Exploiting Asymmetric Systems with Flexible System Software

Research Plan

Kallia Chronaki
kallia.chronaki@bsc.es
Supervisors : Marc Casas and Rosa M. Badia

Department of Computer Architecture

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center (BSC)

Research plan for

Doctor of Philosophy

April 2018

Table of contents

List of figures

Part I

Prologue

Chapter 1

Introduction

Energy efficiency has become the main challenge for future High Performance Computer (HPC) designs motivating prolific research to face the *Power Wall* [28]. The use of asymmetric multi-core architectures is an approach towards increasing energy efficiency [6, 21, 22, 31] in the HPC domain. These architectures feature different types of processing cores that target different performance and power optimization points.

Asymmetric multi-cores have been successfully deployed in the mobile domain, where simple in-order cores (*little*) have been combined with aggressive out-of-order cores (*big*) 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 big or little cores accordingly [25]. In a second generation of asymmetric multi-core processors, all the cores can run simultaneously to further improve the peak performance of these systems [15].

Many researchers are pushing towards building future parallel systems with asymmetric multicores [21, 22, 26, 27, 43] and even mobile chips [39]. However it is unclear if current parallel scientific applications will benefit from these platforms. Load balancing and scheduling are two of the main challenges in utilizing such heterogeneous platforms as the programmer has to consider the system's asymmetry from the very beginning to obtain an efficient parallelization.

To tackle these issues the programmer has to be aware of the platform's asymmetry and modify the code of the application accordingly so that it achieves load balance. In this scenario, the portability of applications is very limited and the existing scientific applications that are developed to target homogeneous multi-cores face significant performance degradation.

The first step of this thesis will be to characterize the maturity of the asymmetric platforms when running scientific applications and how different scheduling options affect per-

formance and energy efficiency. We evaluate for the first time the suitability of currently available mobile asymmetric multi-core platforms for general purpose computing. We compare three scheduling approaches each of them taking place on a different level of the software stack.

First, is the *application-based scheduling*, where the application is threaded statically by the programmer. We demonstrate that out-of-the box parallel applications are not portable to asymmetric multi-cores. The asymmetry of the system can lead to load imbalance and the user has to define generic and highly sophisticated load balancing mechanisms within the application that need to serve symmetric or asymmetric systems. Additionally, our characterization evaluates the impact of an *OS scheduler* (GTS) [15] that is aware of the platform's characteristics and dynamically migrates the threads to the appropriate type of core to increase performance. Finally, we suggest the use of a *task-based* programming model [9, 18, 35, 48, 53]; task-based programming models with dynamic scheduling allow the specification of inter-task dependencies that enable automatic scheduling and synchronization by the runtime system. We compare these options on the PARSECSs benchmark suite [12] as it offers a set of highly parallel scientific applications. This study will give us a complete picture of the state of the art high-level scheduling approaches in terms of performance and energy efficiency on our ARM big.LITTLE [25] 8-core asymmetric system. More details about this part of work can be found in Section 3.1.

Having the proof that the *task-based* scheduling approach is the most sophisticated one, we plan to further improve it by making it aware of the system's asymmetry. This task, is carried out with the introduction of three asymmetry-aware scheduling policies within a task-based programming model. These scheduling policies have the knowledge of the types of cores of the system and, according to the application characteristics, they schedule tasks either on the fast or the slow cores of the system. More specifically, our task-based scheduling policies separate the tasks into groups of critical and non-critical tasks. Then the fast cores are responsible for the execution of the critical tasks while the slow cores execute the non-critical tasks. The main difference between these scheduling policies, apart from the implementation, is the critical task consideration and it is described in more detail in Section 3.2.

We further plan to modify the runtime system of the task-based programming model so that it takes advantage of the asymmetry of the platform. Specifically, we plan to explore the use of assistant cores for the runtime activity. Since in some cases the runtime introduces overheads, it is useful to devote one core for this activity (symbiotic core). For this, we plan to first evaluate all the available possibilities of the static assignment of the runtime thread to one core of the system. This way we will be aware of the performance and energy outcome

of executing the runtime activity on a big or on a little core. We then plan to implement mechanisms for dynamic runtime thread migration according to the current execution circumstances. This would involve the application's characteristics, such as number of tasks or whether the application is dependency intensive (has many dependencies between tasks).

Finally, we plan to incorporate our scheduling and thread migration techniques into a novel asymmetry-aware task-based runtime system. This runtime system will provide high performance and energy efficiency to the future asymmetric multi-cores and will combine the mechanisms of the use of symbiotic cores together with the scheduling policies to offer high adaptivity and portability to the current parallel applications. The runtime will use one assistant runtime thread that will migrate to either a fast or a slow core of the system according to the current circumstances and, at the same time, will perform scheduling according to the task criticality. We expect that this will significantly boost the portability, performance and energy efficiency of parallel scientific applications on the new asymmetric multi-core systems. We describe the above steps in more detail in Chapter 3.

Chapter 2

State of the Art and Related Work

There has been a lot of studies on asymmetric multi-core systems. Some works focus on the design of the system, while other works explore the challenges that appear in efficiently utilizing such a heterogeneous system. Kumar et al [30] present the idea of an asymmetric multi-core system and proposed a feedback-based way to dynamically migrate processes among the different cores. To determine the core that most effectively executed a workload, Kumar et al [31] proposed the use of sampling. The proposed method minimizes the execution time of each single thread and increases performance. Other studies focused on the pipeline design of such asymmetric systems and the area that should be devoted to each component in the system [6, 34]. Other works on asymmetric systems focus on hardware support for critical section detection [43] or bottleneck detection [26, 27]. These approaches are orthogonal to the approaches evaluated in this paper and could benefit from them to further improve the final performance of the system.

Process scheduling on asymmetric systems is one of the most challenging topics in this area of study. Bias [29] scheduling is an operating system scheduler that characterizes the running threads according to their memory or execution intensity. It then schedules the computation intensive threads to the big cores of the system while the memory intensive threads to the little cores of the system. The experimental evaluation is done on Intel Xeon processors and the heterogeneous system is emulated by changing the configuration of three out of the four cores of the processor. Cong et al propose the Energy-Efficient [16] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [13] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et al propose the fairness-aware OS scheduler [45] that focuses on asymmetric multi-core architectures. The performance impact estimation (PIE) scheduler [46] 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 [41] 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 designed for utilizing asymmetric systems. The Levelized Min Time [24] 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 [42] assigns the tasks to the processors according to their *dynamic level* (DL). Heterogeneous Economical Duplication (HED) [2] duplicates the tasks in order to be executed on more than one core but it then removes the redundant duplicates if they do not affect the makespan. 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 [44] that maintains a list of tasks and statically identifies and schedules the tasks belonging to the critical path to the processor that minimizes the sum of their execution times. The Longest Dynamic Critical Path (LDCP) algorithm [17] identifies the tasks that belong to the critical path and schedules them with higher priority.

All these works reflect the remarkable research that is taking place on asymmetric systems. However we consider that their experimental evaluation is limited for three main reasons:

- Their experimental evaluation is done through a simulator or emulation of an asymmetric system [2, 6, 24, 26, 27, 29–31, 34, 41–43, 45, 46].

- The evaluated applications are either random task dependency graph generators or scientific kernels and micro-benchmarks [17, 42, 44].

- Their evaluation does not focus on power and energy consumption [24, 31, 42, 45, 46].

The search for efficient task scheduling on multi-core systems has been intensively studied. Most scheduling heuristics target homogeneous multiprocessors, nevertheless there exists an important number of studies in heterogeneous multiprocessors. In this section we give an overview of different categories of schedulers for heterogeneous systems, we explain some details about schedulers targeting specific systems using compute accelerators and explain details of previous works on criticality-aware schedulers.

2.0.1 Schedulers for Heterogeneous Systems

There are previous works on schedulers for heterogeneous systems that form four different types of schedulers: listing, clustering, guided-random, and duplication-based schedulers.

Listing schedulers [1, 17, 23, 33, 44] have two scheduling stages. In the first stage, each task is given a priority based on the policy defined in each algorithm. In the second stage, tasks are assigned to processors depending on their priorities. Most criticality-aware schedulers fall in this category, and we discuss them in Section 2.0.3. The scheduler proposed in this paper is also a list scheduler.

Clustering schedulers [23, 24, 49, 50] first separate tasks into clusters, where each cluster is to be executed on the same processor. During the clustering stage, the algorithm assumes an unlimited number of available processors in the system. If the number of clusters exceeds the number of available cores, the *merging* stage joins multiple clusters so that they match the number of available processors. An example is the Levelized Min Time [24] clustering scheduler. This heuristic clusters tasks that can execute in parallel according to their *level* (i.e. sibling nodes in a graph have the same level), and assigns priorities to the tasks in a cluster according to its execution time, (i.e. tasks with the highest execution time have the highest priority). The task-to processor assignment is done in decreasing order of priority.

Guided-random schedulers [32, 36, 51] randomize their schedules by applying policies influenced by other sciences. Genetic algorithms [51] group tasks into generations and schedule them according to a randomized genetic technique. Chemical reaction algorithms [32] mimic molecular interactions to map tasks to processors. Some of these guided-random approaches are designed for heterogeneous systems [32, 51]. The scheduler by Page et al. [36] enables dynamic scheduling of multiple-sized tasks for heterogeneous systems. However, it does not support dependencies between tasks.

Duplication-based schedulers [2, 7, 52] aim to eliminate communication costs between processors by scheduling tasks and their successors on the same processor. If a task has many successors, it is duplicated and executed in multiple cores prior to its successors so all successor tasks get the data from their predecessors with the lowest communication cost. This scheduling potentially introduces redundant duplications of tasks which may lead to bad schedules. The Heterogeneous Economical Duplication scheduler [2] performs task duplication in an economical manner as it removes the redundant duplicates if they do not affect performance.

These previous works schedule tasks statically and assume the prior knowledge of the task execution times on the different processor types in the heterogeneous system.

2.0.2 Schedulers for Compute Accelerators

The schedulers in the previous section target the scheduling of generic TDGs on generic heterogeneous architectures. In this section we cover schedulers that target specific systems with compute accelerators. These works are more focused on the scheduling of tasks on the target platform based on the abstractions provided by the corresponding mixture of programming models for the general-purpose processors and the compute accelerators in the system.

Most heterogeneous systems with compute accelerators nowadays combine general-purpose CPUs and GPU compute accelerators. There is a set of programming models providing abstractions to ease the development of applications on these platforms. OmpSs [4, 18] offers this abstraction by allowing multiple implementations of a given task to be executed on different processing units [38]. The scheduler then assigns the execution of a task to the best resource according to its earliest finish time. Another case is StarPU [3], a library that offers runtime heterogeneity support and provides priority schedulers for task-to-processor allocation. AHP [37] is another framework that generates software pipelines for heterogeneous systems and schedules tasks to their earliest executor, based on profiling information gathered prior to runtime.

None of these works, however, take into account the criticality of tasks regarding task dependencies, but they rather focus on the earliest execution time of individual tasks on the processor types in the specific system configuration.

2.0.3 Criticality-Aware Schedulers

Several previous works propose scheduling heuristics that focus on the critical path in a TDG to reduce total execution time [17, 23, 33, 44]. To identify the tasks in the critical path, most of these works use the concept of *upward rank* and *downward rank*. The upward rank of a task is the maximum sum of computation and communication cost of the tasks in the dependency chains from that task to an exit node in the graph. The downward rank of a task is the maximum sum of computation and communication cost of the tasks in the dependency chain from an entry node in the graph up to that task. Each task has an upward rank and downward rank for each processor type in the heterogeneous system, as the computation and communication costs differ across processor types.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [44] maintains a list of tasks sorted in decreasing order of their upward rank. At each schedule step, HEFT assigns the task with the highest upward rank to the processor that finishes the execution of the task at the earliest possible time. Another work is the Longest Dynamic Critical Path (LDCP)

algorithm [17]. LDCP also statically schedules first the task with the highest upward rank on every schedule step. The difference between LDCP and HEFT is that LDCP updates the computation and communication costs on multiple processors of the scheduled task by the computation and communication cost in the processor to which it was assigned.

The Critical-Path-on-a-Processor (CPOP) algorithm [44] also maintains a list of tasks sorted in decreasing order as in HEFT, but in this case it is ordered according to the addition of their *upward rank* and *downward rank*. The tasks with the highest *upward rank* + *downward rank* belong to the critical path. On each step, these tasks are statically assigned to the processor that minimizes the critical-path execution time.

The main weaknesses of these works are that (a) they assume prior knowledge of the computation and communication costs of each individual task on each processor type, (b) they operate statically on the whole dependency graph, so they do not apply to dynamically scheduled applications in which only a partial representation of the dependency graph is available at a given point in time, and (c) most of them use randomly-generated synthetic dependency graphs that are not necessarily representative of the dependencies in real workloads.

Part II

Plan

Chapter 3

Research Plan

This section describes the goals for this thesis step by step and how the work is organised. The general subject of this thesis is the efficient exploitation of asymmetric systems.

3.1 Exploring the maturity of asymmetric systems

In this step of the thesis, we evaluate for the first time the suitability of currently available mobile asymmetric multi-core platforms for general purpose computing. This part of work has been implemented and is submitted to an international conference being currently under review. First, we demonstrate that out-of-the-box parallel applications do not 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.

When load-balancing techniques are not included in the original application, we evaluate alternative solutions that, without relying on the programmer, can leverage the opportunities that asymmetric systems offer. In particular, we evaluate a state of the art dynamic scheduler at the Operating System (OS) level that is aware of the characteristics of each core type [15]. This scheduler effectively exploits the system by running high CPU utilization processes on the big cores and low CPU utilization processes on the little cores.

An alternative to tackle the challenges of asymmetric multi-cores consists in transferring the responsibility of managing the parallel workload from the OS to the runtime system. Recent task-based programming models rely on an advanced runtime system to dynamically schedule tasks [5, 9, 18, 35, 47, 48, 53]. By allowing the programmer to specify the inter-task dependencies, these programming models rely on the runtime system for the dynamic scheduling of tasks to achieve load balancing.

More precisely, this part of work aims to evaluate three different scenarios of parallel execution when transferring the scheduling responsibility at different levels of the software stack:

- *Static Threading:* the scheduling responsibility is on the application level. The implementation of the application performs the scheduling.
- *OS Scheduling (GTS):* the operating system is responsible for performing the scheduling. Specifically we use the Global Task Scheduler (GTS) provided by ARM that is aware of the characteristics of the asymmetric system.
- *Task-based:* the runtime system is responsible for the efficient scheduling. Specifically we use applications written with the OmpSs programming model [18] [4] and the runtime is unaware of the platform.

3.2 Scheduling policies for asymmetric systems

We aim to improve the efficiency of the runtime system and make it aware of the underneath architecture. This is the motivation for the proposed scheduling techniques that take into account the asymmetry of the platform. We implement different scheduling policies in the runtime system of the OmpSs programming model as an approach to increase performance and energy efficiency. Even though task-based programming models is a powerful mechanism, the efficient mapping of ready tasks to different types of cores on an asymmetric system remains a challenge. Task-based parallel applications expose different characteristics that can affect the total application duration such as complex task dependency graphs (TDGs) with long critical paths or different levels of task cost variability. These characteristics influence researchers to develop smart scheduling techniques within a task-based programming model and accelerate the overall application. The criticality-aware schedulers detect the critical tasks of an application and increase performance by running critical tasks on fast cores. Unlike the previous works mentioned in Section 2.0.3, our schedulers, do not rely on profiling information and are evaluated on real scientific applications.

In this work, we propose three novel dynamic task schedulers that detect the critical tasks of the TDG. The purpose of these scheduling techniques is to separate the tasks into two groups: critical and non-critical. To to so, these schedulers maintain two ready queues: the critical queue and the non-critical queue. When a task becomes ready, the scheduler decides according to the policy whether it is critical or not. If the task is critical it is inserted in the critical ready queue while if the task is non-critical it is inserted in the non-critical ready queue. Each time a processor becomes idle, it retrieves a task from one of the ready

queues to execute. In our scheduling approaches, the fast cores of the system always check for ready tasks in the critical queue and the slow cores look for tasks in the non-critical queue. As a result, fast cores execute critical tasks and slow cores execute the non critical tasks. In the following sections we describe in more detail the different scheduling policies for asymmetric systems. A journal paper has been authored and is currently under peer review.

3.2.1 Criticality-Aware Task Scheduler

The first proposed scheduling algorithm generally applies to task-based programming models supporting task dependencies, but for simplicity we explain it in the context of the OmpSs programming model. This part of work has been already carried out and is published in the International Conference of Supercomputing (ICS) in June 2015 [14].

The Criticality-Aware Task Scheduler (CATS) [14] uses bottom-level longest-path priorities and consists of three steps:

- *Task prioritization*: when a task is created and added to the task graph, it is assigned a priority and the priority of the rest of tasks in the graph is updated accordingly.
- *Task submission*: when a task becomes *ready*, i.e., all its predecessors finished their execution, it is submitted to a *ready queue*. At this point, the algorithm decides whether the task is considered *critical* or *non critical*. The task is then inserted in the corresponding ready queue: tasks in the *critical ready queue* will be executed by fast cores, and tasks in the *non-critical ready queue* will be executed by slow cores.
- *Task-to-core assignment*: when a core becomes idle, it tries to retrieve a task from its corresponding ready queue to execute it. If the queue is empty, it might try to steal from the other queue depending on the work stealing policy. Currently, we support two work stealing mechanisms: *simple* work stealing, i.e., fast cores can steal from slow cores; and *bi-directional* (*2DS*) work stealing, i.e., both types can steal from the other. The default policy is *simple*.

These steps are performed dynamically and potentially in parallel in different cores. This means that while some tasks are being prioritized, previously created tasks may be submitted, and others assigned to available cores or executed.

To give an overview of the scheduling process, Figure 3.1 shows a scheme of the operation of CATS. In the TDG on the left, each node represents a task and each edge of the graph represents a dependency between two tasks. The number inside each node is the *bottom level* of the task: the length of the longest path in the dependency chains from this

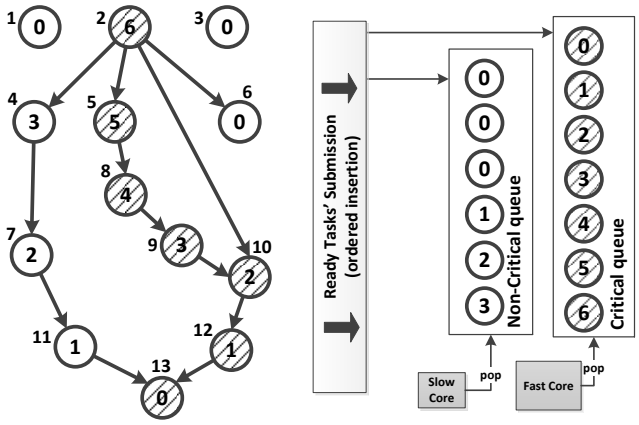


Fig. 3.1 Task submission with CATS. Nodes are marked with the *bottom level* of each task. Pattern-filled nodes mark the critical tasks.

node to a leaf node. The priority of a task is given by its bottom level. The pattern-filled nodes indicate tasks that are considered critical. The number outside each node is the task id and is used in the text to refer to each task. Critical tasks are inserted in the critical queue, and non-critical tasks to the non-critical queue. The insertion is ordered with the highest priorities at the head of the queue and the lowest priorities at the tail. Slow cores retrieve tasks from the head of the non-critical queue and fast cores from the critical queue. The following sections describe these scheduling steps in more detail.

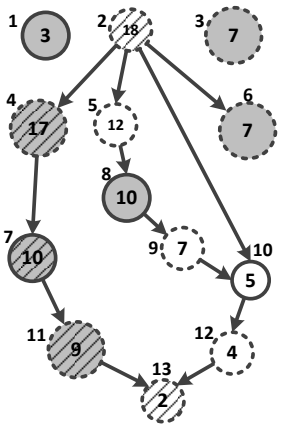
3.2.2 Critical Path Scheduler

The Critical Path scheduler (CPATH) dynamically detects the critical path of the TDG. Like CATS, CPATH separates tasks into two groups: critical and non-critical tasks. The detected critical tasks are executed by the fast cores in the system and non-critical tasks are executed by slow cores. The difference with CATS is the algorithm for critical path detection. CPATH takes into account the task execution time, about which CATS is unaware. To do so, CPATH implements a more complex and accurate critical path detection algorithm that takes into account task execution time.

CPATH scheduler consists of three steps:

- *Task prioritization*: this step takes place when a task is finishing its execution (task completion). This is different than CATS since at the end of a task execution the algorithm may record the task execution time (task cost). According to the discovered task cost CPATH assigns priorities to tasks by traversing the TDG from top to bottom.

- *Task submission*: when a task becomes *ready*, it is submitted to a *ready queue*. At



	TaskA	TaskB
Input1	1	3
Input2	2	7

Task Cost	Task IDs
1	7, 10
2	2, 5, 9, 12, 13
3	1, 8
7	3, 4, 6, 11

Fig. 3.2 Priority assignment taking into account the task costs. Task costs are assumed known and are shown in the tables.

this point, CPATH decides whether or not the task is *critical* and inserts it in the corresponding ready queue. This step has only slight implementation differences with CATS.

- *Task-to-core assignment*: this step is identical to CATS and supports the same work stealing mechanisms.

Figure 3.2 is used to describe the priority assignment with CPATH. The specific TDG contains tasks of two different types and two different input sizes. Node color shows the different task types and the outline of the circle (dashed or solid) shows the different input sizes. The upper table in Figure 3.2 indicates the execution time of the tasks according to their type and input size. The algorithm assumes that task instances of the same type with the same input size have the same (or very similar) execution time. To track this information, CPATH discovers the cost of every possible task type-input size duple (tt-is duple) that appears on the TDG. The numbers inside the nodes show the bottom cost-based priorities that CPATH assigns. We define the *bottom cost* of a node on a directed acyclic graph as the maximum estimated time in the dependency chains from this node to a leaf node.The numbers outside the nodes show their task ID.

3.2.3 Hybrid Criticality Scheduler

The Hybrid Criticality Scheduler (HYBRID) is a combination of the CATS and CPATH scheduling policies. HYBRID keeps the simplicity of the implementation of CATS and introduces the task execution time only if available. This results in an efficient low-overhead scheduler that computes the critical path of a TDG more faithfully than CATS and with lower overheads than CPATH. This section describes HYBRID through its relation to CATS

and CPATH described in Sections 3.2.1 and 3.2.2. We focus our description on the task prioritization, since task submission and task-to-core assignment for HYBRID are identical to CPATH.

As shown, CPATH computes priorities on task completion. The algorithm for priority computation is an expensive operation and is in the critical path of the execution: on task completion the core becomes available but the start of the next task is delayed by priority computation. Also, when multiple cores are completing tasks, there will be contention on accessing the TDG for priority computation. On the other hand, CATS computes priorities during task creation. The computation of priorities during task creation is more efficient because, unless there is nested parallelism, one core creates all tasks and therefore there is no contention on priority computation. The downside is that there is potentially less information available on task execution time on task creation, as some task type may have not been executed yet at the time all tasks are created.

When comparing CPATH and HYBRID schedulers their logical operation is similar. However the difference in their implementation may result in different task priorities potentially leading to different schedules. For applications with small TDGs, HYBRID may not be able to compute an accurate critical path because task creation does not overlap with a sufficient amount of task exits. Therefore, task execution information will not be available during priority computation and HYBRID will prioritize based on bottom-level priorities (like CATS). If the application has a large TDG and task creation overlaps with a sufficient amount of task exits, HYBRID will use bottom-cost priorities.

3.3 Runtime thread migration mechanisms

The asymmetry-aware scheduling policies may increase the runtime overheads. Especially when the runtime operations are executed by the fast cores of the system, preventing them from executing user tasks, the runtime activity can create a bottleneck at the task execution. This is the motivation for reserving one core responsible for the execution of the runtime activities. This way the rest of the cores will be devoted for the uninterrupted task execution. The following subsections describe our motivation and background of this work as well as the current software modifications.

3.3.1 Motivation

Task creation is a bottleneck for most applications especially when using smart scheduling policies like the ones we described. Figure **??** shows the runtime activity of the master

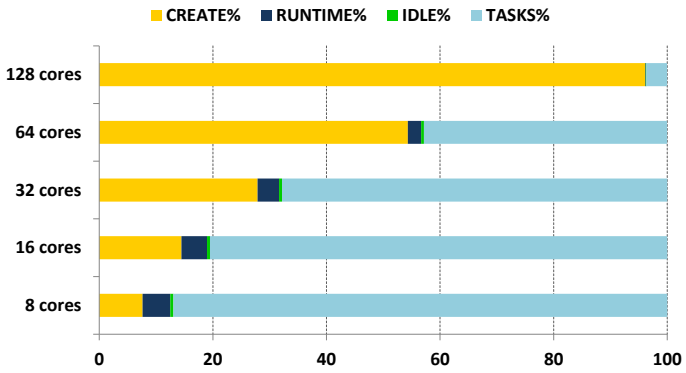


Fig. 3.3 Activity breakdown of the master thread when executing a parallel application.

thread during the execution of the Cholesky benchmark on 8, 16, 32, 64 and 128 cores. Each one of the series represents a different runtime overhead from the ones described above. CREATE represents the *Creation* step, RUNTIME refers to the *Finish* step, IDLE shows the master thread's idle time and the TASKS is the time spent on task execution. The percentage of time spent on task creation is increasing as we increase the number of cores. This is because the creation overhead is static so the more we reduce the application's execution time by adding resources the more important this step becomes in terms of execution time. On the other hand, the task execution time percentage is decreased as we increase the number of cores because the computational activity is being shared among more resources. The RUNTIME decreases as we increase the number of cores because this activity is also shared among the resources.

Our motivation for this work is the bottleneck introduced by task creation (CREATION) as shown in Figure **??**. We implement a runtime proposal that decouples this piece of the runtime and accelerates it on a specialized hardware resulting in higher performance.

3.3.2 Background

Once again, OmpSs programming model offers the most appropriate environment for exploring this heuristic. Like OmpSs, task-based parallel programming models[35],[18], [4], are widely used to facilitate the programming of parallel codes for multi-core systems. These programming models offer annotations that the programmer can add to the application's sequential code. By adding these annotations, the programmer decomposes the application into *tasks* and specifies the input and output data dependencies between them. A compiler is responsible to translate the annotations into code by adding calls to the programming model's runtime system. The runtime system consists of software threads and is responsible for the efficient execution of the tasks with respect to the data dependencies as

```
    ...
//task_clause
memalloc(&task, args, size);
createTask(deps, task, parent, taskData);
    ...
```

Listing 3.1 Compiler generated pseudo-code equivalence for task annotation.

well as the availability of resources.

When the compiler encounters one task annotation in the code, it transforms it to the pseudo-code shown in Listing 3.1. `Memalloc` is performing the memory allocation for the task and its arguments. Next is a runtime call, which is the createTask, responsible for the linking of the task with the runtime system. At this point a task is considered *created* and below are the three possible states of a task inside the runtime system:

- *Created:* A task is initialized with the appropriate data and function pointers and it is inserted in the Task Dependency Graph (TDG). The insertion of a task in the TDG implies that the data dependencies of the tasks have been identified and the appropriate data structures have been created and properly initialized.
- *Ready:* When all the data dependencies of a created task have been satisfied, the task is ready and it is inserted in the *ready queue* where it waits for execution.
- *Finished:* When a task has finished execution and has not been deleted yet.

The runtime system creates and manages the software threads for the execution of the tasks. Typically one software thread is being bound to each core. One of the threads is the *master thread*, and the rest are the *worker threads*. The master thread starts executing the code of Listing 3.1 sequentially. The allocation of the task takes place first. What follows is the task creation, that includes the analysis of the dependencies of the created task and the connection to the rest of the existing dependencies. Then, if there are no task dependencies, which means that the task is ready, the task is also inserted in the ready queue and waits for execution.

Listing 3.2 shows the pseudo-code for the task creation step within the runtime. The `createTask` function is first initializing the task by copying the corresponding data to the allocated memory as well as connecting the task to its parent task (`initAndSetupTask`). After this step, the task is ready to be inserted in the TDG; this is done by the `insertToTDG` function. This function takes as arguments a list with all the memory addresses that are to be written or read by the task (`dList`), and the task itself. If for a task the `dList` is empty, this means that there are no memory addresses that need to be tracked during the execution;

thus, the task is marked as *ready* by inserting it in the *ready queue* (`rQueue_submission`). Each entry of `dList` contains the actual memory address as well as the access type (read, write or read-write). The runtime keeps a unified dependency tracking structure (`depMap`) where it stores all the tracked memory addresses together with their writer and reader tasks. For each item in the `dList` the runtime searches for an existing representation inside the `depMap`. If the memory address of an entry of the `dList` is not represented in the `depMap`, it is being added; if the address of a `dList` item belongs to the `depMap`, this means that a prior task has already referred to this memory location. Thus at this point there is a data dependency. Then according to the type of the access type of `d`, the readers and the writers of the specific address are updated in the `depMap`.

To reduce the lookup into the `depMap` calls, every time a memory address is modified, the tasks keep track of their *successors* as well as the number of *predecessors*. The *successors* of a task are all the tasks that their input depends on the output of the current task. The *predecessors* of a task are the tasks whose output is used as input for the current task. When a `read` access is identified, the task that is being created is added to the list of successors of the last writer task (lineXX).

As tasks are executed, the dependencies between them and their successors are satisfied. So the successor tasks that are waiting for input, eventually become *ready* and are inserted to the ready queue. When a task becomes *finished*, the runtime has to perform some actions in order to prepare the successor tasks for execution. These actions are described in Listing 3.3. The runtime first updates the `depMap` to remove the possible references of the task as reader or writer. Then, if the task does not have any successors, it can safely be deleted. If the task has successors, the runtime traverses the successor list and for each successor task it decreases its predecessor counter. If for a successor task its predecessor counter reaches zero, then this task becomes *ready* and it is inserted in the ready queue (`rQueue_submission(succ)`).

To summarize, the runtime activity mainly takes place at the task state changes. One state change corresponds to the task creation, so a task from being just allocated it becomes created, and the second change occurs when a task from being ready becomes finished. In these two runtime phases the runtime system interferes and introduces runtime overheads.

3.3.3 Runtime Activity Manager

RAM assumes the existence of a specialized hardware that accelerates the task creation step. RAM relieves the master and worker threads from this intensive runtime activity by offloading it on the special purpose hardware. In our design, apart from the master and the worker threads, we introduce the Special Runtime Thread (SRT). When the runtime system

```
void createTask(Deplist dList, Task task1,
            Task parent, Data taskData) {
  initAndSetupTask(task1, parent, taskData);
  insertToTDG(dList, task1);
}

Dependency depMap[];

void insertToTDG(DepList dList, Task task1) {
 if( dList.empty() ) {
   rQueue_submission(task1);
   return;
 }
 Dependency entry;
 for( d in dList ) {
   entry = depMap.lookupAddress(d.address());
   if(entry == NULL)
     depMap.add(d.address(), d.accessType(), task1);
   if(d.accessType() == "write") {
     entry.addLastWriter(task1);
   }
   else if(d.accessType() == "read") {
     entry.addReader(task1);
     entry.lastWriter()->addSuccessor(task1);
   }
   else if(d.accessType() == "read-write") {
     entry.addLastWriter(task1);
     entry.addReader(task1);
   }

 }
}
```

Listing 3.2 Pseudo-code for task creation.

```
void task_finish(Task *t) {
  depMap.deleteAsWriter(t);
  depMap.deleteAsReader(t);
  if(t->successors.empty()) delete task;
  else {
    for( succ in t->successors ) {
      succ.decreasePredecessors();
      if(succ.numPredecessors == 0)
        rQueue_submission(succ);
    }
  }
```

Listing 3.3 Pseudo-code for task_finish runtime activity.

starts, it creates the SRT and binds it to the task creation accelerator, keeping its thread id in order to manage the usage of it. During runtime, the master and worker threads look for ready tasks in the task ready queue and execute them along with the runtime. SRT, instead of querying the ready queue for tasks, it looks for runtime activity requests in the runtime ready queue (RRQ) and if there are requests, it executes them.

Figure 3.4 shows the communication infrastructure between threads within our runtime. Our system maintains two queues; the ready task queue (`TASKQ`) and the runtime requests queue (`RRQ`). The TASKQ is used to keep the tasks that are ready for execution. The RRQ is used to keep the runtime activity requests. The master and the worker threads can push and pop tasks to and from the TASKQ and they can also add runtime activity to the RRQ. The special runtime thread (SRT) pops runtime requests from the RRQ and under circumstances, it also pops ready tasks from the TASKQ.

When the master thread encounters a task clause in the application's code, after allocating the memory needed, it calls the `createTask` as shown in Listing **??** and as described in Section **??**. RAM decouples the execution of `createTask` from the master thread. To do so, when a thread encounters a call to the `createTask`, the runtime system checks if the SRT is enabled; if so, instead of performing the task creation itself, it generates a *CREATE* request and inserts it in the RRQ so that the SRT can read and execute it. The running thread then continues by executing tasks. The *CREATE* runtime request includes the appropriate info to execute the code described in Listing 3.2. That is, the dependence analysis data, the address of the allocated task, its parent and the taskData.

At the same time that the master and worker threads are executing tasks, the SRT is looking for *CREATE* requests in the RRQ to execute. Listing 3.4 shows the code that the SRT is executing until the end of the parallel execution. The special runtime thread continuously checks whether there are requests in the RRQ (line 3). As long as there is a pending

```
1 void SRTloop() {
   int maxTasks = runtime.numWorkers * MAX;
2  while( true ) {
3    while( not RRQ.empty() ) {
4      executeRequest( RRQ.pop() );
5    if( RRQ.empty() and readyTasks > maxTasks )
7      executeTask( readyQ.pop() );
8  }
9  if( runtime.SRTstop() ) break;
10 return;
11}
```

Listing 3.4 Pseudo-code for the SRT loop.

task creation, the SRT executes the task submission and inserts the task in the TDG with a call to the `executeRequest` (line 4). If at some point the RRQ becomes empty, the SRT checks the number of ready tasks in the ready queue (lines 5, 6) and if the number of ready tasks is greater than `maxTasks`, which means that the workers are very loaded, it executes the next ready tasks in the queue.

3.4 Asymmetry-aware runtime system

Having completed the previous steps we plan to combine our implementations of scheduling techniques and runtime thread migration mechanisms and end up with a novel runtime system for asymmetric architectures that offers two levels of adaptability: first is the choice of the appropriate core type for the runtime execution and second is the appropriate scheduling of the tasks on the available cores.

The first step in discovering the appropriate scheduling-migration mechanism combination is to perform experiments of scientific applications using all the possible combinations of migration and scheduling policies as well as use different machine set-ups in terms of numbers of fast and slow cores. The analysis of these results will show us whether the combining existing implementations is efficient and under what circumstances. After concluding to the most appropriate mechanism, we will further verify the results on a larger scale, using an HPC machine with a larger number of cores.

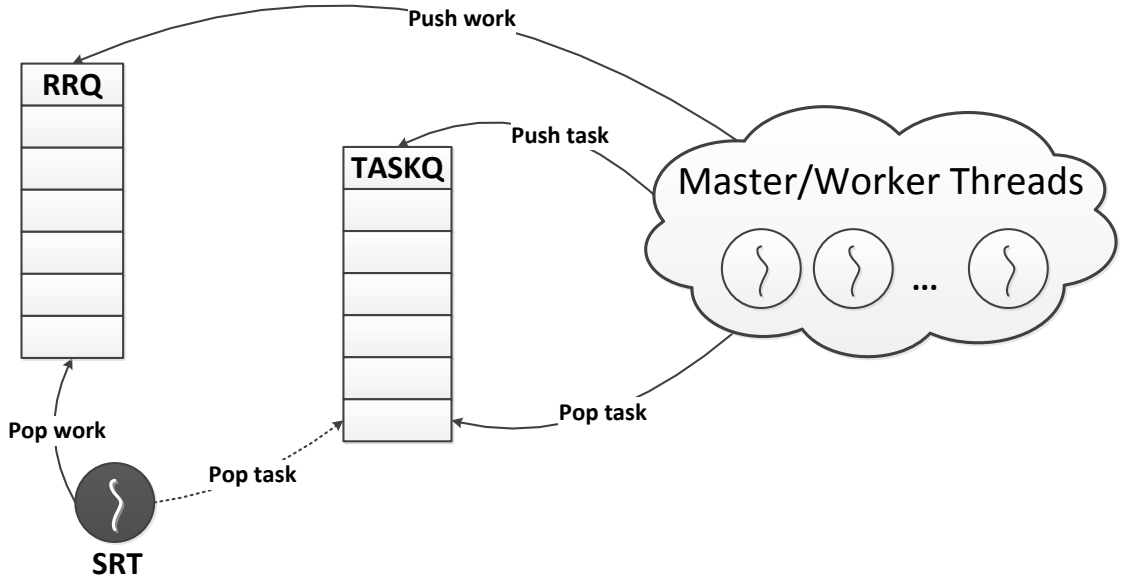


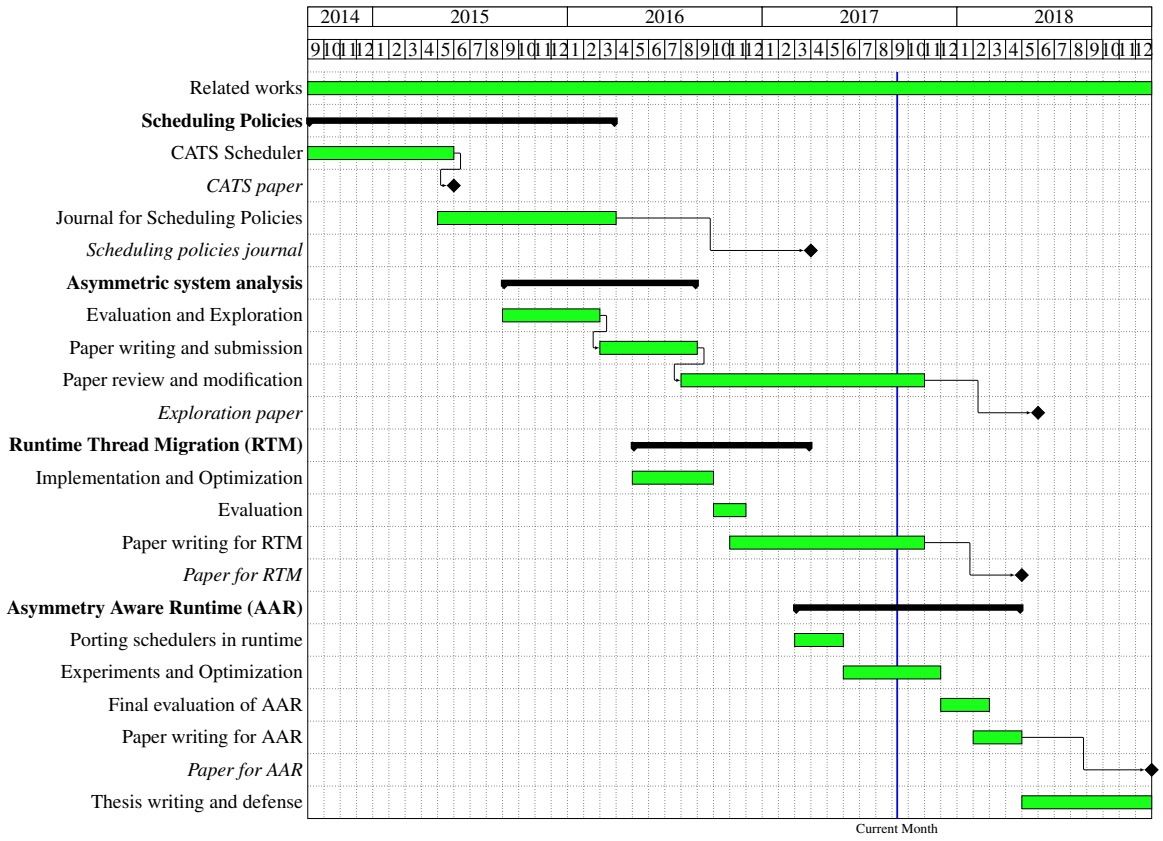
Fig. 3.4 Communication mechanism between master/workers and SRT threads.

3.5 Thesis Roadmap

The Gantt graph below describes the whole plan with the various stages. Each stage corresponds to a section of this chapter. The literature review process (named Related works in the graph) is a constant activity that lasts the entire course of the PhD.

A part of work for this PhD was conducted before the PhD enrolment. This work is described in the graph thus the graph starts from September of 2014. However, as is noted on the graph, the official PhD enrolment is on September of 2015.

The Gantt graph below has been updated with the current status of the publications. During this year, we successfully published the Journal of Scheduling policies. Furthermore we modified improved and submitted the paper "Asymmetric System Analysis" to multiple conferences but none of them has accepted our work so far. We plan to perform further modifications in the text and resubmit the paper this October. In addition, we are in progress of writing the paper for the Runtime Thread Migration and submit it in October as well.


2014
2015
2016
2017
2018
Related works
Scheduling Policies
CATS Scheduler
CATS paper
Journal for Scheduling Policies
Scheduling policies journal
Asymmetric system analysis
Evaluation and Exploration
Paper writing and submission
Paper review and modification
Exploration paper
Runtime Thread Migration (RTM)
Implementation and Optimization
Evaluation
Paper writing for RTM
Paper for RTM
Asymmetry Aware Runtime (AAR)
Porting schedulers in runtime
Experiments and Optimization
Final evaluation of AAR
Paper writing for AAR
Paper for AAR
Thesis writing and defense
Current Month

Chapter 4

Methodology

4.1 The OmpSs programming model

We choose to implement the new mechanisms proposed in this document in the OmpSs programming model. The OmpSs programming model is a task-based programming model that offers a high level abstraction to the implementation of parallel applications for various homogeneous and heterogeneous architectures [4, 18]. It enables the annotation of code blocks or function declarations with the task directive, which declares a task. Every invocation of such a function creates a task that is executed concurrently with other tasks or parallel loops. OmpSs also supports task dependencies and dependency tracking mechanisms [20]. OmpSs is composed of two main elements: the Mercurium compiler, responsible for the translation of the OmpSs annotation clauses to source code that calls to the Nanos++ API, and the Nanos++ runtime system, responsible for the internal creation and execution of the tasks.

Nanos++ is an environment that serves as the runtime platform of OmpSs. It provides device support for heterogeneity and includes different plug-ins for implementations of schedulers, throttling policies, barriers, dependency tracking mechanisms, work-sharing and instrumentation. This design allows to maintain the runtime features by adding or removing plug-ins, facilitating the implementation of a new scheduler, or the support of a new architecture.

The implementations of the different scheduling policies in Nanos++ perform various actions on the states of the tasks. A task is *created* if a call to this task is invoked but it is waiting until all its inputs are produced by previous tasks. When all the input dependencies are satisfied, the task becomes *ready*. The ready tasks of the application at a given point in time are inserted in the *ready queues* as stated by the scheduling policy. Ready queues can be thread-private or shared among threads. When a thread becomes idle, the scheduling

policy picks a task from the ready queues for that thread to execute. The default OmpSs scheduler employs a *breadth-first* policy (BF) [19] and implements a single first-in-first-out ready queue shared among all threads. When a task is ready, it is inserted in the tail of the ready queue and when a core becomes available, it retrieves a task from the head of the queue. BF does not differentiate among core types and assigns tasks in a first-come-first-served basis. We use this scheduler as our baseline.

The Nanos++ internal data structures support task prioritization. The task priority is an integer field inside the task descriptor that rates the importance of the task. If the scheduling policy supports priorities, the ready queues are implemented as *priority queues*. In a priority queue, tasks are sorted in a decreasing order of their priority. The insertion in a priority queue is always ordered and the removal of a task is always from the head of the queue, i.e., the task with the highest priority. The priority of a task can be either set in user code, by using the *priority* clause, which accepts an integer priority value or expression, or dynamically by the scheduling policy, as is described in the next section.

Nanos++ runtime is the baseline of our runtime implementations. We choose to implement our proposed scheduling policies within this runtime. Also Nanos++ is easy to modify in order to reserve one thread responsible for the runtime.

4.2 Evaluation

4.2.1 Evaluation platform

The Hardkernel ODROID-XU3 development board has an 8-core Samsung Exynos 5422 chip with an ARM big.LITTLE architecture and 2GB of LPDDR3 RAM at 933MHz. The chip has four Cortex-A15 cores at 2.0GHz and four Cortex-A7 cores at 1.4GHz. The four Cortex-A15 cores form a *cluster* with a shared 2MB L2 cache, and the Cortex-A7 share a 512KB L2 cache. The two *clusters* are coherent, so a single shared memory application can run on both clusters, using up to eight cores simultaneously. In our experiments, we evaluate a set of possible combinations of fast and slow cores varying the total number of cores from two to eight. For the reminder of the paper, we refer to Cortex-A15 cores as *big* and to Cortex-A7 cores as *little*.

4.2.2 Simulator

To evaluate our approaches on larger multi-core systems we use the heterogeneous multi-core TaskSim simulator [40]. TaskSim allows the specification of a heterogeneous system with two different types of cores: fast and slow. We can configure the amount of cores of

Table 4.1 Evaluated benchmarks from the PARSEC benchmark suite

Benchmark	Description	Input	Parallelization	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	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	4.16
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design.	2.5 million elements, 6,000 temperature steps	unstructured	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	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	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	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	2.64
streamcluster	Solves the online clustering problem.	200,000 points per block, 5 block	data-parallel	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	2.78

each type and the difference in performance between the different types (performance ratio) in the TaskSim configuration file. In our experiments, we will use up to a total of 80 distinct heterogeneous machine configurations. These comprise systems with the total number of cores ranging from 16 to 128, and the number of fast cores ranging from 1 to 16. For all these configurations, we evaluate the following performance ratios between fast and slow cores: $2\times$, $2.5\times$, $3\times$, $3.5\times$ and $4\times$.

4.2.3 Applications

In our experiments we use the PARSECSs benchmark suite [12]. This benchmark suite offers a set of scientific real world applications implemented in OmpSs as well as pthreads. Table 4.1 shows the applications and their characteristics.

Apart from the PARSECSs benchmark suite we use four scientific kernels implemented in the OmpSs programming model: Cholesky factorization, QR factorization, Heat diffusion and Integral Histogram. These benchmarks are accessible in the BSC Application Repository [8].

Cholesky factorization is a dense matrix operation that is used for solving linear equations in linear least square systems. The OmpSs implementation of Cholesky blocks the input matrix into square blocks of floats and each task is responsible for performing the factorization on one block.

QR Factorization is a linear algebra algorithm that is used to solve the linear least

squares problem [11]. We evaluate the performance of a blocked, communication avoiding QR implementation in OmpSs. We use an input blocked matrix of 8192×8192 doubles forming 16×16 blocks.

Heat diffusion uses the Gauss-Seidel method to compute the heat distribution on a matrix from x heat sources. Heat diffusion implements an iterative solver of the equation that invokes the Gauss-Seidel method until the desired convergence is reached. We use a matrix of 8192×8192 doubles and block size of 512×512.

Integral histogram is a method to compute a cumulative histogram for each pixel of an image. The OmpSs implementation performs a horizontal and a vertical scan that transmit histograms to the blocks that reside on the right or below the current block. Due to these transmissions, the application introduces many task dependencies. We use as input an image of 4096×4096 pixels and block size of 512×512.

Blackscholes from the PARSEC Benchmark Suite [10], is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). The OmpSs implementation [40], divides the work into units of a predefined block size. This block size allows having much more task instances than threads, which implies a much better load balance, as this is an embarrassingly parallel application with no dependencies among tasks in the same run.

4.2.4 Metrics

All the experiments in this paper are performed on the Hardkernel Odroid XU3 described in Section 4.2.1. In our experiments, we make use of the `cpufreq` driver to set the big cores to run at 1.6GHz and the little cores at 800MHz.

Performance

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`. 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 serial execution time on one little core. Equation 4.1 shows the formula to compute the speedup.

$$\text{Speedup(L, B, method)} = \frac{\text{Exec. time(1, 0, serial)}}{\text{Exec. time(L, B, method)}} \tag{4.1}$$

Power and Energy

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 gather the power and energy measurements, a background daemon is reading the machine power sensors periodically during the application execution with negligible 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 SoC was consuming during five executions of each configuration, 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 4.2 and 4.3 show the formulas for these calculations.

$$\text{Normalized Energy(L, B, method)} = \frac{\text{Energy(L, B, method)}}{\text{Energy(4, 0, static-threading)}} \tag{4.2}$$

$$\text{Normalized EDP(L, B, method)} = \frac{\text{EDP(L, B, method)}}{\text{EDP(4, 0, static-threading)}} \tag{4.3}$$

4.2.5 Results

We have already made some progress in this thesis and have already covered some of the work to be done. From our evaluation of the asymmetric system described in Section 3.1 we found that adding little cores to a symmetric multi-core with big cores presents significant challenges for the application, OS and runtime developers. Little cores increase load imbalance and can degrade performance as a result. Relying on the application developer to deal with this asymmetry is complex and many applications are not ready. A dynamic OS scheduler such as *GTS* can help in mitigating these problems, but the best results in terms of performance are obtained with the *Task-based* approach. In terms of power and energy, the asymmetric multi-core provides significant benefits, although the symmetric multi-core with little cores remains the most energy-efficient configuration. The *Task based* approach offers the highest performance with the lowest energy consumption when used on our 8-core system. With this in mind, the system-aware scheduling policies of a task-based programming model are meant to improve scheduling at the most appropriate level of the software stack.

Evaluation of Scheduling Policies

This section shows the results obtained from our scheduling policies described in Section 3.2. We compare our approaches against a dynamic implementation of the Heterogeneous Earliest Finish Time scheduler [44] (dHEFT) in the OmpSs programming model. The implementation assumes two different types of cores (big and little) and keeps records of the tasks' execution times in each core. The original HEFT [44] implementation assumes the prior knowledge of the TDG as well as the costs of the tasks. In our case, since the evaluation consists of running real applications, the best way to compare HEFT to our proposal is to keep the scheduling idea of HEFT and transform it from a static to a dynamic scheduler. This means that dHEFT discovers the costs of the tasks at runtime, computes a mean value of the costs for each task-type and parameter-size duple for each type of core, and then finds the core that will finish the task at the earliest possible time.

To find the earliest possible executor, dHEFT maintains one list per core (wlist) including the ready tasks waiting to be executed by that core. When another task becomes ready, dHEFT first checks if there are records of prior execution of this task. If the number of records is sufficient (in our experiments we require a minimum of three records) then the estimated execution time of the task is considered stable. Then, using that estimated execution time, the task is scheduled to the earliest executor by consulting the wlist of all the cores. If the number of records is not sufficient for one of the core types, then the task is scheduled to the earliest executor of this core type to get another record of that task-type and core-type execution time. In all cases, dHEFT updates the history of records on every task execution to adapt for phase changes in the application.

We evaluate the applications Cholesky, QR, Heat diffusion, Integral Histogram and Bodytrack described in Section 4.2.3. To more precisely characterise the benchmarks, we plot the task cost variability for each benchmark on Figure 4.1. For each of these plots, the *x* axis shows the normalized task cost and the *y* axis the number of tasks that correspond to this task cost (e.g. how many tasks have this cost). This is used to show how heterogeneous each application is and explain the behaviour of the heterogeneous schedulers that take into account the execution time.

Figure 4.2 shows the speedup obtained for each application, scheduler and machine setup. The x axis shows for each application the total number of cores at the bottom and the number of big cores at the top. We classify the benchmarks according to their task cost variability to easier explain the results.

Heat diffusion is the kernel with the lowest task variability (e.g. the most homogeneous benchmark) as shown in Figure 4.1d. CATS, HYBRID and dHEFT increase the performance of heat by 10% on 8 cores and obtain similar results for the other numbers of cores

by rearranging the tasks according to the type of the resources. Due to its high per-task overheads and the homogeneity of the benchmark, CPATH scheduler cannot outperform the random BF scheduler. Moreover, for this benchmark, CPATH detects only 23% of the tasks to be critical while CATS and HYBRID detect approximately 54%, when running on 8 cores. This happens because with CPATH, it is more likely to have zero-priority tasks during the task submission step, due to the post-exit task priority assignment that the algorithm introduces. The zero-priority tasks are considered non-critical and this limits the utilization of the big cores with CPATH.

Cholesky 16×16 has also low task cost variability. The improvements of CATS, dHEFT and HYBRID over BF are limited to around 7% when running on 8 cores. These schedulers perform almost the same for the rest numbers of cores and CPATH performs almost the same as BF. The increased overheads of CPATH do not pay off with better schedules since, for the same reason as in the case of Heat diffusion, only 10% of the tasks are marked as critical on 8 cores (while 21% CATS and 16% HYBRID).

Bodytrack shows low task cost variability, since 99% of its tasks have similar execution times. In this case, contrarily to the previous benchmarks CPATH manages to achieve similar speedups to CATS and HYBRID and outperform BF by up to 15%. This is due to the very high number of tasks of bodytrack; CPATH overcomes its overheads by using the detected task execution times for a higher number of tasks. In other words, the learning phase of CPATH becomes a smaller proportion of the total execution of the benchmark. Since bodytrack has so many tasks, the per-task overhead of CPATH is around 120us while for CATS it is 93us. On the other hand, dHEFT shows poor performance because of the overheads of analyzing a TDG with a high number of tasks to compute the earliest finish time schedule.

Integral histogram is characterized by medium task cost variability and high amount of tasks. This benchmark is dependency intensive with limited parallelism, which makes scheduling decisions very important. CATS and HYBRID schedulers achieve the best results since they focus more on the TDG structure and dependencies, improving BF by 30% and 27% respectively. CPATH and dHEFT are slightly less efficient and improve BF by 19 and 21% respectively.

For Cholesky 8×8, the heterogeneous schedulers CATS, HYBRID and dHEFT constantly improve the performance of BF and reach up to 45% improvement on 8 cores. It is observed here that dHEFT indeed performs better when the number of tasks is limited as this workload has 120 tasks in total. The additional overheads of CPATH do not compensate with increased performance in this case because there are not enough tasks to apply the better scheduling.

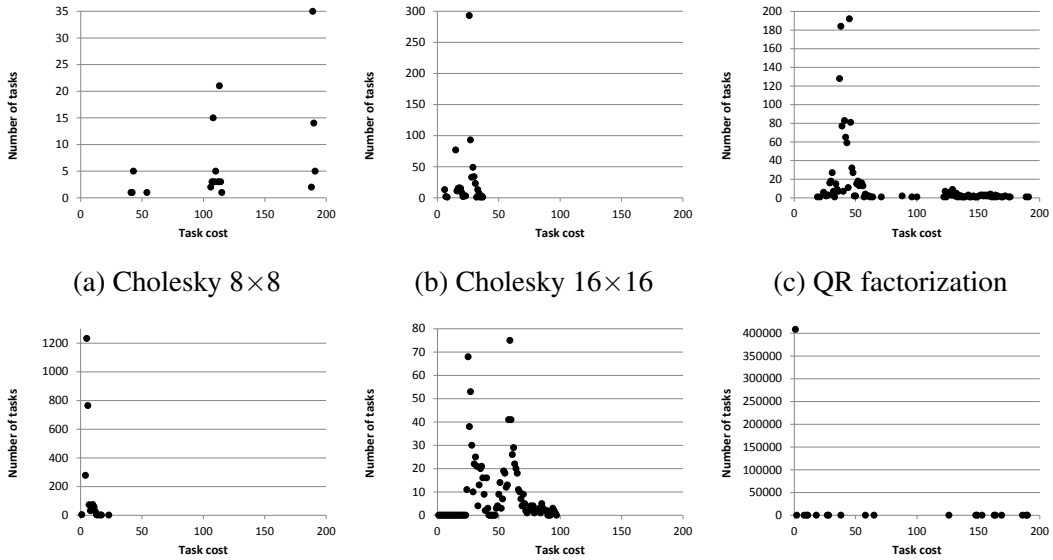
QR factorization is the highest task cost variability benchmark as shown in Figure 4.1c. This is the reason why HYBRID gradually outperforms CATS as we increase the number of cores. HYBRID manages to efficiently detect critical tasks that reside on the critical path and boost their execution reaching 17% improvement over the baseline. For this benchmark, CPATH also reaches a 13% improvement over BF since task cost matters in this case. However, CPATH speedup is still limited compared to HYBRID because of the higher scheduling overheads which in this case is 1.8× higher than CATS overheads. dHEFT also improves BF by finding the earliest executor of each task, but the improvement is limited to 11% which is lower than the other approaches.

This section showed a straight comparison between different heterogeneous schedulers. It is important to note that schedulers like CPATH and HYBRID, that detect the time-based critical path, are the best choices when the application has a large amount of tasks. This is because the additional overheads of these schedulers for critical path computation take place only when there are new tasks on the TDG or when there is a task exit of an untracked task type. When the TDG has been completely created, and as soon as the cost of every task type of the application has been tracked, the schedules of these approaches are purely beneficial. On the other hand, schedulers like dHEFT perform the same steps for every single task that becomes ready, affecting the entire execution since the exit of a task triggers the execution of its successors that become ready. Thus, as the number of tasks is increased, the additional scheduling overheads are increased when using dHEFT-like approaches. CATS scheduler is an efficient scheduling solution for any number of tasks and task cost distributions. The additional CATS overheads take place only during task creation and are smaller than CPATH overheads with the drawback of not considering the task execution time. If we have to choose the best and most generic heterogeneous scheduling approach among the presented schedulers the HYBRID scheduler is the best choice, since it computes an accurate critical path only

Evaluation of Runtime Activity Manager

In our preliminary evaluation of three workloads using RAM we observe very promising results. Specifically, Figure 4.3 shows the speedup of the baseline and RAM approach when we simulate the applications on up to 512 cores. We can see that for the baseline runtime, performance saturates as we increase the number of cores. This happens due to the task creation overhead, which we overcome by accelerating it on the special hardware.

We plan to further improve this evaluation by adding results for asymmetric systems.



(a) Cholesky 8×8 (b) Cholesky 16×16 (c) QR factorization

(d) Heat diffusion (e) Integral Histogram (f) Bodytrack

Fig. 4.1 Task cost distribution for each application. Results are based on 4BIG-core executions. *x* axis shows the cost of the tasks and *y* axis shows the number of tasks with the corresponding task cost.

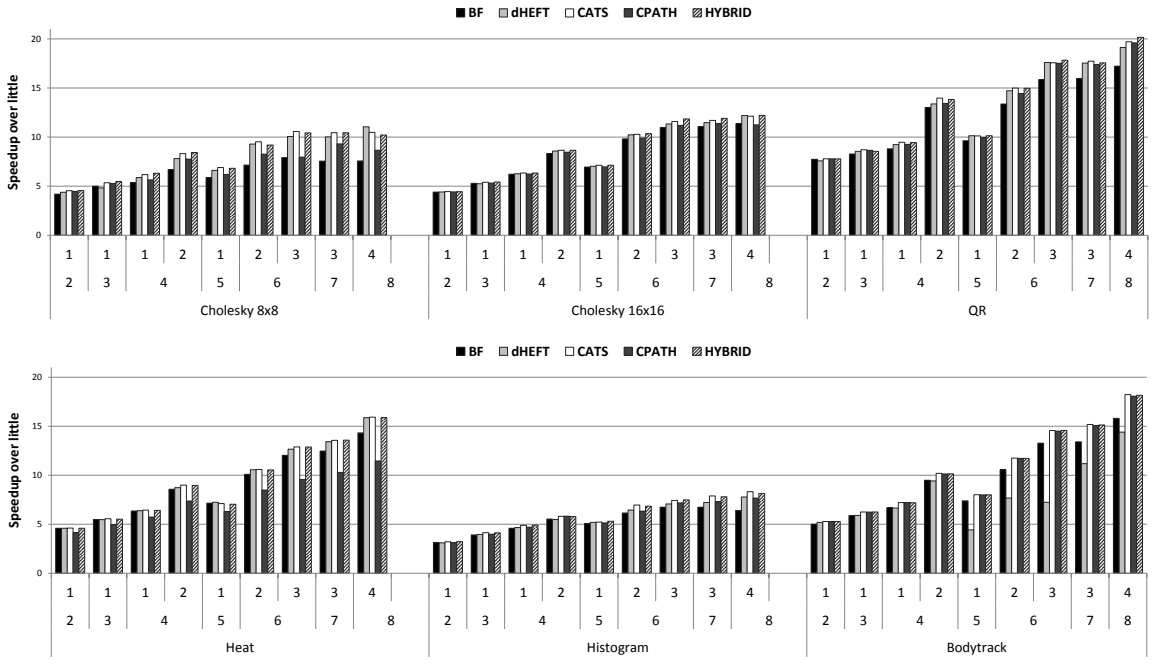


Fig. 4.2 Speedups obtained for each scheduler and each application

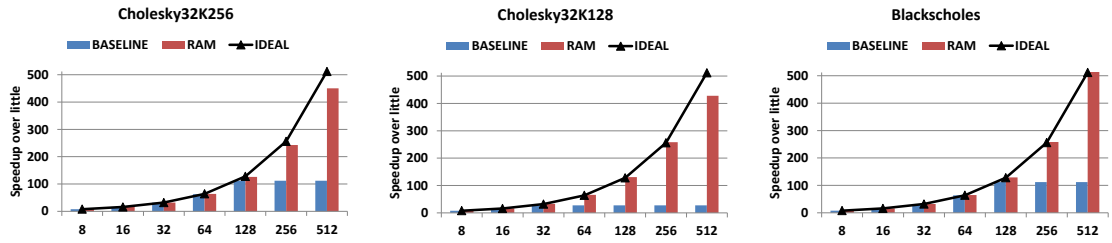


Fig. 4.3 Speedups obtained for default OmpSs runtime (BASELINE) and RAM for each application

Part III

Epilogue

Chapter 5

Conclusion

The goal for this PhD thesis is to incorporate techniques of task and thread scheduling in order to fully utilize asymmetric systems. The main contributions of the thesis rely on the efficient exploitation of future asymmetric multi-core systems in terms of performance and energy efficiency as well as on the development of future asymmetric systems that fit the needs of high performance computing. Existing parallel scientific applications will become portable when moving from a traditional multi-core to an asymmetric multi-core system. Our useful observations throughout this study will also contribute and give guidelines for the design of the future multi-core asymmetric systems for high performance computing.

Our current results have shown that the state-of-the-art asymmetric multi-core systems are not ready to efficiently run out-of-the-box high performance applications and that the most efficient way is by using a task-based approach. This increases the need of research in this direction through the paths of scheduling and thread migration as described in the previous Chapters. In our first attempts to follow these paths, we have seen the high potential of the criticality-aware task schedulers to speed up dependency-intensive applications and take advantage of the asymmetric compute resources.

We are optimistic that following our second research approach of runtime thread migration will also contribute positively. The greatest challenge will be to increase performance without sacrificing energy, thus the dynamic search for the appropriate assistant core for the runtime thread has to consider all these obstacles. We expect that this approach will also influence designers to consider the use of assistant cores in the future asymmetric multi-cores. Finally, in our last and most complete approach we will need to synchronize all of our tools (e.g. scheduling and thread migration) to adapt to the runtime circumstances and boost performance with decent energy consumption.

Since a part of this work is already complete, we expect that our goals will be successfully accomplished and this study will be a useful reference for the future research.

References

[1] Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). A Comparison of List Schedules for Parallel Processing Systems. *Commun. ACM*, 17(12):685–690.

[2] Agarwal, A. and Kumar, P. (2009). Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 87–93.

[3] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198.

[4] Ayguadé, E., Badia, R., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Pérez, J., Planas, J., and Quintana-Ortí, E. (2010). Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, 38(5-6):440–459.

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

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

[7] Bansal, S., Kumar, P., and Singh, K. (2003). An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(6):533–544.

[8] Barcelona Supercomputing Center (https://pm.bsc.es/projects/bar). BSC Application Repository.

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

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

[11] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2007). Parallel tiled QR factorization for multicore architectures. Technical report.

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

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

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

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

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

[17] Daoud, M. and Kharma, N. (2006). Efficient Compile-Time Task Scheduling for Heterogeneous Distributed Computing Systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 9 pp.–.

[18] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193.

[19] Duran, A., Corbalán, J., and Ayguadé, E. (2008a). Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110.

[20] Duran, A., Perez, J. M., Ayguadé, E., Badia, R. M., and Labarta, J. (2008b). Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 111–122.

[21] Fedorova, A., Saez, J. C., Shelepov, D., and Prieto, M. (2009). Maximizing Power Efficiency with Asymmetric Multicore Systems. *Communications of the ACM*, 52(12):48.

[22] Greenhalgh, P. (2011). big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, (September 2011):1–8.

[23] Hakem, M. and Butelle, F. (2005). Dynamic Critical Path Scheduling Parallel Programs onto Multiprocessors. In *In the Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'05, pages 203b–203b.

[24] Iverson, M. A., Özgüner, F., and Follen, G. J. (1995). Parallelizing Existing Applications in a Distributed Heterogeneous Environment. In *4th Heterogeneous Computing Workshop (HCW '95)*, pages 93–100.

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

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

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

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

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

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

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

[32] Li, K., Zhang, Z., Xu, Y., Gao, B., and He, L. (2012). Chemical Reaction Optimization for Heterogeneous Computing Environments. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 17–23.

[33] Liu, C.-H., Li, C.-F., Lai, K.-C., and Wu, C.-C. (2006). A dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 8 pp.–.

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

[35] OpenMP Architecture Review Board (2013). OpenMP architecture review board: Application program interface.

[36] Page, A. and Naughton, T. (2005). Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 189a–189a.

[37] Pienaar, J. A., Chakradhar, S., and Raghunathan, A. (2012). Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 24:1–24:12.

[38] Planas, J., Badia, R., Ayguade, E., and Labarta, J. (2013). Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 138–149.

[39] Rajovic, N., Carpenter, P. M., Gelado, I., Puzovic, N., Ramirez, A., and Valero, M. (2013). Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13.

[40] Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M. (2012). On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20.

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

[42] Sih, G. and Lee, E. (1993). A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):175–187.

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

[44] Topcuoglu, H., Hariri, S., and Wu, M.-Y. (2002). Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

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

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

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

[48] Vandierendonck, H., Tzenakis, G., and Nikolopoulos, D. S. (2011). A unified scheduler for recursive and task dataflow parallelism. In *PACT*, pages 1–11.

[49] Wu, M.-Y. and Gajski, D. (1990). Hypertool: a Programming Aid for Message-Passing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(3):330–343.

[50] Yang, T. and Gerasoulis, A. (1994). DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *Parallel and Distributed Systems, IEEE Transactions on*, 5(9):951–967.

[51] Yu, H. (2007). A Hybrid GA-based Scheduling Algorithm for Heterogeneous Computing Environments. In *IEEE Symposium on Computational Intelligence in Scheduling*, SCIS'07, pages 87–92.

[52] Zong, Z., Manzanares, A., Ruan, X., and Qin, X. (2011). EAD and PEBD: Two Energy-Aware Duplication Scheduling Algorithms for Parallel Tasks on Homogeneous Clusters. *Computers, IEEE Transactions on*, 60(3):360–374.

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