Exploiting Asymmetric Multi-Core Processors with Flexible System Sofware

***Abstract*—The power consumption levels of current multicore architectures have boosted the interest towards heterogenous designs, i. e., towards combining different kinds of cores and adding on-chip accelerators. The improvements that mobile processor technology is experimenting, which are driven by the enormous market they target, also push towards considering asymmetric designs that combine fast out-of-order cores with simple in-order low-power cores for general purpose computing.**

This paper presents a comparison between three parallel paradigms aimed at fully using mobile heterogenous architectures: One is based in a static thread allocation specified by the programmer, another is a thread allocation policy driven by the operating system that moves parallel threads across the heterogeneous platform and the third is a runtime system software specifically designed to optimally schedule tasks around multi-core designs. We use nine parallel desktop applications from the PARSEC benchmark suite on an ARM big.LITTLE processor architecture to carry out our experimental campaigns. Our evaluation shows that even though the runtime system paradigm spends more power, the increased performance that it offers is enough to provide improvements of XXXXX in terms of Energy Delay Product (EDP). Furthermore, we explore the potential of the little cores and the circumstances that the addition of such cores to the systems helps on improving performance and energy efficiency, showing that only a runtime system based approach is flexible enough to fully use heterogeneous architectures.

I. Introduction

The future of High-Performance Computing (HPC) is highly restricted by energy efficiency [1]. Energy efficiency has become the main challenge for HPC 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 multi-cores, Graphics Processing Units (GPUs), and different accelerators. Another approach towards heterogeneity is the use of asymmetric multi-cores with different types of cores targeting different performance and power optimization points.

The use of asymmetric multi-core architectures is an approach towards increasing energy efficiency [2], [3]. Asymmetric multi-cores have been successfully deployed in the mobile arena, where simple in-order cores (slow) have been combined with aggressive out-of-order cores (fast) to build these systems. In a first generation of asymmetric multi-cores, the system could switch from low power to high responding operation modes, activating or de-activating the cluster of fast or slow cores accordingly. In a second generation of asymmetric multi-core processors, all the cores can run simultaneously to further improve the peak performance of these systems.

Many researchers are pushing towards building future parallel and HPC systems with asymmetric multi-cores [4]–[8] and even mobile chips [9]. However, it is unclear if current parallel applications will benefit from these asymmetric platforms. Load balancing and scheduling are two of the main challenges in utilizing such heterogeneous platforms, as the programmer has to consider them from the very beginning to obtain an efficient parallelization.

In this paper, we evaluate the maturity of currently available asymmetric multi-core platforms for emerging parallel applications. More specifically, we try to answer if out-of-the-box parallel applications can run efficiently on asymmetric multi-cores. Fully exploiting the computational power of these processors is challenging as the asymmetry in the system can lead to load imbalance, undermining the scalability of the parallel application. Consequently, only applications that incorporate user-defined load balancing mechanisms can benefit immediately from asymmetric multi-cores.

For the cases where the programmer did not implement advanced load balancing techniques, we evaluate alternative solutions that, without relying on the programmer, can leverage the opportunities that asymmetric systems offer. In particular, we evaluate a dynamic scheduler at the Operating System (OS) level that balances the load between the cores of the asymmetric system.

An alternative to tackle the challenges of asymmetric multi-cores consists in using advanced programming models with dynamic scheduling techniques. Recent task-based programming models arise as a suitable solution for these systems [10]–[16]. These programming models allow the specification of inter-task dependences and rely on an advanced runtime system to track dependences between tasks. When these dependences are satisfied, tasks are automatically scheduled, effectively balancing the load in the system.

In order to determine the usefulness of these solutions at different levels of the software stack, we perform a comprehensive evaluation of emerging parallel applications on a real asymmetric multi-core, an Odroid-XU3 development board featuring 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 main findings of this evaluation are the following:

- On this system, the best configuration in terms of energy and power is using only the in-order cores. Out-of-the-box applications obtain the best average performance when running only on the aggressive out-of-order cores. Many of these applications are not ready to fully exploit asymmetric multi-cores as they suffer from load imbalance. As a result, an average 12% performance degradation is obtained when using all the cores in the system instead of the four out-of-order cores.

- Dynamic scheduling techniques at OS or runtime level reduce the total execution time by 5.3% and 13% respectively when adding four slow cores to a system with four fast cores. A runtime scheduler fully utilizes the resources at the cost of power increase. However, the 13% performance increase is enough to compensate, as the Energy-Delay Product (EDP) is reduced by 40%.
- Finally, we evaluate the usefulness of slow cores to offload runtime activities. Similarly to the assistant core in the IBM Blue Gene Q and the Fujitsu SPARC64 XIfx processors [17], [18], we explore the possibilities of devoting a slow in-order or a fast out-of-order core to the runtime activity. However, the noise introduced by the runtime activity is less significant than in MPI, which allows to use this assistant core to run user tasks, increasing the final performance of the parallel application.

The rest of this document is organized as follows: Section II describes the evaluated asymmetric multi-core processor, while Section III offers information on dynamic schedulers at OS and runtime levels for these systems. Next, Section IV describes the experimental framework, while Section V evaluates the performance and energy of the selected applications on asymmetric systems. Finally, Section VI describes the related work and Section VII concludes this work.

II. The ARM big.LITTLE Architecture

The ARM big.LITTLE [19] 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 be mixed and matched 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 cores to act as *big* cores are the ARM Cortex-A15, A17, A57 and A72.

The little cores in a big.LITTLE system are designed targeting low power operation. They 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 [**?**], [20]. The big cores are designed for high performance. They 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 [21], [22]. 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 with *big* and *little* cores each can be integrated into an SoC through a cache coherent interconnect such as ARM CoreLink CCI or ARM CoreLink CCN [**?**], [**?**].

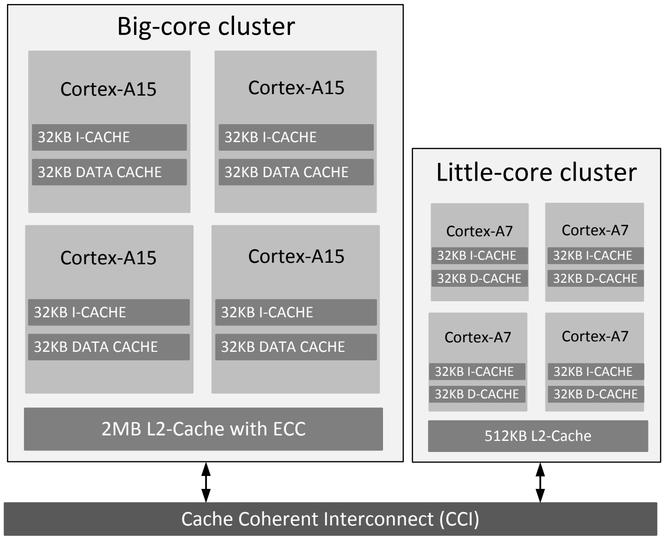

Fig. 1. The Samsung Exynos 5422 processor implementing the ARM big.LITTLE architecture.

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 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 [23]. The frequency of the little cores can be changed 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 [24]. 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 instruction prefetch by maintaining a 2-level dynamic predictor with branch target buffer, a static branch predictor and an indirect predictor [25]. 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 [25]. In the instruction dispatch unit the instruction is being analysed and the integer execute unit includes 2 Arithmetic Logical Units (ALUs) with forwarding for faster execution. The big cores frequency can be varied 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*.

III. Scheduling in Asymmetric Multi-Cores

Scheduling a set of processes on an asymmetric multi-core 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. In the ARM big.LITTLE processor two different scheduling approaches are supported: the *cluster switching*, and the *global task scheduling*, described in the next two sections. In the case of task-based parallel applications, *dynamic scheduling at the runtime level* can be exploited to balance the workload among the different cores in the system and is described at the end of this section.

A. Cluster Switching

In the cluster switching (CS) approach, only one of the clusters is active at any given time: either the cluster with little cores or the cluster with big cores is allowed to execute tasks. Thus, the OS scheduler operates on a *de-facto* symmetric multi-core with only four cores, namely the cores of the current activated cluster. As a result, the entire workload of tasks is being managed as a unique entity and the OS keeps track of its operational intensity. If the workload reaches a pre-defined threshold, then the entire workload is moved to the other cluster, and the current cluster gets deactivated. In this approach, all L2 cache contents are moved from one cluster to the other via the cache coherent interconnect. In the cluster switching approach, an enhanced version of the `cpufreq` driver makes the decision to transition from one cluster to the other without involving changes at kernel level.

B. Global Task Scheduling

The second OS scheduling approach, the Global Task Scheduling (GTS), allows running applications on all the 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 and each type of core. Moreover, the workload is managed as a set of tasks characterized by their *intensity* **Miquel:** Definition of intensity?. Thus, GTS schedules the high intensity tasks to the big cores and the low intensity tasks to the little cores. As a result, cores are activated according to the characteristics of the tasks in the workload. GTS is described as the most sophisticated scheduling method in production on current big.LITTLE processors [26][1]. The key benefits of GTS over CS are:

- In GTS tasks are directly migrated to cores, while in CS the entire workload is migrated to clusters. This makes the scheduling more flexible and helps on the more effective utilization of the system according to the workload.
- GTS can support clusters with different number of cores (e.g. with 2 big cores and 4 little cores), while CS requires to have exactly the same number of cores per cluster.
- The ability to use all cores simultaneously provides improved peak performance of the system.

As a result of these characteristics, Samsung reported 20% improvement in performance over CS for mobile benchmarks [26].

C. Dynamic Scheduling at Runtime Level

The use of parallel programming models on such systems could increase performance and energy efficiency. Current programming models for shared memory systems such as OpenMP rely on a runtime system to manage the execution of the parallel application. Recent advances in programming models recover the use of task-based data-flow programming models to simplify parallel programming in future many-cores [11]–[15]. In these models the programmer splits the code in sequential pieces of work, called tasks, and specifies the data and control dependences between them. With this information the runtime system manages the parallel execution, schedules tasks and takes care of synchronization. These models not only ease programmability, but also can increase performance by avoiding global synchronization points. The increased task granularity that the runtime system can offer has a positive effect on their most effective execution with the minimum possible energy consumption.

To evaluate this approach we use OmpSs, a task-based programming model conceived as a forerunner of OpenMP [12]. Both programming models allow expressing tasks and data-dependences between them using equivalent code annotations. They conceive the parallel execution as a *task dependence graph* (TDG), where nodes are sequential pieces of code (the tasks) and the edges are control or data dependences between them. The runtime system builds this TDG during execution time and is in charge of dynamically scheduling tasks.

A task is *created* when a call to this task is discovered in the code. The runtime system adds it to the TDG using the specified input dependences. The task cannot be scheduled until all its inputs dependences are produced by other tasks previously created. When all dependences are satisfied, the task becomes *ready* and is inserted in a *ready queue*. At this point, the task can be executed according to the scheduling policy of the runtime. Ready queues can be thread-private or shared among multiple threads. When a thread becomes idle, the scheduling policy picks a task from a ready queue and it is executed in that particular thread.

The default scheduling policy is processing the tasks in a *first in, first out* manner (FIFO). This policy is implemented with a single ready FIFO queue shared among all threads for keeping the ready tasks. Whenever a task becomes ready it is pushed to the tail of the ready queue; the first available

[1] Different implementations of GTS exist based on the same concepts. Samsung denotes its proprietary implementation *Heterogeneous Multi-Processing*.

processor, pops the ready task that resides at the head of the ready queue. Simultaneous pushes and pops are protected by locking mechanisms. Because the tasks are scheduled dynamically to the available cores, this scheduler automatically balances the load without the need of work stealing. Finally, the scheduler is not aware of the task intensity or the core type and its characteristics, but only relies on the availability of tasks and resources.

More advanced runtime features are also available, providing runtime support for NUMA-aware allocation, task priorities, device support for heterogeneity, different scheduling policies, throttling policies, dependence tracking mechanisms, work-sharing and work stealing. However, in this paper we make use of the simple FIFO scheduler and the vanilla features of the runtime system for a fair comparison with GTS.

IV. Experimental Methodology

A. Metrics

All the experiments in this paper are performed on the Hardkernel ODROID XU3 described in Section II. In this platform, there are four separated current sensors to measure in real time the power consumption of the cluster of A15 cores, the cluster of A7 cores, the GPU and DRAM.

To estimate the impact of the different kinds of cores, we evaluate seven configurations with different numbers of *little* (L) and *big* (B) cores, denoted `L+B`. In our experiments, big cores run at 1.6GHz, while little cores run at 800MHz. For each configuration and benchmark, we report the average performance of five executions taking into account only the parallel region of the application. Then, we report the speedup of the application over its execution on one little core with the task-based scheduling[2]. Equation 1 shows the formula to compute the speedup.

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

To gather the power and energy measurements, a background daemon is reading the machine's power sensors periodically during the application's execution with very low overhead. Sensors are read every 0.27 seconds, and their values are written in a file. With the help of timestamps, we correlate the power measurements with the parallel region of the application in a *post-mortem* process. The reported power consumption is the average power that the application was consuming during five executions of each set-up, considering only the parallel region of the application. We then report the average power in Watts during the execution.

Finally, in terms of energy and EDP, we report the total energy and EDP of the benchmark's region of interest normalized to the value that it consumes when running on four little cores with static threading. Equations 2 and 3 show the formulas for these calculations.

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

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

B. Applications: The PARSEC Benchmark Suite

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 [27] 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 [27] together with a taskified version of nine representative benchmarks of the suite [28]. Table I describes the benchmarks included in the study along with their respective inputs, parallelization strategy and the lines of code (LOC) of each 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 does not fit in the memory system of our platform. Instead, we reduce the size of the file to compress 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 [28], 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 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 I 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 has a positive effect on this application's performance. 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 the performance ratio we can then 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 computa-

[2] All approaches running on a single core report the same performance.

TABLE I
EVALUATED BENCHMARKS FROM THE PARSEC BENCHMARK SUITE

Benchmark	Description	Input	Parallelization	Synchronization	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	dataflow/barrier	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	dataflow	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	locks/atomics	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	dataflow/locks	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	dataflow/barrier	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	dataflow	10,552	3.59
fluidanimate	Uses an extension of the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid for interactive animation purposes.	500 frames, 500,000 particles	data-parallel	dataflow/barrier	2,348	2.64
streamcluster	Solves the online clustering problem.	200,000 points per block, 5 block	data-parallel	dataflow/barrier/atomics	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	dataflow/barrier	1,225	2.78

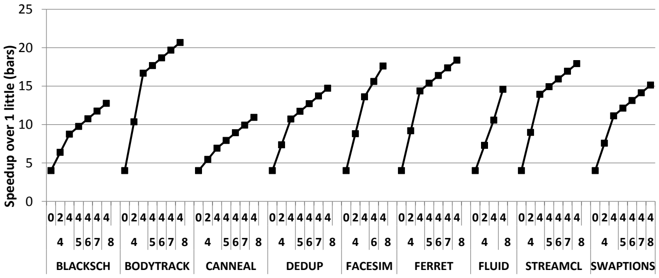

Fig. 2. Ideal speedup over 1 little core according to the performance ratio observed for each application. Results are reported for systems with 0, 2 and 4 big cores, and a total number of cores between 4 and 8.

tion according to the number of big (B) and little (L) cores.

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

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 and zero dependencies between tasks. Thus these speedups are unachievable, but we report them for reference. As expected, benchmarks with higher performance ratios can achieve potentially higher speedups on the asymmetric multi-core.

V. EVALUATION

As described in the previous section, we measure the execution time, power, energy and EDP of nine applications from the PARSEC benchmark suite [27]. 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*: dynamic coarse-grained OS scheduling using the Global Task Scheduler integrated in the Linux kernel [19], [26] using the default PARSEC benchmarks.
- *Task-based*: dynamic fine-grained scheduling at runtime level with the task-based implementations of the benchmarks provided in PARSECSs [28].

A. Exploiting Parallelism in Asymmetric Multi-Cores

This section examines the opportunities and challenges that current asymmetric multi-cores 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*). The OS-based dynamic scheduling (denoted *GTS*) and the task-based runtime dynamic scheduling (denoted *Task-based*) are also evaluated for the same applications. The system configurations evaluated in this section are the following:

- Four little (in-order) cores (0+4)
- Two big (out-of-order) and two little (in-order) cores (2+2)
- Four big (out-of-order) cores (4+0)

For these configurations, Figure 3 shows the execution time 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 can see that 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 the approaches; this shows that all the parallelization strategies are effective

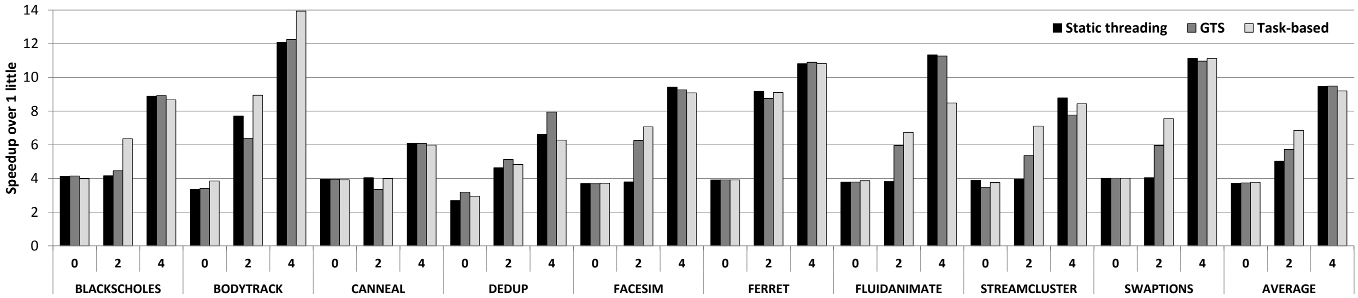

Fig. 3. Execution time speedup over 1 little core for systems that consist of 4 cores in total with 0, 2 and 4 big cores.

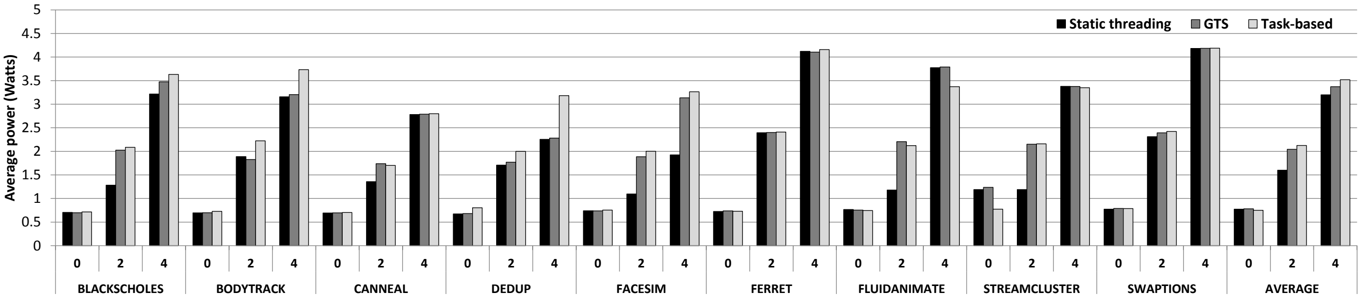

Fig. 4. Average power measurements on a 4-core system with 0, 2, and 4 big cores.

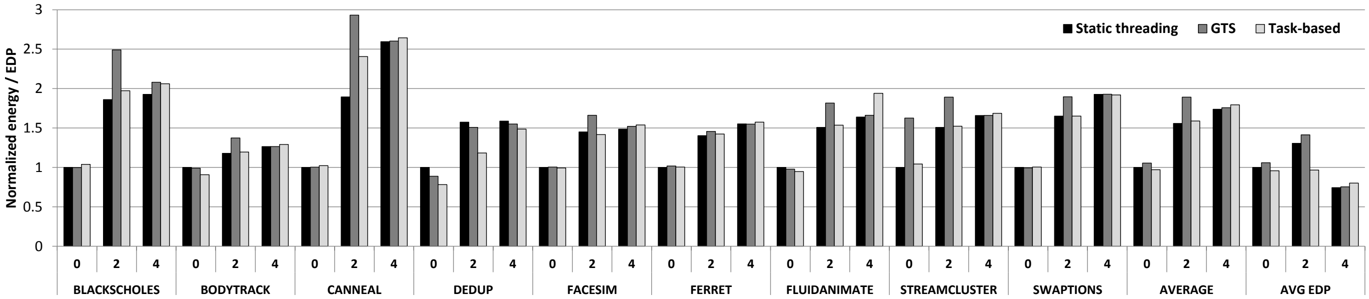

Fig. 5. Normalized energy consumption on a 4-core system with 0, 2, and 4 big cores. Results are normalized with respect to static-threading energy consumption with 4 little cores. Average results for EDP normalized to the same configuration are also included.

for this core count. In the asymmetric configuration, *Static threading* slightly improves the 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 [6]. 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, the configuration with the highest average power dissipation. At this point, it is interesting to note that 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. It is worth noting that in configuration 2+2, *Task-based* achieves the same EDP results than 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 approach is effectively utilizing the little cores.

Thus, if the number of cores remains constant, the best configuration in terms of performance and EDP corresponds to a symmetric configuration with big cores (e.g. 0+4), while in terms of power and energy the best configuration corresponds to a symmetric configuration with little cores (e.g. 4+0).

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-

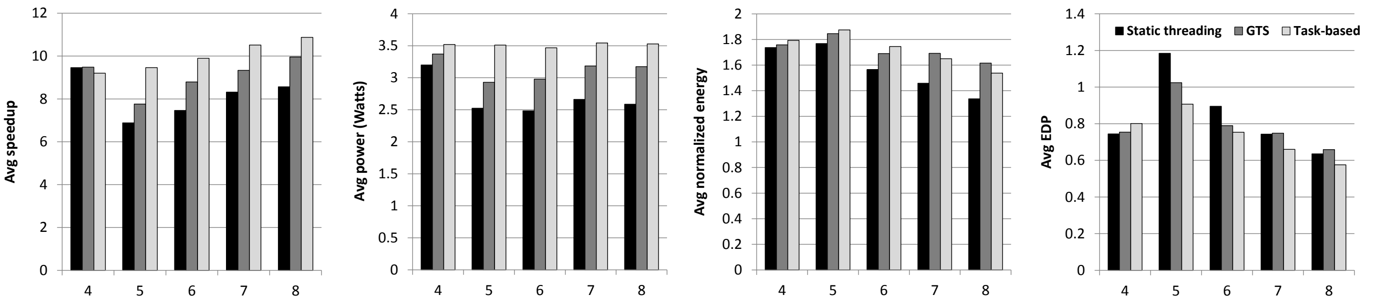

Fig. 6. Average results when running on 4 to 8 cores with 4 of them big. Speedup are normalized over 1 little core, power is measured in Watts, energy consumption is normalized with respect to 4-little static threading configuration, and EDP is normalized with respect to the same configuration.

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 asymmetric 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*. It is clear that 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 on 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. A suitable alternative consists in relying on the OS system or the runtime to dynamically schedule tasks on the asymmetric processor, reaching much better performance, power and energy results in these approaches, especially in the case of the *Task-based* solution.

B. Adding Little Cores to a Symmetric Multi-core

In the following experiments, we explore if an application running on a symmetric multi-core 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 asymmetry in the evaluated approaches.

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 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 I. 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 significantly reduced in the case of *Static threading* as

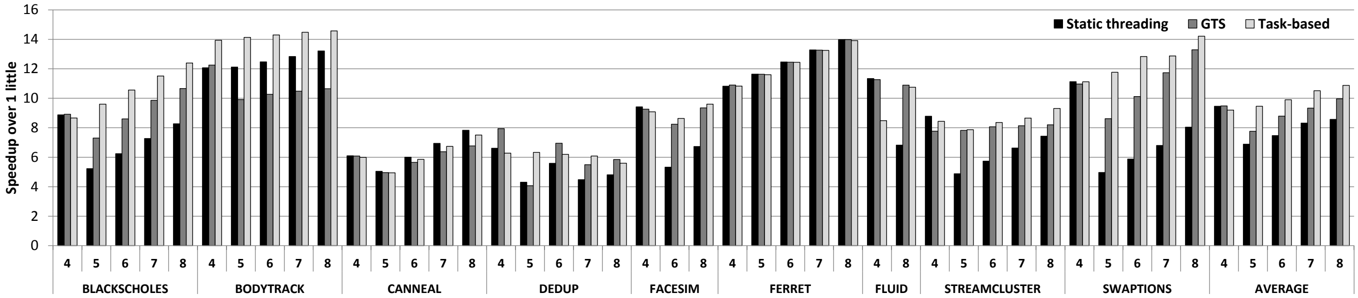

Fig. 7. Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big

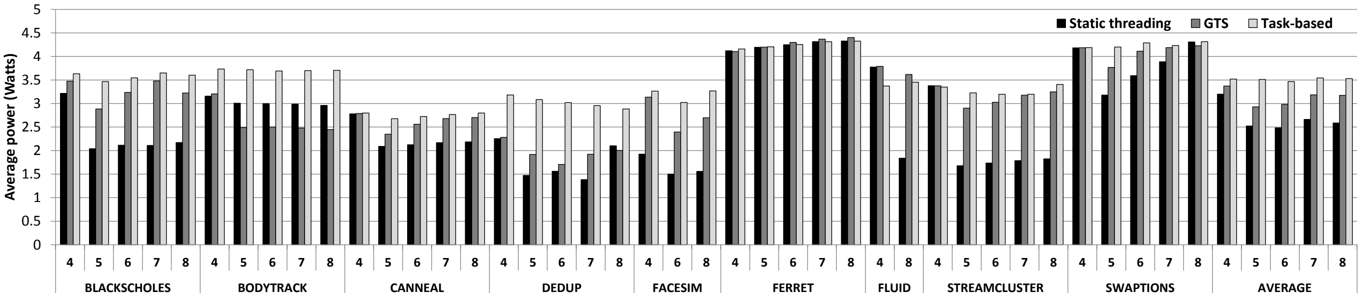

Fig. 8. Average power when running on 4 to 8 cores and 4 of them are big

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 the case of *GTS* and *Task-based*, at least two extra cores are needed to reduce energy. In configuration 4+4, energy is reduced by 31% for *Static threading*, 9.3% for *GTS*, and 16% 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 energy delay product (EDP) on 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 7% while *GTS* and *task based* approaches do so by 10% and 37% respectively.

Figure 7 shows a more detailed exploration of the performance results. According to Table I 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 too high, but as the number of little cores increases, energy is reduced. Moreover, the applications with more advanced load balancing techniques like pipelined parallelism are bodytrack, dedup and ferret. These applications take advantage 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. Kallia: Please check this and let me know if makes sense. I had to write a comment for bodytrack. In the case of dedup, results show more variability. This benchmark is very I/O intensive and, depending on which core executes these I/O operations, performance drastically changes. In oder to deal with this problem, a smarter dynamic scheduling mechanism would be required. Finally, canneal application does not scale according to its ideal speedup reported on Figure 2 as it has a memory intensive pattern which 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 don't spend the same power as *GTS* and *Task-based*. Each one of the pipeline-parallel applications bodytrack and ferret maintains nearly the same power levels among the configurations for each scheduling approach. Dedup is an exeption, 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

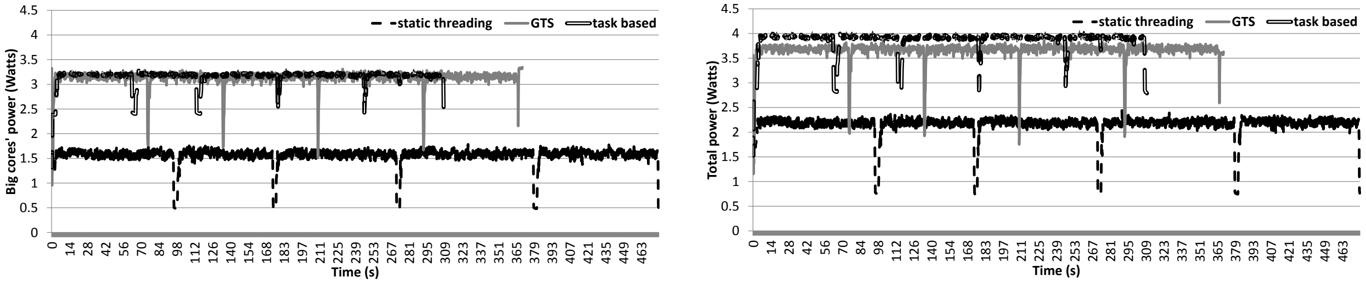

Fig. 9. Power consumption of the Streamcluster benchmark on the 8-core Octodroid Platform. Left: Consumption of the 4 big cores. Right: Consumption of the whole chip.

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 which is the best system configuration for the higest 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.

C. 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 because the power samples are stable and can be plotted with visibility. Figure 9 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 9, 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 task-based programming models against thread-based approaches in asymmetric multi-core systems.

D. Exploiting Assistant Cores for Runtime Activity

This section explores the impact of changing the core that executes the runtime activity in the *Task-based* approach. Similarly to the assistant core in the IBM Blue Gene Q and the Fujitsu SPARC64 XIfx processors [17], [18], we explore the utility of devoting a little core to process this runtime activity. In those systems, the assistant core takes care of managing OS activity and also the MPI runtime, sending and receiving network messages that could pollute the cache hierarchy and bring noise to the parallel application.

In the analyzed task-based programming model, there is a *main thread* that contains most of the programming model's runtime activity. In the evaluated applications, task creation is always done by the main thread as well as task scheduling. This activity can potentially pollute the cache hierarchy and degrade the performance of user tasks.

The default execution mode starts the main thread start executing on core 0. After a barrier is met, depending on the availability of resources, the main thread might migrate to another core. Thus by default, the execution of runtime activities is not tied to one specific core. In the traditional symmetric multi-cores these observations are not important. The most flexible way of main thread migration is the best approach since it is dynamic and does not let cores to remain idle. However, the execution of the runtime activity matters if the system is asymmetric; cores in this case have different performance and energy consumption. This implies complex and interesting trade-offs between the runtime system overhead and the tasks execution time. For example, if the runtime

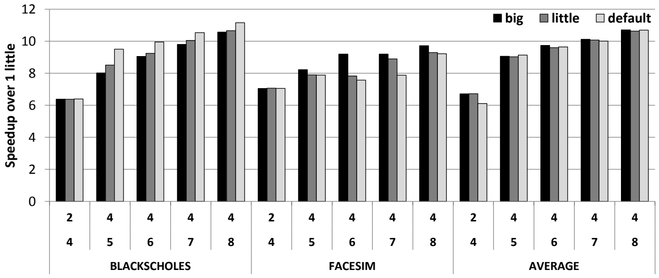

Fig. 10. Speedup obtained from the three different scenarios of the main thread's execution. Bottom line numbers show the total number of cores and upper line numbers show the number of big cores.

system runs on a little core, some tasks will be created under a slow frequency, but tasks will have all the big cores available to run as a counterpart. In contrast, if tasks are created on a big core, they will be able to start execution earlier than in the previous scenario, but one of the big cores is not going to be devoted entirely on executing tasks. To explore this, we perform an evaluation comparing the following three possible scenarios for the main thread's execution:

- Tying the main thread on a *big* core: the runtime activity is forced to execute on a big core.
- Tying the main thread on a *little* core: the runtime activity is forces to execute on a little core.
- The *default* main thread dynamic migration policy: the runtime activity is executed on any available core.

In each case the cores responsible for the execution of the runtime are not exclusively dedicated to the runtime, but they are allowed to execute tasks and serve as assistant cores.

Figure 10 shows the obtained speedup from the above scenarios. The experiments were conducted on all the nine applications of the evaluation, but we only show results from blackscholes and facesim applications as well as the average speedup of all the applications of this paper. The blackscholes results clearly indicate that the application does not benefit from the big or little policies. When running on 5 cores (4+1 configuration), the default policy provides a speedup of 9.5× while the other two are below 8.3×. This application benefits from the default policy where runtime activity takes place on the first available core. Forcing the runtime to execute on a specific core can lead to increased idle time of the main thread while it is waiting for the specific core to finish with the execution of tasks.

The facesim results show that the migration of the main thread on big provides significant benefits: when running on 6 cores, 4 big and 2 little, the big policy provides a speedup beyond 9×, while the default policy just provides speedup below 8×. The reason why facesim's behaviour is opposed to the one seen in case of blackscholes relies on the idle time of the big cores. The overall scalability of facesim is limited compared to blackscholes: it has frequent barriers followed by short sequential phases. As a result, big cores are not fully-utilized. Therefore, it is beneficial to assign the runtime activities to a big core.

The average results of all the applications do not provide any significant gain due to the big and little policies since the speedups observed for some applications are compensated by the slowdowns seen in some others. In general, the noise introduced by the runtime activity to user tasks in the task-based programming model is very reduced. The memory footprint of this activity is very small, as the runtime only accesses some data structures and does not need to buffer large messages as in the case of the MPI runtime. Also, since the task-based runtime is asynchronous and dynamically schedules tasks, it can automatically deal with the load imbalance that it creates. Thus, this activity can be executed on the same core as user tasks without degrading the final performance of the application.

VI. Related Work

Miquel: The past work has predominantly considered sequential applications and has mostly been simulation-based, which limits the size of possible workloads and the ability to accurately measure performance and energy. We found that our conclusions differ as a result of these differences in experimental methods.

There has been a lot of study on asymmetric heterogeneous systems. Some works focus on the design of the system and some works explore the challenges that appear in efficiently utilizing such a heterogeneous system. Kumar et.al [29] presented an idea of an asymmetric single-ISA heterogeneous 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 [2] proposed the use of sampling. The proposed method, minimizes the execution time of each single thread and increases performance. Morad et.al [30] performed a study on asymmetric multi-core systems. This study was based on the area spent on the processor design and their evaluation is using emulation of the heterogeneous system.

Process scheduling on asymmetric systems is one of the most challenging topics in this area of study. Bias [31] scheduling is an operating system scheduler that characterizes the running threads according to their memory or execution intensity. It then schedules the execution intensive threads to the fast cores of the system while the memory intensive threads to the slow 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 [32] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [33] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et.al propose the fairness-aware OS scheduler [34] that focuses on heterogeneous single-ISA architectures. The performance impact estimation (PIE) scheduler [35] 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.

Similar to OS scheduling approaches there have been many task scheduling approaches that are directed for utilizing asymmetric systems. The Levelized Min Time [37] 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 [38] assigns the tasks to the processors according to their *dynamic level* (DL). Heterogeneous Economical Duplication (HED) [39] 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 [40] is designed for asymmetric systems like big.LITTLE and dynamically schedules the *critical* tasks to the fast 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 [41] 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 [42] 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:

- Their experimental evaluation is done through a simulator or emulation of an asymmetric system [29], [2], [30], [31], [34], [35], [36], [38], [37], [39], [41], [42].
- The evaluated applications are either random task dependency graph generators or scientific kernels and microbenchmarks [38], [40], [41], [42].
- Their evaluation does not focus on power and energy consumption [2], [34], [35], [38], [37], [40], [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 [32].

VII. Conclusions

The advantages of asymmetric multi-cores for mobile applications have been proved in the past. In this paper, we show that these systems can also be useful for emerging parallel applications. If the number of cores remains constant, the best configuration in terms of performance, power and energy is always homogeneous. However, asymmetric configurations offer an intermediate solution that can be interesting in some scenarios. Furthermore, if the number of big cores remains constant, having an increasing number of little cores in an asymmetric multi-core can help increasing performance while reducing final energy.

However, more advanced techniques are required at the application, runtime or OS levels to balance the load across all cores and fully exploit the available resources. Relying on the programmer to develop an application that adapts to asymmetric multi-cores complicates the design and deployment of the application. For this reason, many current parallel applications do not adapt well to these systems. OS-based dynamic schedulers can help mitigating the problems of such applications. However, our results suggest that it is critical to use a flexible parallel programming model to allow the runtime system layer to make decisions dynamically. Having application-agnostic mechanisms to properly balance the load or to perform heterogeneity-aware domain decomposition (i. e. assign more load to more powerful hardware components), allows to fully exploit the potential of asymmetric multi-core processors.

When applying these techniques, average performance benefits of XX% are obtained, with XX% reduction in final energy consumption. Taking into account these results, we envision that future desktop and server processors will incorporate heterogeneous cores in their designs.

References

[1] 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," University of Notre Dame, CSE Dept., Tech. Rep., 2008.

[2] 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*, 2004, pp. 64–75.

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

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

[5] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto, "Maximizing Power Efficiency with Asymmetric Multicore Systems," *Communications of the ACM*, vol. 52, no. 12, 2009.

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

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

[8] ——, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *ISCA*, 2013, pp. 154–165.

[9] 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.

[10] 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.*, vol. 20, no. 3, pp. 404–418, 2009.

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

[12] 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*, vol. 21, 2011.

[13] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "Codelet" program execution model for exascale machines: Position paper," in *EXADAPT*, 2011, pp. 64–69.

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

[15] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011, pp. 1–11.

[16] H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos, "Deterministic scale-free pipeline parallelism with hyperqueues," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 32:1–32:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503233

[17] R. Haring, "The Blue Gene/Q Compute Chip," *HotChips*, 2011.

[18] T. Yoshida, "SPARC64 XIfx: Fujitsu's Next Generation Processor for HPC," *HotChips*, 2014.

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

[20] K. Krewell, "Cortex-A53 is ARM's next little thing," 2012. [Online]. Available: www.mpronline.com

[21] J. Bolaria, "Cortex-A57 extends ARM's reach," 2012. [Online]. Available: www.mpronline.com

[22] M. Demler, "Cortex-A72 takes big step forward," 2015. [Online]. Available: www.mpronline.com

[23] ARM, "Cortex-A7 MPCore, revision: r0p3," 2011. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464d/DDI0464D_cortex_a7_mpcore_r0p3_trm.pdf

[24] J. Turley, "Cortex-A15 eagle flies the coop," 2011. [Online]. Available: www.mpronline.com

[25] ARM, "Cortex-A15 technical reference manual, revision: r2p0," 2011. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438c/DDI0438C_cortex_a15_r2p0_trm.pdf

[26] M. K. Hongsuk Chung and H.-D. Cho, "Heterogeneous multi-processing solution of exynos 5 octa with ARM big.LITTLE technology," *Samsung White Paper*, 2013.

[27] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[28] 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.

[29] 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*, 2003, pp. 81–92.

[30] 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.*, vol. 5, no. 1, pp. 4–17, Jan. 2006.

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

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

[33] 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*, 2012, pp. 1–8.

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

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

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

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

[38] G. Sih and E. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, 1993.

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

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

[41] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, 2002.

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