such applications are ready to efficiently utilize an AMC system as well as finding the most appropriate software level for performing scheduling in order to maintain the load balance of the system. Quantifying such results requires a big effort in characterizing all parts of the evaluation including the applications as well as the scheduling approaches. The main findings of our work are the following:

- **Current implementations of parallel applications using pthreads are not ready to to fully utilize an AMC.** Our analysis covered a broad set of applications with different characteristics. From these, applications with highly sophisticated parallelization strategies such as parallel pipelines were able to exploit AMCs at the application level. However, this requires a significant programming effort and is not applicable to all workloads. The rest of the workloads are data-parallel applications that when moved to an AMC introduce load imbalance that limits their performance.

- **A highly sophisticated asymmetry-aware OS scheduler is not the ideal solution to schedule parallel applications on AMCs.** Our results demonstrated that GTS can only slightly improve performance of data-parallel applications. For applications with sophisticated parallelization strategies GTS fails to increase performance of the application as it introduces high overheads due to thread migration.

- **Even if it is asymmetry-unaware, the task-based solution is the most appropriate as it allows dynamic load balancing and eliminates the thread migration costs.** We saw that with a dynamic scheduling approach on the runtime system we have multiple benefits. First, it improves performance for all types of applications. In addition, there are cases where due to the increased programming flexibility in expressing parallelism, performance is improved (body-track). We further compared task-based solution against loop scheduling approaches on the runtime system and highlighted the benefits of the task-based solution.

R3Q6: As future work we aim to explore how the performance ratio between the types of cores affects the performance of the evaluated scheduling approaches. It is expected that increasing the performance ratio, the task-based approach is going to achieve even better results, as the load would be balanced again by dynamically assigning more work to the big cores. In this study, the performance-energy trade-off is something that needs to be taken into account as well.

Acknowledgements

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and by the European Union's Horizon 2020 research and innovation programme under grant agreement No 671697 and No. 779877. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, Others, Exascale Computing Study: Technology Challenges in Achieving Exascale Systems, Tech. rep., University of Notre Dame, CSE Dept. (2008).

[2] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, S. Mahlke, Composite cores: Pushing heterogeneity into a core, in: MICRO, 2012, pp. 317–328.

[3] Y. Wu, C. Gillan, U. Minhas, S. Barbhuiya, A. Novakovic, K. Tovletoglou, G. Tzenakis, H. Vandierendonck, G. Karakonstantis, D. Nikolopoulos, Heterogeneous servers based on programmable cores and dataflow engines, in: EnESCE, 2017.

[4] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, K. I. Farkas, Single-isa heterogeneous multi-core architectures for multithreaded workload performance, in: ISCA, 2004, pp. 64–75.

[5] S. Balakrishnan, R. Rajwar, M. Upton, K. K. Lai, The impact of performance asymmetry in emerging multicore architectures, in: ISCA, 2005, pp. 506–517.

[6] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, H. Wang, Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor, in: PACT, 2008.

[7] M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. Ward, R. Campbell, L. Carrington, Characterization and bottleneck analysis of a 64-bit armv8 platform, in: ISPASS'16, 2016, pp. 36–45.

[8] M. A. Suleman, O. Mutlu, M. K. Qureshi, Y. N. Patt, Accelerating critical section execution with asymmetric multi-core architectures, in: ASPLOS, 2009, pp. 253–264.

[9] A. Fedorova, J. C. Saez, D. Shelepov, M. Prieto, Maximizing Power Efficiency with Asymmetric Multicore Systems, Communications of the ACM 52 (12).

[10] P. Greenhalgh, big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7, ARM White Paper.

[11] J. A. Joao, M. A. Suleman, O. Mutlu, Y. N. Patt, Bottleneck identification and scheduling in multithreaded applications, in: ASPLOS, 2012.

[12] J. A. Joao, M. A. Suleman, O. Mutlu, Y. N. Patt, Utility-based acceleration of multithreaded applications on asymmetric CMPs, in: ISCA, 2013, pp. 154–165.

[13] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, M. Valero, Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?, in: SC, 2013.

[14] X. Zhan, Y. Bao, C. Bienia, K. Li, Parsec3.0: A multicore benchmark suite with network stacks and splash-2x, SIGARCH Comput. Archit. News 44 (5) (2017) 1–16.

[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, The design of OpenMP tasks, IEEE TPDS 20 (3) (2009) 404–418.

[16] OpenMP architecture review board: Application program interface (2013).

[17] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures., Parallel Processing Letters 21.

[18] B. Ren, S. Krishnamoorthy, K. Agrawal, M. Kulkarni, Exploiting vector and multicore parallelism for recursive, data- and task-parallel programs, in: PPoPP '17, New York, NY, USA, 2017, pp. 117–130.

[19] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: SC, 2012.

[20] K. Dichev, H. Jordan, K. Tovletoglou, T. Heller, D. Nikolopoulos, G. Karakonstantis, C. Gillan, Dependency-aware rollback and checkpoint-restart for distributed task-based runtimes, 2017.

[21] H. Vandierendonck, G. Tzenakis, D. S. Nikolopoulos, A unified scheduler for recursive and task dataflow parallelism, in: PACT, 2011.

[22] H. Vandierendonck, K. Chronaki, D. S. Nikolopoulos, Deterministic scale-free pipeline parallelism with hyperqueues, in: SC, 2013.

[23] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, C. R. Das, Controlled kernel launch for dynamic parallelism in gpus, in: HPCA'17, 2017, pp. 649–660.

[24] H. Chung, M. Kang, H.-D. Cho, Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology, Tech. rep., Samsung Electronics Co., Ltd. (2013).

[25] L. Gwennap, Cortex-A75 Has DynamIQ Debut, Microprocessor Report (2017).

[26] ARM, Juno ARM Development Platform.

[27] K. Krewell, Cortex-A53 is ARM's next little thing (2012).
URL `www.mpronline.com`

[28] J. Bolaria, Cortex-A57 extends ARM's reach (2012).
URL `www.mpronline.com`

[29] ARM, Cortex-A7 MPCore, revision: r0p3 (2011).
URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464d/DDI0464D_cortex_a7_mpcore_r0p3_trm.pdf`

[30] J. Turley, Cortex-A15 eagle flies the coop (2011).
URL `www.mpronline.com`

[31] ARM, Cortex-A15 technical reference manual, revision: r2p0 (2011).
URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438c/DDI0438C_cortex_a15_r2p0_trm.pdf`

[32] T. Cao, W. Huang, Y. He, M. Kondo, Cooling-aware job scheduling and node allocation for overprovisioned hpc systems, in: IPDPS'17, 2017, pp. 728–737.

[33] Mathieu Poirier, In Kernel Switcher: A solution to support ARM's new big.LITTLE technology, Embedded Linux Conference 2013 (2013).

[34] S. Zuckerman, J. Suetterlein, R. Knauerhase, G. R. Gao, Using a "Codelet" program execution model for exascale machines: Position paper, in: EXADAPT, 2011.

[35] C. Bienia, Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (2011).

[36] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, M. Valero, PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite, TACO.

[37] B. Jeff, big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling, ARM White Paper.

[38] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, D. M. Tullsen, Single-isa heterogeneous multi-core architectures: The potential for processor power reduction, in: MICRO, 2003, pp. 81–92.

[39] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, E. Ayguade, Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors, IEEE Comput. Archit. Lett. 5 (1) (2006) 4–17.

[40] D. Koufaty, D. Reddy, S. Hahn, Bias scheduling in heterogeneous multi-core architectures, in: EuroSys, 2010, pp. 125–138.

[41] J. Cong, B. Yuan, Energy-efficient scheduling on heterogeneous multi-core architectures, in: ISLPED, 2012, pp. 345–350.

[42] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, R. Iyer, Quickia: Exploring heterogeneous architectures on real prototypes, in: HPCA, 2012, pp. 1–8.

[43] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, L. Eeckhout, Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores, in: PACT, 2013.

[44] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, J. Emer, Scheduling heterogeneous multi-cores through performance impact estimation (pie), in: ISCA, 2012, pp. 213–224.

[45] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, Scalable thread scheduling in asymmetric multicores for power efficiency, in: SBAC-PAD, 2012, pp. 59–66.

[46] M. Anderson, Scheduler Options in big.LITTLE Android Platforms.

[47] Ian Rickards and Amit Kucheria, Energy Aware Scheduling (EAS) progress update, `https://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-progress-update` (2015).

[48] M. A. Iverson, F. Özgüner, G. J. Follen, Parallelizing Existing Applications in a Distributed Heterogeneous Environment, in: HCW, 1995.

[49] G. Sih, E. Lee, A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures, IEEE TPDS 4 (2).

[50] A. Agarwal, P. Kumar, Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems, in: IACC, 2009.

[51] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, M. Valero, Criticality-aware dynamic task scheduling for heterogeneous architectures, in: ICS, 2015, pp. 329–338.

[52] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing, IEEE TPDS 13 (3).

[53] M. Daoud, N. Kharma, Efficient Compile-Time Task Scheduling for Heterogeneous Distributed Computing Systems, in: ICPADS, 2006.

***Author Biography & Photograph**

Kallia Chronaki received the B.Sc in computer science and M.Sc. in computer architecture from the Computer Science Department (CSD) of the University of Crete, Greece.
She is currently a Ph.D. candidate at the department of Computer Architecture of the Polytechnic University of Catalonia (UPC), Spain and a research engineer at Barcelona Supercomputing Center (BSC).
Her research interests include high performance computing architectures, runtime systems and heterogeneous computing.

Miquel Moreto is a senior researcher at the Barcelona Supercomputing Center (BSC). Prior to joining BSC, he spent 15 months as a post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from UPC. His research interests include studying shared resources in multithreaded architectures and hardware-software co-design for future massively parallel systems.

Marc Casas is a senior researcher at the Barcelona Supercomputing Center.
Prior to this, he spent 3 years as a post-doctoral fellow at the Lawrence Livermore National Laboratory (LLNL).
He received his B.Sc. and M.Sc. degrees in mathematics in 2004 from the UPC and the PhD in Computer Science in 2010 from the Computer Architecture Department of UPC.
His research interests are high performance computing, runtime systems and parallel algorithms.

Alejandro Rico is a Staff Research Engineer at Arm working on processor architecture and microarchitecture for high-performance computing. Previously, he was a post-doctoral researcher at BSC working on multi-core simulation and microarchitecture. He received a Ph.D. from UPC and a M.Sc. and B.Sc. from Universitat Pompeu Fabra. His research interests are multi-core scalability, heterogeneous architectures and vector processors.

Rosa M. Badia holds a PhD on Computer Science (1994) from UPC. She is a Scientific Researcher from the Consejo Superior de Investigaciones Científicas (CSIC) and team leader of the Workflows and Distributed Computing research group at BSC.
Her research interests are programming models for complex platforms (from multicore, GPUs to Grid/Cloud). Dr Badia has published more than 150 papers in international conferences and journals in these topics. She has participated in a significant number of European funded projects and contracts with industry.

Eduard Ayguadè is full professor of the Computer Architecture Department at UPC. He is currently associate director of research in Computer Sciences at BSC. His research interests include multicore architectures, programming models and compilers for high-performance architectures. He published around 250 publications in these topics and participated in several research projects with other universities and industries, in framework of the European Union programmes or in direct collaboration with technology leading companies.

Mateo Valero is full professor at Computer Architecture Department, UPC and director at BSC.
He has published 700 papers and served in organization of 300 international conferences.
His main awards are: Seymour Cray, Eckert-Mauchly, Harry Goode, ACM Distinguished Service, "Hall of Fame" member IST European Program, King Jaime I in research, two Spanish National Awards on Informatics and Engineering. Honorary Doctorate: Universities of Chalmers, Belgrade, Las Palmas, Zaragoza, Complutense of Madrid, Granada and University of Veracruz. Professor Valero is a Fellow of IEEE, ACM, and Intel Distinguished Research Fellow. He is a member of Royal Spanish Academy of Engineering, Royal Academy of Science and Arts, correspondent academic of Royal Spanish Academy of Sciences, Academia Europaea and Mexican Academy of Science.

Formatted Manuscript

Click here to download Companion File (Non-Source File/Other Type): elsarticle-template-num_formatted.pdf