Energy efficiency optimization of task-parallel codes on asymmetric architectures

Luis Costero, Francisco D. Igual, Katzalin Olcoz and Francisco Tirado Departamento de Arquitectura de Computadores y Automática Universidad Complutense de Madrid Email: {lcostero, figual, katzalin, ptirado}@ucm.es

Abstract—We present a family of policies that, integrated within a runtime task scheduler (Nanox), pursue the goal of improving the energy efficiency of task-parallel executions with no intervention from the programmer. The proposed policies tackle the problem by modifying the core operating frequency via DVFS mechanisms, or by enabling/disabling the mapping of tasks to specific cores at selected execution points, depending on the internal status of the scheduler. Experimental results on an asymmetric SoC (Exynos 5422) and for a specific operation (Cholesky factorization) reveal gains up to 29% in terms of energy efficiency and considerable reductions in average power.

Task parallelism; runtime task schedulers; asymmetric architectures; energy efficiency; DVFS

I. INTRODUCTION

Asymmetric Multiprocessors (AMPs) are a class of heterogeneous parallel architectures in which cores that implement different microarchitectures share a common ISA (Instruction Set Architecture) and, possibly, a subset of memory resources. Typically, the available architectural heterogeneity is exploited pursuing energy efficiency and performance restrictions on heterogeneous software environments. One of the most popular implementations of AMPs is the big.LITTLE architectural paradigm present in many ARM SoCs (Systems-on-chip), that combines a number of high performance ARM Cortex-A57/A15 BIG cores with a (possibly different) number of energy-efficient ARM Cortex-A53/A7 LITTLE cores. Leveraging low-power architectures to the HPC (High Performance Computing) arena is one of the main trends in the road towards the Exaflop barrier. Among them, ARM Cortex-A processors, and more specifically, asymmetric SoCs based on this microarchitectural family, are nowadays on the spotlight as the most promising architectures to achieve such a goal.

However, increasing the heterogeneity entails a nonnegligible impact on the programmability of such platforms. In the last decade, task-parallel programming models have emerged as an interesting solution that combines a correct orchestration of parallel programs and a reduced impact on the complexity of the parallel versions of existing or new codes. These models aim at casting a complete computation in terms of discrete pieces of code (*tasks*) with data dependences among them with the aid of task annotations provided by the programmer, and rely on a *runtime task scheduler* (or just *runtime* in the following) that orchestrates the correct ordering of tasks execution as dependences are satisfied at run time.

The extension of these programming models and associated runtimes to heterogeneous architectures, managing data coherency and data transfers among isolated memory spaces has been implemented in a number of software efforts, together with techniques that drive to performance gains in multi-core, many-core, accelerator-based and distributed-memory architectures. The necessary efforts to adapt these programming models to AMPs is also a topic of interest of recent works [1]-[5], pursuing the goal of boosting performance by correctly mapping critical tasks to the most appropriate element of the asymmetric architecture. These works complement energyefficiency studies specifically targeting asymmetric architectures [6], [7]. However, the impact and possibilities of *task* schedulers in terms of improving energy efficiency of taskparallel implementations has not been previously studied in such a level of detail. As of today, similar efforts, together with their impact on performance and energy efficiency, have not been ported or adapted to AMPs.

In this paper, we propose an extension of Nanox, the runtime task scheduler underlying the OmpSs [8] programming model that pursues the goal of reducing energy consumption with minimal impact on performance and programmability. We introduce a set of policies that modify both task scheduling algorithms and frequency of operation of modern AMPs via DVFS depending on the internal status of the task scheduler, and evaluate their impact on both performance and energy efficiency on a Cholesky factorization (a widely used routine in many problems that arise in science and engineering, and illustrative of others DLA (dense linear algebra) implementations with similar features) and an implementation of the big.LITTLE architecture (a Samsung Exynos 5422 SoC).

The rest of the paper is structured as follows. Section II reviews the state-of-the-art in modern task-parallel programming models and their adaptation to asymmetric architectures. Section III describes a number of energy-aware policies and mechanisms that pursue an improvement in performance and energy-efficiency of Nanox on AMPs. Section IV reports the impact of the aforementioned policies in terms of performance and energy efficiency on the Exynos SoC. Section V closes the paper with general remarks and future work.

II. RUNTIME-BASED PARALLEL EXECUTION ON ASYMMETRIC PLATFORMS

A number of task-based programming models have previously proved to be an efficient solution towards the exploitation of parallelism on multi-core, many-core and heterogeneous architectures. In general, these models provide a mechanism to annotate sequential codes and to indicate potential points of parallelism, that is exploited at runtime by a task scheduler that takes care of data dependences among tasks and a proper task-to-processor mapping, typically improved by heuristics. Among others, following the path pioneered by Cilk [9], efforts like StarPU [10], Superglue [11], 18 QUARK [12], Kaapi [13], and OmpSs [8] pursue a common ¹⁹₂₀ goal: extracting and exploiting task parallelism on modern parallel architectures with minimal intervention of the programmer.

OmpSs is one of the most widely accepted programming models nowadays. At a glance, this programming model is based on the inclusion of directives (pragmas) similar to those used in OpenMP, that annotate specific sections of codes as tasks, that is, minimum scheduling units to the available execution resources or processors. These annotations include information about operands and their directionality (input, output and input/output). At runtime, this information is handled by a task scheduler (named Nanox) that maps each task to the most appropriate computational resource available as the inferred data dependences are satisfied.

A. A driving example: the Cholesky factorization

In the following, we employ the Cholesky factorization of a dense matrix as an illustrative example of the necessary modifications required by OmpSs to extract and exploit the available task parallelism in a specific operation. Given a symmetric positive definite matrix A of dimension $n \times n$, the Cholesky factorization decomposes it into $A = U^T U$, where the Cholesky factor U is an upper triangular matrix. Listing 1 sketches a C implementation of a blocked Cholesky implementation for a blocked matrix A composed of $s \times s$ blocks of dimension (block size) $b \times b$ each. Note that the routine decomposes the global operation into a collection of basic kernels or fundamental operations, namely: po_cholesky (Cholesky factorization of the diagonal block); tr solve (solution of a triangular system); ge_multiply (general matrix-matrix multiplication); and sy update (symmetric rank-b update).

These are the fundamental parts of the overall computation, or tasks. Obviously, provided each task is internally executed in a sequential fashion, the aforementioned code would not extract any further level of parallelism. Listing 2 includes the necessary modifications in the definitions of each task in order to exploit the OmpSs programming model and, thus, to extract task parallelism in a transparent manner. Note how each task is annotated with the corresponding pragma omp task directive, including the directionality of each operand involved in the computation. At runtime, the invocation of each task in Listing 1 is intercepted by the runtime task

```
void cholesky (double *A[s][s], int b, int s) {
   for (int k = 0; k < s; k++) {
     // Cholesky factorization
      // (diagonal block)
     po_cholesky (A[k][k], b, b);
      for (int j = k + 1; j < s; j++)
         // Triangular system solve.
         tr_solve (A[k][k], A[k][j], b, b);
      for (int i = k + 1; i < s; i++) {
         for (int j = i + 1; j < s; j++)
            // Matrix-matrix multiplication.
            ge_multiply (A[k][i], A[k][j],
                         A[i][j], b, b);
         // Rank-b update.
         sy_update (A[k][i], A[i][i], b, b);
      1
  }
```

2

3

4

}

Listing 1. C implementation of the blocked Cholesky factorization.

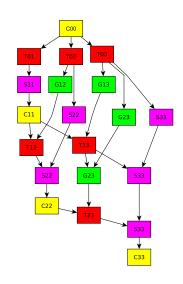


Fig. 1. DAG with tasks and data dependences extracted from the application of the code in Listing 1 on a matrix with 4×4 blocks (s=4). The labels in the nodes specify the type of kernel/task as follows: "C" for the Cholesky factorization; "T" for the triangular system solve; "G" for the matrix-matrix multiplication, and "S" for the symmetric rank-b update. The subindices (starting at 0) specify the submatrix updated by the corresponding task.

scheduler (Nanox), that dynamically builds a DAG (Directed Acyclic Graph) as the one shown in Figure 1, including tasks (nodes) and data dependences among them (edges). Only when all data dependences for a given task are satisfied, the runtime dispatches that task to an available processor, effectively exploiting task parallelism.

B. Asymmetry-aware task schedulers

The design of efficient task scheduling algorithms on multicore and heterogeneous systems has been extensively studied in the past. Some of these works have been recently extended in order to accommodate AMPs as the target platform. Examples of these efforts are CATS [1] or CPATH and HYBRID [2], which aim at dynamically identifying which

```
#pragma omp task inout([b][b]A)
    void po_cholesky (double *A, int b, int ld) {
3
                       INFO = 0;
     static int
     static const char UP
4
                            = 'U';
6
     // LAPACK Cholesky factorization
     dpotrf (&UP, &b, A, &ld, &INFO);
8
10
    #pragma omp task in([b][b]A) inout([b][b]B)
    void tr_solve (double *A, double *B, int b, int ld) {
     static double DONE = 1.0;
static const char LE = 'L', UP = 'U',
TR = 'T', NU = 'N';
13
15
     // BLAS-3 triangular solve
16
17
     dtrsm (&LE, &UP, &TR, &NU, &b, &b,
18
            &DONE, A, &ld, B, &ld);
19
    }
20
    #pragma omp task in([b][b]A,[b][b]B) inout([b][b]C)
    23
24
     static double
                       DONE = 1.0, DMONE = -1.0;
     static const char TR = 'T', NT
                                          = 'N';
26
27
     // BLAS-3 matrix multiplication
     dgemm (&TR, &NT, &b, &b, &b,
            &DMONE, A, &ld, B, &ld, &DONE, C, &ld);
29
30
    }
31
    #pragma omp task in([b][b]A) inout([b][b]C)
32
33
    void sy_update (double *A, double *C, int b, int ld) {
                     DONE = 1.0, DMONE = -1.0;
ar UP = 'U', TR = 'T';
34
     static double
35
     static const char UP
36
37
     // BLAS-3 symmetric rank-b update
38
     dsyrk (&UP, &TR, &b, &b,
39
            &DMONE, A, &ld, &DONE, C, &ld);
40
```

2

5

7

9

11

12

14

21 22

25

28

Listing 2. Annotated tasks for the blocked Cholesky factorization.

tasks belong to the critical path of the DAG, assigning them to the fastest cores, thus reducing the total execution time.

In OmpSs the CATS implementation is called BOTLEV (Bottom level-aware scheduler), and it has been used as a starting point for our work. BOTLEV dynamically detects the longest path of the DAG, assigning those tasks that belong to it to the fast cores of the system. In order to determine which tasks belong to the longest path, each task is internally annotated with the longest distance between itself and a leaf task. Each time a new task is inserted into the DAG, all of its predecessor nodes in the graph are updated only if the longest path increases; proceeding this way, the longest distance between each task and a leaf node is always updated.

When a task becomes ready for execution, it is classified as critical or non-critical based on the annotated distance: if it belongs to the longest known path, it is stored as a critical task. Ready critical and ready non-critical tasks are stored in two different priority queues sorted by its annotated distances. When a core becomes idle, it retrieves a ready task depending on the kind of core: BIG cores execute ready tasks stored in the critical queue, and LITTLE cores retrieve tasks from the noncritical queue. BOTLEV enables work stealing for BIG cores by default, allowing BIG cores to execute non-critical tasks if the critical-queue is empty. Optionally, work stealing can be activated in a bi-directional fashion.

Exynos 5422 System-on-Chip											
Cortex-A 15 quad CPU	Cortex-A7 quad CPU										
Cortex-A15 32+32KbL1 Cortex-A15 32+32KbL1	Cortex-A7 32+32KbL1 Cortex-A7 32+32KbL1										
Cortex-A15 32+32KbL1 Cortex-A15 32+32KbL1	Cortex-A7 32+32KbL1 Cortex-A7 32+32KbL1										
DVFS domain (800, 900, 1000, 1100, 1200, 1300) MHz	DVFS domain (800, 900, 1000, 1100, 1200, 1300) MHz										
2 Mb L 2 cache	512 KB L2 cache										

Fig. 2. Samsung Exynos 5422 SoC employed in our experiments.

C. Target asymmetric architectures

The target architecture for our experiments is an ODROID-XU3 board comprising a Samsung Exynos 5422 SoC with an 32-bit ARM processor and 2GB DDR3 RAM. The chip features an ARM Cortex-A15 quad-core processing cluster and a Cortex-A7 quad-core processing cluster. Each ARM core (either Cortex-A15 or Cortex-A7) has a 32+32-KByte L1 (instruction+data) cache. The four A15 cores share a 2-MByte L2 cache, while the four A7 cores share a smaller 512-KByte L2 cache. All cores of the same cluster share the same frequency of operation, clocking from 800MHz to 1300MHz in steps of 100MHz in both cases. The board exposes independent power measurements for each cluster. Figure 2 shows a schematic view of the Exynos SoC.

III. PROPOSED ENERGY-AWARE POLICIES

We introduce two different general approaches that pursue an improvement on the energy efficiency of task-parallel codes on asymmetric architectures. The first group of policies (named as FS) is based on the dynamic application of DVFS techniques at runtime. The goal is to integrate these techniques on an asymmetry-aware scheduler, and to reduce energy consumption by modifying the frequency of one of the clusters based on the internal state of the scheduler, without further modifications on the scheduling algorithm. Pursuing the same goal, the second group of policies (named TS) implements different asymmetry-aware scheduling algorithmic variations on existing task schedulers.

A. Policies based on frequency scaling (FS)

Applying DVFS techniques to a task-parallel problem requires three main runtime decisions to be made, namely: (a) which frequencies (among those available) to use; (b) at which moments of the parallel execution these changes need to be made; and (c) which elements of the architecture (among those that support DVFS) are affected by the voltage/frequency scaling. The set of frequencies that a processor can run at is usually defined by the architecture, so the first decision is reduced to choosing between using all the available frequencies or just a subset of them. The remaining decisions are directly related to the specific problem to tackle, and the knowledge that the task scheduler has of it.

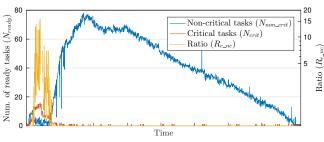
Figure 3a shows, for a Cholesky factorization of a $1024 \times$ 1024 matrix divided in blocks of dimension 64×64 , the evolution in time of the amount of critical and non-critical tasks ready for execution (N_{crit} and N_{non_crit} , respectively, being $N_{ready} = N_{crit} + N_{non_crit}$), together with the ratio between them ($R_{c_nc} = N_{crit}/N_{non_crit}$). In the following, we also consider N_{max}^{nc} and N_{max} as the maximum amount of ready non-critical tasks and ready tasks (critical and non-critical) observed from the beginning of the execution at each moment. Both values, N_{max}^{nc} and N_{max} , are constantly monitored and updated at runtime by the scheduler. Finally, $R_{non_crit} = N_{non_crit}/N_{max}^{nc}$ denotes the ratio of non-critical ready tasks compared with the maximum amount observed for this value.

1) Policy FS1. Tasks limited by the critical path: Runtime task schedulers annotate tasks while the DAG is built and. typically, no further external information is used; thus, it is possible that multiple paths of the DAG are detected as critical at the same time. On an asymmetry-aware scheduler like BOTLEV, this situation entails that most of the tasks will be executed by BIG cores (as they are annotated as critical), while LITTLE cores will be in idle state until new non-critical tasks are detected. Asymmetry-aware task schedulers alleviate these situations by allowing critical tasks to be executed by both types of cores until new non-critical tasks are ready to run. However, using LITTLE cores to execute critical tasks can slow down the execution as, despite the fact that tasks can start their execution earlier due to the greater number of available cores, running a task on a slow core can increase its execution time meaningfully.

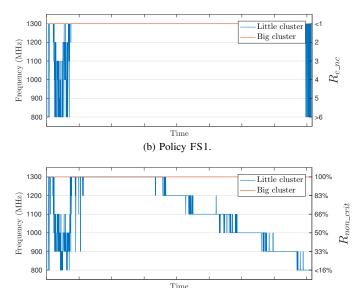
Our approach to respond to this situation is different, as is our goal (reducing energy consumption): the FS1 policy leverages these moments –where the number of ready critical tasks is greater than the number of ready non-critical tasks– to reduce power consumption by decreasing the frequency of the LITTLE cluster. The side effect is that the execution time of non-critical tasks increases, but as the global execution time is limited by the critical tasks executed on the BIG cluster, delaying the execution of non-critical tasks on these moments should not dramatically impact the global performance.

In FS1, the decision on which frequency the LITTLE cluster should run at is made by the scheduler each time the number of ready tasks changes (i.e., when a task becomes ready or a ready task is executed by an idle core), and it is based on the *relation* between the sizes of both queues (R_{c_nc}), that determines the specific frequency step that will be applied to the LITTLE cluster. For example, if $R_{c_nc} == 2$, the LITTLE cluster will run at its second maximum available frequency (in this case, 1200MHz); if $R_{c_nc} == 5$, the cluster will run at its fifth maximum frequency available (in this case, 900MHz).

Figure 3b reports the instantaneous frequency applied by the task scheduler when applying FS1 on the same execution as that shown in Figure 3a. Observe how, when the number of ready critical tasks is higher than the number of ready non-critical tasks (e.g. at the beginning and end stages of the execution in this example), the frequency of the LITTLE cluster is scaled down, and how the frequency chosen for the



(a) Evolution of the number of critical and non-critical ready tasks.



(c) Policy FS2. Notice that policy FS3 will have the same behavior, but applied to the other cluster.

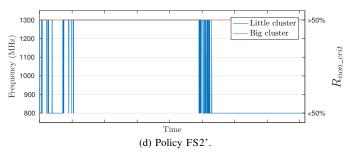


Fig. 3. Behavior of each FS policy when is applied to a Cholesky factorization of a 1024×1024 matrix divided in blocks of 64×64 elements.

cluster is directly related with R_{c_nc} . Also, note how, when N_{non_crit} increases, the policy forces the LITTLE cores to run at a higher (even at the maximum) frequency.

2) Policies FS2 and FS2'. LITTLE cluster frequency scaled based on the workload: Instead of modifying the frequency based on the ratio between the number of both types of ready tasks, policies FS2 and FS2' modify the frequency based on the absolute amount of non-critical tasks at each moment, i.e., if there is a high number of non-critical tasks, the LITTLE cluster will run at a high frequency, and if the number is low, the frequency will be lower.

In order to determine when the number of non-critical tasks is considered high or low, N_{non_crit} is compared with N_{max}^{nc} . If higher, FS2 and FS2' will consider that the number of non-critical tasks is high, and the LITTLE cluster will run at its maximum frequency; if not, frequency is scaled down depending on the value of R_{non_crit} .

The difference between FS2 and FS2' is the set of frequencies to select: while FS2 chooses one between all the available frequency steps according to R_{non_crit} (see Figure 3c), FS2' only uses the highest and lowest available frequencies (see Figure 3d). In this case, if the current number of non-critical tasks is lower than the 50% of the maximum amount recorded (that is, if $R_{non_crit} < 0.5$), the frequency will be the lowest available, in other case, it will be the highest.

Observing the evolution of N_{crit} and N_{non_crit} in Figure 3a, two different phases can be distinguished: a first phase where the number of ready non-critical tasks increases, and a second phase where it decreases. This behaviour matches with a Cholesky factorization *DAG*, which enlarges very fast at the beginning, and it reduces slowly later. While the first phase occurs, the maximum amount of ready non-critical tasks is growing, so the frequency which LITTLE cluster is running at is its maximum frequency; during the second phase, the scheduler scales down frequency based on the amount of non-critical tasks and available frequencies.

3) Policy FS3. BIG cluster frequency scaled based on the workload: The behavior of policy FS3 is similar to that of FS2, but, instead of modifying the frequency of the LITTLE cluster, FS3 scales the frequency of the BIG cluster.

B. Policies based on task scheduling (TS)

The TS policies described next are based on the same ideas as FS policies but, instead of applying DVFS techniques, they decide at runtime the phases in which both clusters are considered to execute tasks, or just one of them is used as a scheduling target. On one hand, using only one of the clusters in specific moments means that power consumption is likely to decrease, but on the other hand, performance will also be affected. Our goal is to find a trade-off between both parts, and thus to improve energy efficiency.

1) Policies TS1 and TS2. Making cluster unusable depending on the workload: Similar to policies FS2 and FS3, these policies track the value of N_{ready} at each moment, and determine when the amount of tasks is increasing or decreasing (comparing this value with N_{max}). If the number of ready tasks is low enough, the policy will not assign any new task to one of the clusters, making it to be in an idle state from the scheduler's perspective, and saving power consumption. If the number of tasks increases later, the cluster becomes available again and it will execute new tasks as they become available. The amount of tasks (or threshold) that determines when to disable or enable the cluster (denoted as N_{thres} in the following) is configurable and not defined by the policy; several experiments with different values for N_{thres} can be found in the next section.

The difference between policies TS1 and TS2 is that, while TS1 acts on the LITTLE cluster, TS2 acts disabling and enabling the BIG cluster. As TS2 disables the BIG cluster in

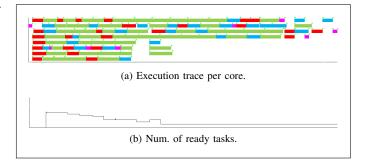


Fig. 4. Policy TS2: task scheduling based on the number of ready tasks, for a Cholesky factorization of a square 4096×4096 elements matrix, grouped in square blocks of 512×512 elements each executed on an ODROID platform. Color key: red=TRSM, pink=POTRF, blue=SYRK, green=GEMM, white=IDLE.

some moments of the execution, critical tasks are executed on LITTLE cores until the BIG cluster is enabled again.

Figure 4 shows an execution of policy TS2 applied to a Cholesky factorization, where the cluster is disabled when the current number of ready tasks is under the 30% of N_{max} (that is, $N_{thres} = 30\%$). Each line in the trace corresponds to a specific core executing tasks (coloured areas) or in idle state (white areas). The trace has been obtained on an ODROID platform, where cores (numbered from the top to the bottom) 0-3 belong to the LITTLE cluster, and cores 4-7 to the BIG cluster. The plot at the bottom shows the number of ready tasks at each moment. Observe how, at the beginning, the task scheduler assigns tasks to all the available cores, until the number of ready tasks is under 30% of maximum recorded; from that moment on, no tasks are assigned to BIG cluster. As there are less cores to execute ready tasks, in some moments of the execution the number of ready tasks becomes greater than N_{thres} , starting BIG cores to execute ready tasks until number of ready tasks decreases again and the cluster becomes unavailable for scheduling purposes.

2) Policy TS3. Cluster disabled based on workload: Some platforms allow switching off one of the clusters under demand via the OS, which entails an important decrease on power consumption, as shown in Figure 5. Policy TS3 is similar to policy TS2, but in addition to deactivating the BIG cluster to the task scheduler, it switches it off completely¹.

IV. EXPERIMENTAL RESULTS

In the following, we report the experimental results obtained for the Cholesky factorization on the ODROID SoC applying the proposed policies. In all cases, we show results for performance (in terms of GFLOPS), average power (in Watts) and energy efficiency (in GFPLOPS/Watt). All experiments were carried out using single precision and gathering power results from the internal meters in the board. Each experiment was repeated ten times, showing the average measurements in the following.

¹As the Linux Kernel does not allow powering off the core number zero in our platform, experiments related with switching off the LITTLE cluster could not be performed.

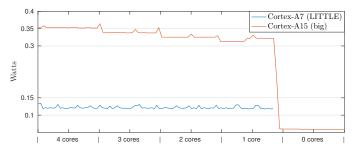


Fig. 5. Power consumption of each cluster on idle state with different number of active cores. Linux kernel does not allow switching off the whole LITTLE cluster, thus measures could not be made for this scenario.

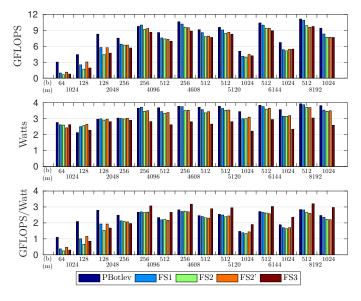


Fig. 6. Experimental measures for policies from FS1 to FS3 on an ODROID platform. Policy PBOTLEV stands for a normal execution using the asymmetry-aware scheduler BOTLEV without any policy. Tags in the horizontal axis represent the sizes of the matrix and blocks of each experiment.

A. Policies based on frequency scaling (FS)

Figure 6 shows the results obtained when applying policies from FS1 to FS3 to different Cholesky factorizations on an ODROID platform. The experiments cover a range of different matrix sizes and block dimensions. A number of general, preliminar remarks can be extracted from the results. Depending on the matrix size, the conclusions differ, namely:

- For small matrices, $(m \le 2048)$, there is a considerable difference between the performance obtained when the factorization is made without any policy (named PBOTLEV in the Figures) and when using any of our policies. This big difference in the performance has a huge impact on the energy efficiency.
- For large matrices ($m \ge 4096$), applying our policies also implies a penalty in performance, as expected. However, energy efficiency measurements are very similar to PBOTLEV. In this case, FS3 clearly outperforms PBOTLEV in terms of energy efficiency. In addition, as a positive side effect and for this range of problem sizes, the

 TABLE I

 Improvement of average power consumption (in Watts) for policies from FS1 to FS3.

	Matrix size $(m \times m)$ and block size $(b \times b)$.													
(m)	1024		4096		4608		5120		6144		8192			
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024		
FS1	-0.93	-0.30	-0.04	0.17	0.00	0.13	0.11	0.42	0.07	0.39	0.05	0.27		
FS2	-1.22	-0.46	0.15	0.27	0.22	0.30	0.21	0.41	0.22	0.40	0.20	0.34		
FS2'	-0.78	-0.09	0.13	0.22	0.19	0.23	0.20	0.33	0.18	0.35	0.22	0.31		
FS3	-1.09	-0.21	0.73	0.93	0.87	0.96	0.89	1.18	0.85	1.21	0.86	1.21		

TABLE II IMPROVEMENT OF ENERGY EFFICIENCY (IN GFLOPS/WATT) FOR POLICIES FS1-FS3 OF DIFFERENT CONFIGURATIONS OF A CHOLESKY FACTORIZATION ON AN ODROID PLATFORM.

Matrix size $(m \times m)$ and block size $(b \times b)$.													
(m)	1024		4096		4608		5120		6144		8192		
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024	
FS1	-4.83	-3.84	0.04	-0.19	-0.12	-0.07	-0.05	-0.10	-0.06	-0.20	-0.03	-0.12	
FS2	-5.12	-4.87	-0.04	-0.16	-0.11	-0.16	-0.17	-0.16	-0.11	-0.26	-0.18	-0.25	
FS2'	-4.64	-2.49	-0.01	-0.23	-0.16	-0.19	-0.12	-0.04	-0.15	-0.19	-0.25	-0.27	
FS3	-5.01	-4.22	0.41	0.31	0.34	0.43	0.41	0.43	0.31	0.48	0.36	0.50	

application of any FS policy clearly reduces the average power consumption (in Watts) of the execution.

Diving into details of average power and energy efficiency results for each policy, a number of specific insights can be extracted. First, the gap in performance, average power and energy efficiency between policies FS2 and FS2' is not remarkable and, similar to policy FS1, experimental results do not show any improvement in terms of energy efficiency when using these policies for this application and platform. However, a decrease in the power consumption is observed, making these policies of great appeal when targeting environments where the power consumption is limited by design. Table I reports the decrease of power consumption (in Watts) achieved for each policy. In the first set of matrices (the ones with lowest size). the power consumption increases, but, in the second group, the power consumption decreases in all matrix configurations and for all the policies, achieving a decrease up to 0.41 Watts (12.85%) for policies FS2 and FS2', and a decrease up to 1.21 Watts (34.88%) for policy FS3.

Second, the penalty introduced by the application of FS1, FS2 and FS2' in terms of performance does not make up for the improvements in average power introduced by the frequency scaling in those policies. Thus, for this problem, they actually increase the energy consumption of the solution.

Finally, from Figure 6 we can observe that the policy which obtains the best results is FS3, outperforming BOTLEV in terms of energy efficiency. Table II reports a detailed study of the energy efficiency improvement (in GFLOPS/Watt) of each policy and matrix configuration compared with a normal execution using BOTLEV. For matrices larger than 2048 elements, FS3 obtains a rise on energy efficiency, achieving improvements from 11.7% up to 29.3%.

TABLE III Amount of time when the LITTLE cluster is unusable for different configurations of policy TS1 (rows) and Cholesky factorization (columns) in a Juno platform.

	Matrix size $(m \times m)$ and block size $(b \times b)$.													
(m)	1024		40	4096		4608		5120		44	8192			
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024		
50%	69.36	45.83	43.44	50.83	39.39	38.48	40.91	42.07	39.15	35.24	40.32	48.33		
40%	68.20	31.25	29.41	33.33	30.61	32.73	32.73	30.54	32.83	30.42	30.15	37.92		
30%	63.42	34.58	21.14	32.50	20.88	24.85	25.00	28.06	23.08	25.43	21.81	33.75		
20%	20.40	17.92	11.40	31.67	11.97	17.27	18.41	21.04	14.56	19.12	13.11	26.67		
10%	23.10	15.00	5.27	20.00	4.87	11.52	9.55	15.13	7.69	10.3	5.64	12.92		

 TABLE IV

 IMPROVEMENT OF ENERGY EFFICIENCY FOR POLICY TS2.

	Matrix size $(m \times m)$ and block size $(b \times b)$.													
(m)	1024		40	4096		4608		5120		44	8192			
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024		
10%	-3.67	4.07	-0.52	-0.27	-0.52	-0.37	-0.32	0.05	-0.33	-0.08	-0.44	-0.22		
20%	-3.98	1.98	-0.30	-0.16	-0.48	-0.23	-0.25	-0.01	-0.27	0.00	-0.28	-0.13		
30%	-4.04	-0.99	-0.15	-0.12	-0.27	-0.13	-0.12	0.00	-0.15	-0.03	-0.29	-0.03		
40%	-3.69	1.56	0.06	-0.06	-0.15	-0.01	-0.02	-0.02	-0.06	-0.01	-0.19	0.01		
50%	-4.04	-1.47	0.16	0.04	0.06	0.01	0.05	-0.01	0.01	-0.03	-0.11	0.08		

B. Policies based on task scheduling (TS)

Opposite to FS, TS policies do not pre-define a specific moment of the execution in which a cluster is disabled. The experiments described below take into account different configurations of the policies, from disabling the cluster when the amount of ready tasks is 50% of the maximum amount recorded (that is, $N_{thres} = 50\%$), to disabling it when the amount is only at 10%. Note that disabling the cluster when the current number of ready tasks is, for example, half of the maximum amount recorded does not imply that the cluster will be unusable 50% of the execution time. Table III shows the percentage of time in which the LITTLE cluster is unusable for policy TS1, depending on the configuration of the policy and problem dimensions.

Figures 7 and 8 report the behavior of policies TS1 and TS2, respectively, for different matrix sizes and policy configurations, in terms of performance, average power and energy efficiency. The experiments reveal that N_{thres} has a high impact on the final performance, independently of the cluster which is affected by the policy. In general, both policies exhibit worse energetic results than not using any policy. Whereas policy TS2 has similar energy efficiency results than PBOTLEV, the results obtained when TS1 is used are worse than when not using it.

Table IV shows the improvement of GFLOPS/Watt obtained when policy TS2 is compared with a normal execution (policy PBOTLEV). Although this policy does not achieve an improvement in energy efficiency, it obtains similar energy-efficiency measurements with lower overall power consumption, making this policy, together with policies FS2 and FS2', good candidates for scenarios where the power consumption is limited.

Policy TS3 does achieve an improvement in terms of energy efficiency on most of the tested configurations. Figure 9 shows the results obtained when this policy was applied for different

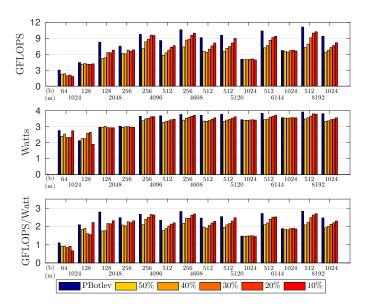


Fig. 7. Experimental results for different TS1 configurations applied to multiple matrix sizes.

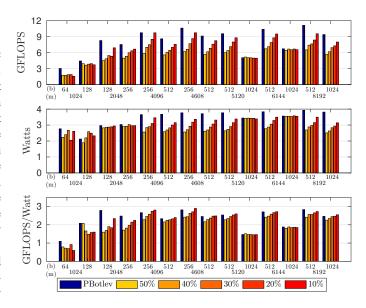


Fig. 8. Experimental results for different TS2 configurations applied to multiple matrix sizes.

problem dimensions. The application of the policy attains an improvement of up to 17.1%. Table V shows the improvements for each configuration in terms of GFLOPS/Watt.

Although policies TS2 and TS3 exhibit similar behavior (policy TS2 does not use BIG cores meanwhile policy TS3 switches them off), the performance obtained is lower for policy TS3. This overhead is probably caused by the OS when it migrates the processes running on a BIG core to a LITTLE one when a complete cluster is switched off (and similarly when it is switched on again). However, due to the considerable decrease in power consumption when the cluster is off (as shown in Figure 5), the decrease in performance does not entail a big impact on the overall energy efficiency.

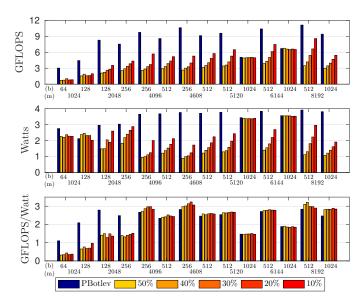


Fig. 9. Experimental results for different TS3 configurations applied to multiple matrix sizes.

TABLE V ENERGY PERFORMANCE IMPROVEMENT (IN GFLOPS/WATT) FOR DIFFERENT TS3 POLICY CONFIGURATIONS COMPARED WITH A NORMAL EXECUTION USING BOTLEV (POLICY PBOTLEV).

	Matrix size $(m \times m)$ and block size $(b \times b)$.													
(m)	1024		4096		4608		5120		6144		8192			
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024		
10%	-4.98	-5.02	-0.16	-0.08	-0.02	0.01	0.02	-0.02	-0.01	0.02	0.24	0.33		
20%	-4.95	-4.70	0.00	-0.05	0.03	-0.03	0.01	0.01	0.01	-0.02	0.35	0.35		
30%	-4.71	-4.83	0.12	0.07	0.16	0.04	0.05	0.01	0.05	-0.05	0.13	0.33		
40%	-4.92	-4.44	0.15	0.02	0.29	0.07	0.09	0.03	0.04	-0.01	0.13	0.41		
50%	-4.89	-3.92	0.06	0.02	0.14	0.06	0.09	-0.01	0.04	-0.03	0.05	0.37		

V. CONCLUSIONS

In this paper we have explored a number of ways to extend an asymmetry-aware scheduler to optimize the energy efficiency of task-parallel applications, focusing on ARM big.LITTLE systems-on-chip. From the observations made for an illustrative dense linear algebra application with complex data dependencies among tasks (the Cholesky factorization), a number of insights have been extracted, namely: (1) scaling the frequency of the LITTLE cluster does not have a positive effect on the energy efficiency, but a reduction in average power consumption is constantly achieved; (2) scaling the frequency of the BIG cluster does achieve considerable improvements on energy efficiency, increasing it up to 29.3%; (3) we have demonstrated that disabling the use of one of the clusters in some moments of the execution also achieves a decrease on power consumption, but not in energy efficiency, unless the switching off of the whole cluster is supported by the hardware and OS, with improvements on energy efficiency of up to 17.1%.

While the Cholesky factorization is representative of common operations in the dense linear algebra field, future work will extend the experimental study to applications with different features, and also to include extended levels of heterogeneity (e.g. including low-power GPUs in the SoC). Automatically predicting optimal policies for a given application/architecture is also in our roadmap.

ACKNOWLEDGMENTS

This work has been supported by the EU (FEDER) and the Spanish MINECO, under grants TIN 2015-65277-R, TIN 2012-32180 and FPU15/02050.

REFERENCES

- [1] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 329–338. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751235
- [2] K. Chronaki, A. Rico, M. Casas, M. Moretó, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task scheduling techniques for asymmetric multi-core systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2016. [Online]. Available: http://doi.acm.org/10.1109/TPDS.2016.2633347
- [3] L. Costero, F. D. Igual, K. Olcoz, S. Catalán, R. Rodríguez-Sánchez, and E. S. Quintana-Ortí, "Refactoring conventional task schedulers to exploit asymmetric arm big.little architectures in dense linear algebra," in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2016, pp. 692–701.
- [4] Q. Chen and M. Guo, "Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures," ACM Trans. Archit. Code Optim., vol. 11, no. 1, pp. 8:1–8:25, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2579674
- [5] C. Torng, M. Wang, and C. Batten, "Asymmetry-aware work-stealing runtimes," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), June 2016, pp. 40–52.
- [6] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "Sparta: Runtime task allocation for energy efficient heterogeneous manycores," in 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Oct 2016, pp. 1–10.
- [7] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proceedings of the 2013 International Conference on Compilers*, *Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 15:1–15:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555729.2555744
- [8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, and L. e. a. P. Martinell, "Ompss: a proposal for programming heterogeneous multicore architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173– 193, 2011.
- [9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958
- [10] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.
- [11] M. Tillenius, "Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C617–C642, 2015. [Online]. Available: http://dx.doi.org/10.1137/140989716
- [12] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2011.
- [13] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proc. IEEE 27th Int. Symp. on Parallel and Distributed Processing*, ser. IPDPS'13, 2013, pp. 1299–1308.