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

ABSTRACT

Asymmetric multi-cores are a successful architectural solution for the mobile market. They provide low power operation with a set of simple cores and have a set of high-performance cores for more demanding applications such as games. Both core types share the same instruction-set architecture so threads can migrate across core types without code transformations. This design enables long battery life through energy efficiency and high performance when needed. Asymmetric multi-cores have also been successful in supercomputing where they provide specialization for higher performance under the facility power budget. However, little work has been done on evaluating how to program and exploit asymmetric multi-cores in other domains.

In this paper, we evaluate several execution models on an ARM big.LITTLE asymmetric multi-core platform using the PARSEC benchmark suite that includes parallel applications from several areas such as finance, computer vision, physics, image processing and video encoding. We consider schedulers at the user, OS and runtime levels, both static and dynamic options and multiple configurations. We assess the impact of these scheduling options on the well-known problem of balancing the load across asymmetric multi-cores and conclude that the runtime system is the best entity in the software stack to make scheduling decisions on such environment. In our experiments on an asymmetric multi-core, using these multi-threaded applications *as-is* on all cores degrades performance compared to just using the big cores in the system. Contrarily, the heterogeneous-aware OS scheduler and dynamic runtime schedulers provide a 5% and 13% performance uplift despite the overheads incurred managing parallel work.

1 Introduction

The future of parallel computing is highly restricted by energy efficiency [27]. 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. Different types of processors can be specialized for different types of computation, such as the combination of general-purpose cores with accelerators such as Graphics Processing Units (GPUs). Another approach towards heterogeneity is the use of asymmetric multi-cores with different types of cores with the same instruction-set architecture. Different core types target different performance and power optimization points for energy efficiency [8, 31].

Asymmetric multi-cores have been successfully deployed in the mobile market, where low-power simple cores (*little*) are combined with high-performance out-of-order cores (*big*). Low demand applications run on little cores for low power operation and prolong battery life. Demanding applications, such as games, run on the big cores providing high performance when needed.

Supercomputing is another market where asymmetric multi-cores have been successful. The Sunway TaihuLight supercomputer topped the Top500 list of the fastest supercomputers in 2016 using asymmetric multi-cores. In this setup, big cores run the master tasks such as the OS and runtime system, as these tasks require support for speculation and Instruction-Level Parallelism (ILP) exploitation of codes with complex control flow. Little cores are equipped with wide Single Instruction Multiple Data (SIMD) units and lean pipeline structures for energy efficient execution of compute-intensive codes.

Previous experiences have shown that load balancing and scheduling are fundamental challenges that must be addressed to effectively exploit all the resources in these platforms [20, 21, 25, 26, 35, 38]. 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 asymmetric multi-core using the PARSEC benchmark suite. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We first quantify the performance loss of executing the applications *as-is* on all cores in the system. These applications were developed on homogeneous platforms and are bound to suffer from load imbalance on parallel regions that statically distribute the work evenly across cores without considering their performance disparity.

Then, we evaluate several solutions at the OS and runtime level that require different levels of user intervention to exploit asymmetric multi-cores effectively.

The first solution delegates scheduling to the OS. We evaluate the heterogeneity-aware OS scheduler used in existing mobile platforms that assigns threads to different core types based on CPU utilization. This requires no modification of the application, but has limited capability for high-utilization multithreaded applications.

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. The advantage is that the runtime system has knowledge of the application structure and parallel work boundaries so it can react with certain level of predictability. We evaluate dynamic scheduling on top of the existing work-sharing constructs in the applications with an OpenMP statically-scheduled implementation available. This requires code transformations that are straightforward in many cases.

Finally, we evaluate the impact of using an inherently load-balanced execution model such that of task-based programming models. Recent examples [7, 9, 19, 34, 43–45] include clauses to specify inter-task dependences and remove most barriers which are the major source of load imbalance on asymmetric multi-cores.

This paper quantifies the effectiveness of these solutions at different levels of the software stack with a comprehensive evaluation of representative parallel applications running on a real asymmetric multi-core platform: the Odroid-XU3 development board. This platform features an eight-core Samsung Exynos 5422 chip with ARM big.LITTLE architecture including four out-of-order Cortex-A15 and four in-order Cortex-A7 cores.

The rest of this document is organized as follows: Section 2 describes the evaluated asymmetric multi-core processor, while Section 3 offers information on dynamic schedulers at the OS and runtime system levels. Section 4 describes the experimental framework. Section 5 shows the performance and energy results and associated insights of our experiments. Finally, Section 6 discusses related work and Section 7 concludes this work.

2 The ARM big.LITTLE Architecture

The ARM big.LITTLE [15,21] is a state-of-the-art asymmetric multi-core architecture that has been successfully deployed in the mobile market. The observation that mobile devices typically combine phases with low and high computational demands motivated this original design. ARM big.LITTLE combines simple in-order cores with aggressive out-of-order cores in the same System-on-Chip (SoC) to provide high performance and low power dissipation. Both *big* and *little* cores support the same architecture so they can run the same binaries and therefore easily combined within the same system. Current cores implementing the ARMv7-A and ARMv8-A instruction set architectures support big.LITTLE configurations. Thus, the available cores to act as *little* cores are the ARM Cortex-A7, A35 and A53, while the available *big* cores are the ARM Cortex-A15, A17, A57, A72 and A73.

The little cores in a big.LITTLE system are designed targeting low power operation. Current implementations have relatively short pipelines with up to dual-issue in-order execution. L1 caches are split for instructions and data and can be dimensioned according to the target domain from 8 to 64 KB in size [29]. The big cores are designed for high performance. Current designs have deeper pipelines with up to three-issue out-of-order execution, increased number of functional units and improved floating-point capabilities. L1 data cache is 32 KB and L1 instruction cache is up to 48 KB [11,18]. Both little and big cores are grouped in clusters of up to 4 cores each. The cores within a cluster share a unified L2 cache. Multiple clusters can be integrated into an SoC through a cache coherent interconnect such as ARM CoreLink CCI or ARM CoreLink CCN [5].

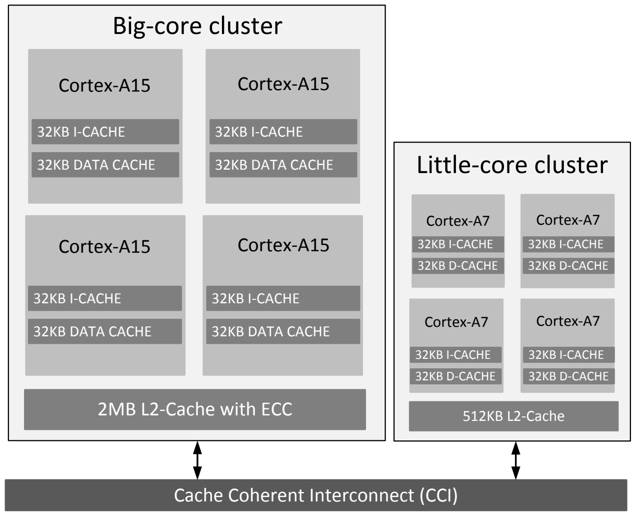

Figure 1: The Samsung Exynos 5422 processor implementing the ARM big.LITTLE architecture.

In this work, we make 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. 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 memory controller provides access to 2 GB of LPDDR3 RAM with dual 32-bit channels at 1866 MT/s. The reason we use this platform instead of the more up-to-date Juno platform [6] is that even if the latter features the more advanced Cortex A53 and Cortex A57 cores, it is limited to six cores instead of the 8 cores in the Samsung Exynos chip.

The ARM 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 core supports instruction prefetch by predicting the outcome of branches; the prefetch unit can fetch up to a maximum of four instructions per cycle. The L1 data cache is four-way set associative with physically-indexed and physically-tagged cache lines and uses a pseudo-random replacement policy [4]. Dynamic Voltage and Frequency Scaling (DVFS) techniques automatically 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 [40]. The L1 instruction and data caches of the Cortex-A15 are both 32 KB and 2-way set-associative with 64 byte cache lines. The processor supports speculative execution of instructions by maintaining a 2-level global history-based dynamic predictor with a branch target buffer [3]. The instruction decode unit performs register renaming to remove the Write-After-Write (WAW) and the Write-After-Read (WAR) hazards, and promote instruction reordering [3]. The instruction dispatch unit analyzes instruction dependences before issue them for execution. The integer execute unit includes 2 Arithmetic Logical Units (ALUs) with support for operand forwarding. DVFS techniques vary the frequency of the big cores from 200 MHz up to 2 GHz. For the remainder 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 asymmetric multicore system is more challenging than the traditional process scheduling on symmetric multi-cores. An efficient OS scheduler has to take into account the different characteristics of the core types of the system. 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 at the end of this section.

3.1 Cluster Switching and In-Kernel Switch

In the Cluster Switching (CS) approach [15], 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 this approach, all L2 cache contents are moved from one cluster to the other via the cache coherent interconnect. An enhanced version of the `cpufreq` driver, which is driven by CPU utilization, makes the decision to transition from one cluster to the other without involving changes at the OS kernel level.

In the In-Kernel Switch (IKS) approach [32], 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. As for CS, an enhanced `cpufreq` driver manages the switching within each core pair.

The advantage of IKS over CS is that different types of cores can be used at the same time to more efficiently execute a mix of high and low CPU utilization processes. The drawback is that, unlike in CS, the system must have the same number of little and big cores, as they must be paired one to one.

3.2 Global Task Scheduling

The Global Task Scheduling (GTS) [15] 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 schedules 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 [15]. 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[1], 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 available cores. Then, every core executes the assigned chunk, leading to a low-overhead scheduling that even the com-

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

piler can generate code for it without the need of runtime intervention. In addition, OpenMP supports dynamic loop scheduling. It generates more chunks than cores, typically with different sizes, and decides the assigns iteration chunks to cores as they become available. This is more appropriate for asymmetric multi-core systems where the cores are not equal and a static assignment of loop iterations would cause load imbalance.

Recent advances in programming models recover the use of task-based data-flow programming models to simplify parallel programming of multi-cores [9, 19, 34, 44, 45]. In these models the programmer splits the code in sequential pieces of work, called tasks, and specifies the data and control dependencies among them. With this information the runtime system schedules tasks and manages synchronization. These models not only ease programmability [9, 19, 34, 43–45], but also can increase performance by avoiding global synchronization points.

To evaluate this approach we make use of OpenMP tasking support [34]. 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.

4 Experimental Methodology

4.1 Metrics

All the experiments in this paper are performed on the Hardkernel Odroid XU3 described in Section 2. In our experiments, 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. 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[2]. 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 normalized 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 [10] 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 codes together with a task-based implementation of nine representative benchmarks of the suite [12].

Table 1 describes the benchmarks included in the study along with their respective inputs, parallelization strategy, lines of code (LOC) 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 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. As a result, both codes achieve the same performance on homogeneous processors with a reduced number of cores [12], while significantly reducing the total number of lines of code for benchmarks with application-specific load balancing mechanisms (bodytrack, dedup and ferret).

When running on the big.LITTLE processor, each benchmark exhibits different performance ratios between

[2]The parallel region duration is several orders of magnitude longer than the reading frequency of power sensors

Table 1: Description of the evaluated benchmarks from the PARSEC benchmark suite and the measured performance ratio between big and little cores per application.

Benchmark	Description	Input	Parallelization	LOC	Perf ratio
blackscholes	Calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation.	10,000,000 options	data-parallel	404	2.18
bodytrack	Computer vision application which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence.	4 cameras, 261 frames, 4,000 particles, 5 annealing layers	pipeline	6,968	4.16
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design.	2.5 million elements, 6,000 temperature steps	unstructured	3,040	1.73
dedup	Compresses a data stream with a combination of global compression and local compression in order to achieve high compression ratios.	351 MB data	pipeline	3,401	2.67
facesim	Takes a model of a human face and a time sequence of muscle activation and computes a visually realistic animation of the modeled face.	100 frames, 372,126 tetrahedra	data-parallel	34,134	3.40
ferret	Content-based similarity search of feature-rich data (audio, images, video, 3D shapes, etc.)	3,500 queries, 59,695 images database, find top 50 images	pipeline	10,552	3.59
fluidanimate	Extended Smoothed Particle Hydrodynamics method to simulate an incompressible fluid for interactive animations.	500 frames, 500,000 particles	data-parallel	2,348	2.64
streamcluster	Solves the online clustering problem.	200K points per block, 5 block	data-parallel	1,769	3.48
swaptions	Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions.	128 swaptions, 1 million simulations	data-parallel	1,225	2.78

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 2.96× on average.

Taking into account these performance ratio, we can estimate the ideal speedup of the platform for each workload assuming a perfect parallelization strategy. 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 synchronization points. Thus, these speedups are an upper bound of the achievable application performance. As expected, benchmarks with higher performance ratios exhibit potentially higher speedups on the asymmetric multi-core.

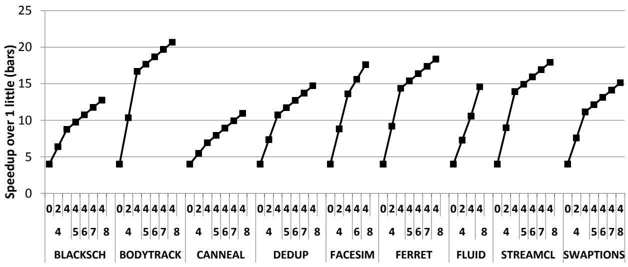

Figure 2: Ideal speedup over 1 little core according to Equation 4. Numbers at the bottom of x axis show the total number of cores, numbers above it show the number of big cores

5 Evaluation

We measure execution time, power, energy and EDP of nine applications from the PARSEC benchmark suite [10]. 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*[3]: dynamic coarse-grained OS scheduling using the GTS scheduler integrated in the Linux kernel [15,24] using the default PARSEC benchmarks.
- *Task-based*: dynamic fine-grained scheduling at the runtime level with the task-based implementations of the benchmarks provided in PARSECSs [12].

5.1 Exploiting Parallelism in AMCs

This section examines the opportunities and challenges that current asymmetric multi-cores (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 (denoted *Static threading*). We also evaluate

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

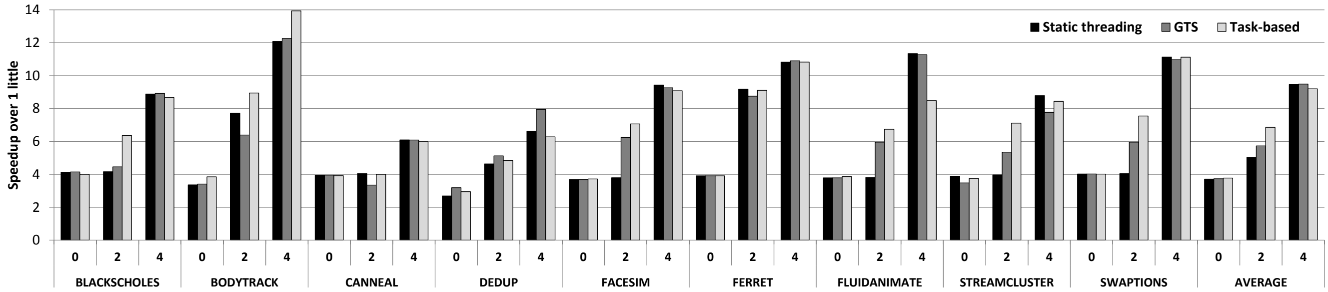

Figure 3: Execution time speedup over 1 little core for systems that consist of 4 cores in total with 0, 2 and 4 big cores. Different schedulers at the application (*static threading*), OS (*GTS*) and runtime (*task-based*) levels are considered.

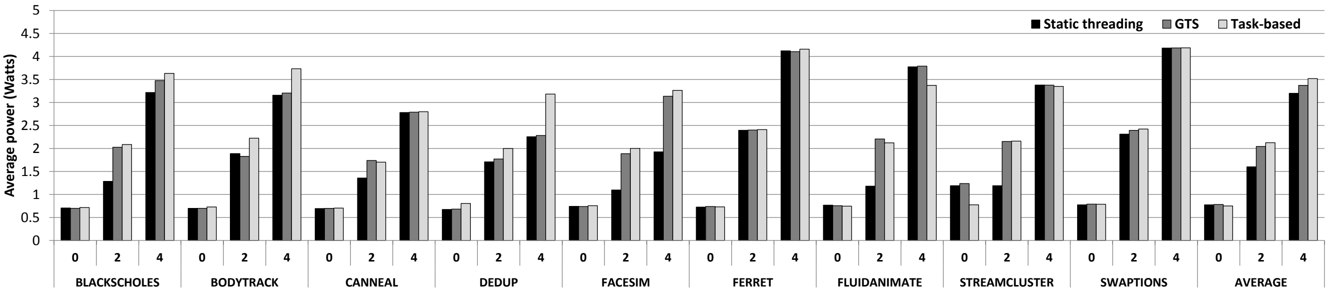

Figure 4: Average power measurements on a 4-core system with 0, 2, and 4 big cores.

the OS-based dynamic scheduling (denoted *GTS*) and the task-based runtime dynamic scheduling (denoted *Task-based*)s for the same applications. The system configurations evaluated in this section are the following: 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 first notice that all approaches perform similarly for the homogeneous configurations. Specifically, applications obtain the best performance on the configuration with four big cores, 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 parallelization strategies are effective for this core count. In the asymmetric configuration, *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 [21]. 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 the increase in power from 1.6W to 2.1W is compensated by a significant improvement in performance of 36%.

Finally, in terms of EDP the best configuration corresponds to using the four big cores, as the performance improvements compensate the increase in total energy. In configuration 2+2, *Task-based* achieves the same EDP results as in 0+4, but with 55% better performance. For the asymmetric configuration, the *Task-based* approach achieves the best combination of performance and energy 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 64% 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

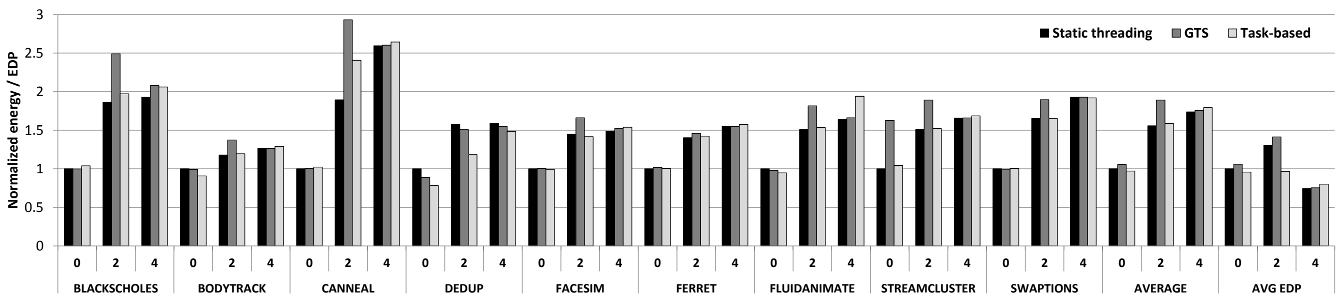

Figure 5: Normalized energy consumption and average EDP on a 4-core system with 0, 2, and 4 big cores. Static threading with 4 little cores is used as baseline in both cases.

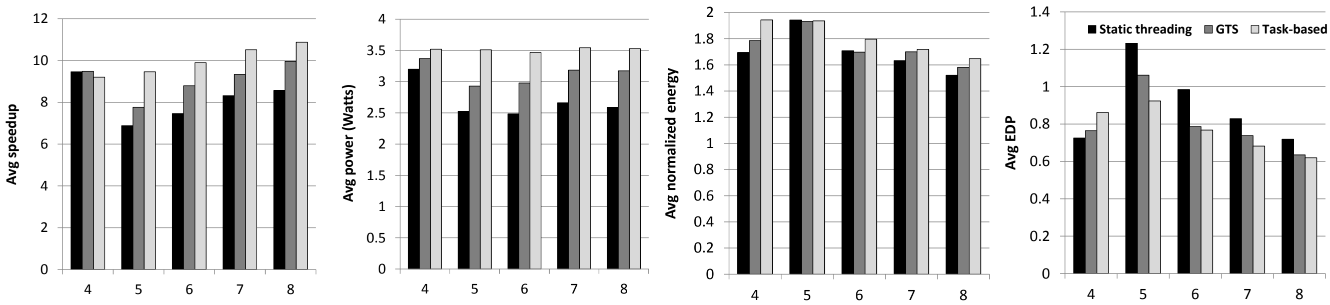

Figure 6: Average results when running on 4 to 8 cores with 4 of them big. Speedup is over 1 little core. Static threading on 4 little cores is the baseline of energy consumption and EDP

more advanced parallelization strategies that could benefit from asymmetric multi-cores, as performed in the remaining applications, or rely on a dynamic scheduling approach at OS or runtime levels.

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 load balancing mechanisms designed by the programmer. As a result, *Static scheduling* with these applications benefits from the extra computational power of the big cores in the configuration `2+2`. These mechanisms are not needed in the *Task-based* code; in this approach the code of the application is simplified and the runtime automatically allows the overlapping of the different pipeline stages. As a result, *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 benefit from asymmetry does not report good results, as it is very hard to predict the system's behaviour at application-level. Only when applications implement advanced features with user-level schedulers and load balancing techniques, they can benefit from asymmetry, of course 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 want to see how many small cores have to be added to the system to compensate the cons of having to deal with asymmetric multi-cores.

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 the *Static threading* approach does not benefit from adding little cores to the system. In fact, this approach brings an average 15% slowdown when adding four little cores for execution (configuration `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 will become idle and under-utilized. GTS achieves a limited speedup of 5% 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 match the performance of the symmetric configuration (configuration `4+3`). Finally, the *Task-based* approach always benefits from the extra computational power as the runtime automatically

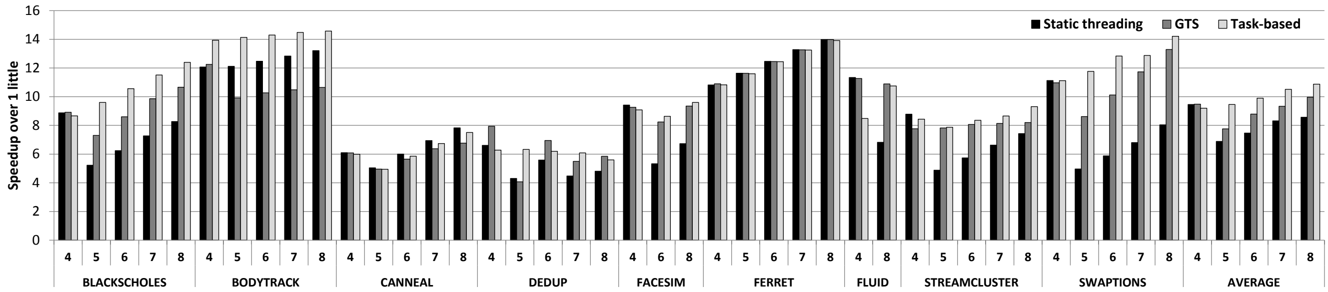

Figure 7: Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big

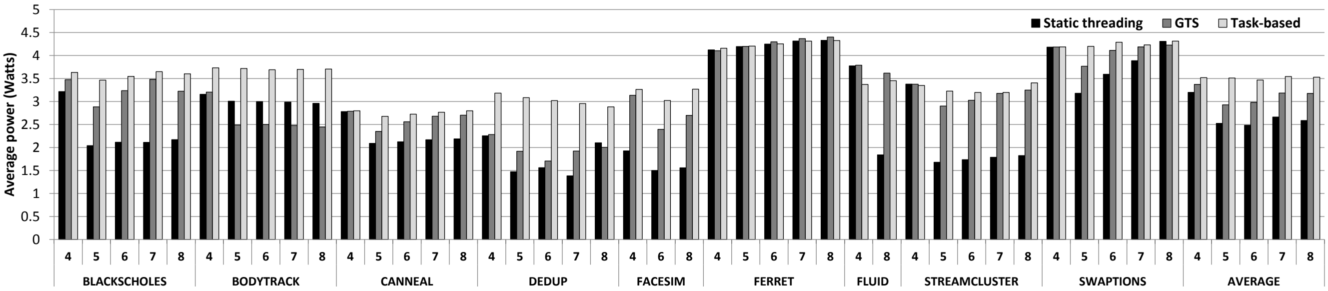

Figure 8: Average power when running on 4 to 8 cores and 4 of them are big

deals with load imbalance. Performance improvements keep growing with the additional little cores, reaching an average improvement of 16% over the symmetric configuration when 4 extra cores are added. An interesting observation is that according to the ideal speedup (Equation 4 and Figure 2), the average ideal performance increase when moving from the `4+0` configuration to the `4+4` is 33%, considering that the average performance ratio is 2.96× as shown in Table 1. Thus, the task-based approach achieves almost half of this theoretical ideal performance.

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.

As a result of the reduction in power, average energy results are reduced in the case of *Static threading* in configuration `4+4`, as shown on the energy chart of Figure 6. As discussed in the previous section, 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 11% for *Static threading*, 13% for *GTS*, and 18% for *Task-based*. Consequently, we can state that asymmetry reduces overall energy consumption, although the best configuration in terms of energy consists in using only four little cores (as average normalized energy is always above 1).

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. We observe that 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 12% while *GTS* and *task based* approaches do so by 11% and 18% respectively.

Figure 7 shows a more detailed exploration of the performance results. According to Table 1 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. Since the big cores reach barriers earlier, power is reduced for these applications, as shown in Figure 8. Energy reduction is less significant with a few extra little cores as the performance degradation is higher, but as the number of little cores increases, energy is reduced.

In the case of applications with more advanced load balancing techniques like pipelined parallelism (bodytrack, dedup and ferret), these applications benefit of the asymmetric hardware and balance the load in all the cores. As a result, performance improves while we increase the number of little cores. In the case of bodytrack, *GTS* reduces performance by 20% 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. 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 ap-

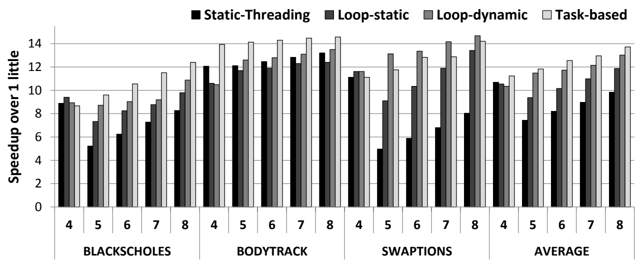

Figure 9: Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big. Four different programming models are considered: Static threading using `pthreads`, parallel loops with static scheduling (loop static), parallel loops with dynamic scheduling (loop dynamic), and a task-based solution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

plication 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 measured in each case. 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 spend 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 I/O operations mentioned above. However, the effect of the lower power for *Static threading* is observed in all the benchmarks and is because of the big cores' long idle times.

As we have seen in this section, adding little cores to a symmetric multi-core 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. Relying on the application developer to deal with this asymmetry is complex and many applications are not ready. A dynamic OS scheduler such as *GTS* can help in mitigating these problems, but the best results in terms of performance are obtained with the *Task-based* approach. In terms of power and energy, the asymmetric multi-core provides significant benefits, although the symmetric multi-core with little cores remains the most energy-efficient configuration. The answer to the question of what is the best system configuration for the highest performance the lowest energy consumption 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 asymmetric multi-core 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. With static loop scheduling the iterations of a loop are statically assigned to the available threads in the system, which minimizes the parallelization overhead. However, the static binding of iterations to resources in asymmetric multi-cores is likely to exhibit the same load-balancing problem seen for pthreads. To overcome this limitation, we can alternatively use dynamic loop scheduling. With this approach, the runtime library assigns chunks of iterations to threads dynamically as cores become available. This is more appropriate for asymmetric systems as it can significantly reduce load imbalance.

Next, we compare the above mentioned solutions to the task-based approach evaluated in the previous sections. Our goal is to evaluate the suitability of each programming model for asymmetric multi-cores. Figure 9 shows the results obtained from running the applications blackscholes, bodytrack and swaptions on all the aforementioned models: static threading, static loop scheduling, dynamic loop scheduling and task-based scheduling. We chose these applications because they are the only ones with implementations available in all the programming models under evaluation: `pthreads`, OpenMP loop directives and OpenMP tasks.

Looking at the 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% for 5-core configuration, 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.

Taking a closer look to the results we observe that for bodytrack, an application with sophisticated parallelization techniques, the static threading solution achieves better results than the loop-static. This is because the static-threading implementation contains specific techniques for the efficient parallelization 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 for bodytrack is offered by the task-based approach that achieves up to 16% improvement over static-threading, due to the flexibility for expressing the irregular parallelization strategies.

Blackscholes and swaptions, are highly parallel ap-

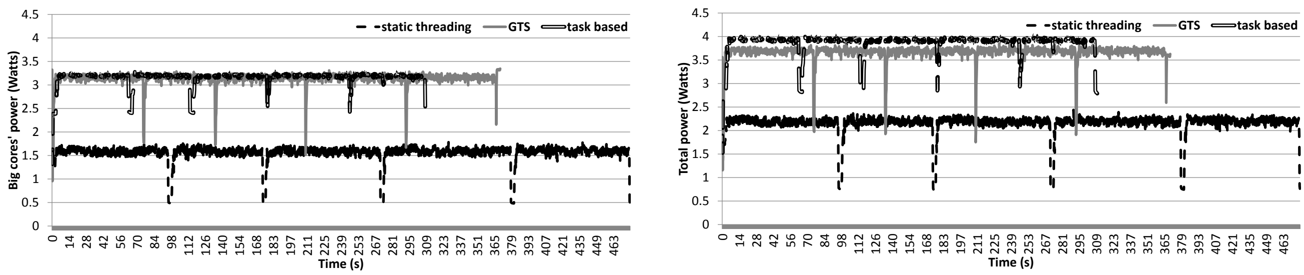

Figure 10: Power consumption of the Streamcluster benchmark on the 8-core Octodroid Platform. Left: Consumption of the 4 big cores. Right: Consumption of the whole chip.

plications that consist of independent tasks and are a good fit for loop parallelism. The first observation is that both applications benefit from the loop-static approach on a homogeneous multi-core with 4 big cores. Moreover, the task-based approach is still the optimal for blackscholes, reaching up to 83% improvement over static threading, while for swaptions loop-dynamic is the most appropriate, improving the baseline by up to 2.6×. The difference in the benefits of these two applications relies on the task granularity; blackscholes consists of 6400 tasks that are about a hundred times smaller that each one of the 128 tasks of swaptions. This shows us that loop-dynamic techniques are more efficient on coarse-grained applications, while the task-based approach effectively hides its runtime overheads when there is a high number of tasks in the application.

5.4 Power Consumption Analysis over Time

This section provides a detailed analysis of the power consumption over time for one of the evaluated applications, streamcluster. We choose this application as an illustrative example of the utilization of the processor resources (performance and power) with the different evaluated approaches. Figure 10 shows the power samples measured over the execution time of streamcluster when running on 8 cores (4+4 configuration). On the left part of the figure we plot the power samples of only the 4 big cores, while on the right part we plot the total power samples of all the big and the little cores. Both charts contain this information for the three scheduling approaches evaluated in this paper (*Static threading*, *GTS* and *Task-based*). As expected, all the approaches display the same five execution phases throughout the execution, as this benchmark is processing five large chunks of points. Seemingly the power samples of the big-core cluster slightly outreach 3W for the *GTS* and *Task-based* approaches, while for *Static threading* they remain close to 1.5W. As shown in Figure 4, steamcluster dissipates 1.2W when running on 4 little cores. Thus, this proves that the *GTS* and *Task-based* approaches better utilize the big cores and, in contrast, the big cores remain idle for a significant amount of time with the *Static threading* strategy.

On the right of Figure 10, where the power samples of the whole chip are plotted, the *Task-based* approach has power samples slightly higher than the *GTS* approach: with the *Task-based* approach, measured power is around 4.0W, while with *GTS* observed power is around 3.6W. The main difference between these measurements and the ones on the left chart of the same figure is the power consumption of the little cores. Thus we derive that both *GTS* and *Task-based* fully utilize the big cores in the system, but the *Task-based* approach utilizes the little cores better than *GTS*. In contrast, *Static threading* does not take advantage of the computational power of this asymmetric multi-core for the streamcluster benchmark.

Despite the fact that performance and power consumption of the little cores are much smaller than the ones of the big cores, the *Task-based* approach significantly reduces the execution time of streamcluster from 374 seconds to 312 seconds. A more effective usage of the little cores at the cost of slightly higher power consumption leads to these results. This clearly demonstrates the benefits of runtime system-based programming models against thread-based approaches in asymmetric multi-core systems.

6 Related Work

There has been a lot of studies on asymmetric multi-core systems. Some works focus on the design of the system, while other works explore the challenges that appear in efficiently utilizing such a heterogeneous system. Kumar et al [30] present the idea of an asymmetric multi-core 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 [31] proposed the use of sampling. The proposed method minimizes the execution time of each single thread and increases performance. Other studies focused on the pipeline design of such asymmetric systems and the area that should be devoted to each component in the system [8, 33]. Other works on asymmetric systems focus on hardware support for critical section detection [38] or bottleneck detection [25, 26]. These approaches are orthogonal to the approaches evaluated in this paper and could benefit from them to further improve the final performance of the system.

Process scheduling on asymmetric systems is one of the most challenging topics in this area of study. Bias scheduling [28] 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 [16] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [13] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et al. [41] propose the fairness-aware OS scheduler that focuses on asymmetric multi-core architectures. The performance impact estimation (PIE) scheduler [42] 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 [36] 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 [2,22]. 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 asymmetric systems. The Levelized Min Time [23] 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 [37] assigns the tasks to the processors according to their *dynamic level* (DL). Heterogeneous Economical Duplication (HED) [1] 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 [14] is designed for asymmetric systems 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 [39] 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 [17] 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 asymmetric systems. However we consider that their experimental evaluation is limited for three main reasons: i) Their experimental evaluation is done through a simulator or emulation of an asymmetric system [1,8,23,25,26,28,30,31,33,36–38,41,42]; ii) The evaluated applications are either random task dependency graph generators or scientific kernels and micro-benchmarks [14,17,37,39]. iii) Their evaluation does not focus on power and energy consumption [14, 23,31,37,41,42].

This paper presents a comprehensive evaluation of performance, power and energy on a real asymmetric system of parallel desktop applications. Another important point that this paper makes is the impact of using different big and little core counts which is not present in previous works [16].

7 Conclusions

Asymmetric multi-cores are a successful architectural solution for mobile and supercomputing systems. Our evaluation shows how they perform for other domains and how the several existing execution models for asymmetric multi-cores behave in terms of performance and power.

We compared the statically-threaded out-of-the-box implementation, the GTS OS scheduler, and dynamic scheduling at the runtime level, both for OpenMP loops and a task-based implementation. When using all the cores in the system, it stands out that the task-based better balances the load and avoids having big cores waiting for little cores to reach synchronization points.

We confirm that the out-of-the-box implementation, for most applications, does not effectively utilize the asymmetric system. This confirms the well-known problem of load imbalance when evenly distributing work among diverging core types. On average, static threading, when using eight cores, is 12% less efficient than when using just four big cores. An exception is the case of ferret, due to its pipelined parallelism that helps on the effective utilization of the little cores when they are added to the system.

The OS scheduler partially increases load balance. However, as it migrates threads based on CPU utilization, its behaviour is mostly reactive. It migrates threads when they become inactive and, at that point, the thread has already been spinning for some time. Using all cores in the system is 5% better than using big cores only.

Finally, using dynamic scheduling on OpenMP worksharing constructs reduces load imbalance and helps to better exploit all resources. Task-based parallelism further reduces imbalance achieving 13% performance uplift with all cores. The fundamental factors for this improvement are the removal of fork-join schemes and barriers thanks to inter-task dependencies.

These solutions provide different levels of application refactoring. Our performance and power discussion and quantification becomes a useful resource to select the right execution model for a given performance-effort point and satisfactorily exploit asymmetric multi-cores.

8 References

[1] A. Agarwal and P. Kumar. Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems. In *IACC*, 2009.

[2] M. Anderson. Scheduler Options in big.LITTLE Android Platforms. `http://events.linuxfoundation.org/sites/events/files/slides/GTS_Anderson.pdf`, 2015.

[3] ARM. Cortex-A15 technical reference manual, revision: r2p0, 2011.

[4] ARM. Cortex-A7 MPCore, revision: r0p3, 2011.

[5] ARM Ltd. CoreLink Interconnect. `http://www.arm.com/products/system-ip/interconnect`, Last accessed Nov 22, 2015.

[6] ARM Ltd. Juno arm development platform. `https://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php`, Last accessed Nov 23, 2015.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[8] S. Balakrishnan, R. Rajwar, M. Upton, and K. K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA*, pages 506–517, 2005.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC*, pages 66:1–66:11, 2012.

[10] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[11] J. Bolaria. Cortex-A57 extends ARM's reach, 2012.

[12] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, and M. Valero. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *Trans. Archit. Code Optim.*, 2015.

[13] 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, and R. Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *HPCA*, pages 1–8, 2012.

[14] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *ICS*, pages 329–338, 2015.

[15] H. Chung, M. Kang, and H.-D. Cho. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology. Technical report, Samsung Electronics Co., Ltd., 2013.

[16] J. Cong and B. Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *ISLPED*, pages 345–350, 2012.

[17] M. Daoud and N. Kharma. Efficient Compile-Time Task Scheduling for Heterogeneous Distributed Computing Systems. In *ICPADS*, 2006.

[18] M. Demler. Cortex-A72 takes big step forward, 2015.

[19] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21, 2011.

[20] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Communications of the ACM*, 52(12), 2009.

[21] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, 2011.

[22] 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.

[23] M. A. Iverson, F. Özgüner, and G. J. Follen. Parallelizing Existing Applications in a Distributed Heterogeneous Environment. In *HCW*, 1995.

[24] B. Jeff. big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling. *ARM White Paper*, 2013.

[25] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234, 2012.

[26] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *ISCA*, pages 154–165, 2013.

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

[28] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, pages 125–138, 2010.

[29] K. Krewell. Cortex-A53 is ARM's next little thing, 2012.

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

[31] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, pages 64–75, 2004.

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

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

[34] OpenMP architecture review board: Application program interface, 2013.

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

[36] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu. Scalable thread scheduling in asymmetric multicores for power efficiency. In *SBAC-PAD*, pages 59–66, 2012.

[37] G. Sih and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2), 1993.

[38] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[39] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3), 2002.

[40] J. Turley. Cortex-A15 eagle flies the coop, 2011.

[41] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores. In *PACT*, 2013.

[42] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, pages 213–224, 2012.

[43] H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos. Deterministic scale-free pipeline parallelism with hyperqueues. In *SC*, pages 32:1–32:12, 2013.

[44] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow

parallelism. In *PACT*, pages 1–11, 2011.

[45] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a “Codelet” program execution model for exascale machines: Position paper. In *EXADAPT*, pages 64–69, 2011.