Exploiting Asymmetric Systems with Flexible System Software

Kallia Chronaki

Department of Computer Architecture

Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of

Doctor of Philosophy

April 2018

BLANK

ABSTRACT

In 2013, U.S. data centres accounted for 2.2% of the country's total electricity consumption, a figure that is projected to increase rapidly over the next decade. A significant proportion of power consumed within a data centre is attributed to the servers, and a large percentage of that is wasted as workloads compete for shared resources. Many data centres host interactive workloads (e.g., web search or e-commerce), for which it is critical to meet user expectations and user experience, called Quality of Service (QoS). There is also a wish to run both interactive and batch workloads on the same infrastructure to increase cluster utilisation and reduce operational costs and total energy consumption. Although much work has focused on the impacts of shared resource contention, it still remains a major problem to maintain QoS for both interactive and batch workloads. The goal of this thesis is twofold. First, to investigate how, and to what extent, resource contention has an effect on throughput and power of batch workloads via modelling. Second, we introduce a scheduling approach to determine on-the-fly the best configuration to satisfy the QoS for latency-critical jobs on any architecture.

BLANK

Exploiting Asymmetric Systems with Flexible System Software

by
Kallia Chronaki

A Dissertation
Presented to the Department of Computer Architecture
at
Universitat Politècnica de Catalunya
in Candidacy for the Degree of
Doctor of Philosophy.

Thesis Advisors:

Rosa M. Badia, Prof.
Universitat Politècnica de Catalunya, Spain

Marc Casas, Prof.
Universitat Politècnica de Catalunya, Spain

Barcelona, July 2018

BLANK

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The use of heterogeneous processing elements is becoming commodity in many aspects of parallel computing. From mobile devices to high performance supercomputers heterogeneous multi-processing is attracting a lot of attention as it achieves high performance and at the same time it maintains energy consumption at low levels. Asymmetric multi-core systems is an interesting type of heterogeneous multi-processor. These systems maintain different types of cores that share the same instruction set architecture. The different core types are designed to target different optimization points. The current state-of-the-art asymmetric system architecture is the Arm big.little architecture [41, 46]. This architecture combines two types of cores: the out-of-order performance-optimized *big* cores and the in-order energy efficient *little* cores. Even though this asymetric multi-core architecture is mainly used on mobile devices, it is a very interesting approach for HPC as the combnation of fast and slow core units can bring benefits in energy consumption.

The future of parallel computing is highly restricted by energy efficiency [50]. Energy efficiency has become the main challenge for future processor designs, motivating prolific research to face the *power wall*. Using heterogeneous processing elements is one of the approaches to increase energy efficiency [61, 91]. Asymmetric multi-core (AMC) systems is an interesting case of heterogeneous systems to utilize for energy efficiency. These systems maintain different types of cores that support the same instruction-set architecture. The different core types are designed to target different (performance or power) optimization points [10, 54, 89].

AMCs have been mainly deployed for the mobile market. Mobile processors are also utilized in HPC platforms aiming to energy savings [56]. Asymmetric mobile SoCs combine low-power simple cores (*little*) with fast out-of-order cores (*big*) to achieve high per-

formance while keeping power dissipation low. Another area where AMCs have been successful is the supercomputing market. The Sunway TaihuLight supercomputer topped the Top500 list in 2016 using AMCs. In this setup, big cores, that offer support for speculation to exploit Instruction-Level Parallelism (ILP), run the master tasks such as the OS and runtime system. Little cores are equipped with wide Single Instruction Multiple Data (SIMD) units and lean pipeline structures for energy efficient execution of compute-intensive code.

Like in other heterogeneous systems, load balancing and scheduling are fundamental challenges that must be addressed to effectively exploit all the resources in AMC platforms [39, 41, 48, 49, 72, 80]. Mobile applications rely on multi-programmed workloads to balance the load in the system, while supercomputer applications rely on hand-tuned code to extract maximum performance. However, these approaches are not always suitable for general-purpose parallel applications.

In this paper, we evaluate several execution models on an AMC using the PARSEC benchmark suite [95]. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We quantify the performance loss of executing the applications *as-is* on all cores in the system. These applications were developed on homogeneous platforms and are bound to suffer from load imbalance on parallel regions that statically distribute the work evenly across cores without considering their performance disparity.

To overcome this matter, we consider two possible solutions at the OS and runtime levels to exploit AMCs effectively. The first solution delegates scheduling to the OS. We evaluate the built-in heterogeneity-aware OS scheduler currently used in existing mobile platforms that automatically assigns threads to different core types based on CPU utilization.

The second solution is to transfer the responsibility to the runtime system so it dynamically schedules work to different core types based on work progress and core availability. We evaluate the impact of using an inherently load-balanced execution model such that of task-based programming models. Recent examples [9, 14, 34, 35, 66, 74, 82, 87, 88] include clauses to specify inter-task dependencies and remove most barriers which are the major source of load imbalance on AMCs. Another approach of scheduling in the runtime system is to change the existing statically-scheduled work-sharing constructs for the applications implemented in OpenMP to use dynamic scheduling.

This paper provides the first to our knowledge comprehensive evaluation of representative parallel applications on a real AMC platform: the Odroid-XU3 development board with ARM big.LITTLE architecture. We analyze the effectiveness of the aforementioned scheduling solutions in terms of performance, power and energy. We show why parallel applications are not ready to run on AMCs and how OS and runtime schedulers can overcome

these issues depending on the application characteristics. Further we point out in which aspects the built-in OS scheduler falls short to effectively utilize the AMC. Finally, we show how the runtime system approach overcomes these issues, and improves the OS and static threading approaches by 13% and 23% respectively.

The rest of this document is organized as follows: Section 5.2 describes the evaluated AMC processor, while Section 4.3 provides information on scheduling at the OS and runtime system levels. Section 5.4 describes the experimental framework. Section 5.5 shows the performance and energy results and associated insights.Finally, Section 5.6 discusses related work and Section 5.7 concludes this work.

CHAPTER 2

BACKGROUND

To familiarize the reader, this chapter describes the background of this thesis. This study includes software enhancements targeting asymetric multi-core systems. Thus, we first give some background information about the asymetric multi-core architecture used in this work, which is the ARM big.LITTLE architecture [41]. The second part of this chapter, provides information about the parallel programming models, that is the currently used method for parallel programming in HPC applications. Finally we we provide a high-level description of the applications used in this work.

2.1 THE ARM BIG.LITTLE ARCHITECTURE

The ARM big.LITTLE [28, 41] is a state-of-the-art AMC architecture that has been successfully deployed in the mobile market. The observation that mobile devices typically combine phases with low and high computational demands motivated this original design. ARM big.LITTLE combines simple in-order cores with aggressive out-of-order cores in the same System-on-Chip (SoC) to provide high performance and low power. *Big* and *little* cores support the same architecture so they can run the same binaries and therefore easily combined within the same system. Current cores implementing the ARMv7-A and ARMv8-A ISA support big.LITTLE configurations.

The little cores in a big.LITTLE system are designed targeting energy efficiency. Current implementations have relatively short pipelines with up to dual-issue in-order execution. L1 caches are split for instructions and data and can be dimensioned according to the target domain from 8 to 64 KB in size [52]. The big cores are designed for high performance. Current designs have deeper pipelines with up to seven-issue out-of-order execution, increased

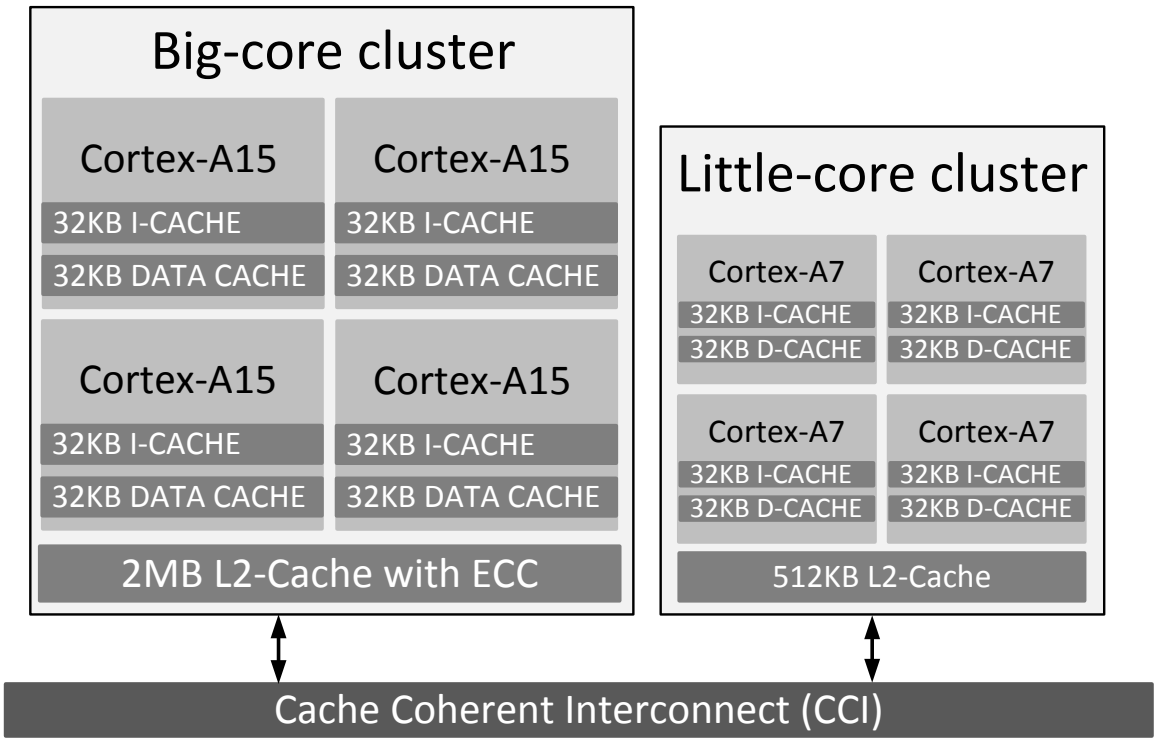


Figure 2.1 Samsung Exynos 5422 processor with ARM big.LITTLE architecture.

number of functional units and improved floating-point capabilities. L1 data cache is up to 64 KB and L1 instruction cache is up to 64 KB[19, 32, 42]. Little and big cores are typically integrated in a hierarchical manner. A set of cores form a cluster that may include a cache that is shared among cores in the cluster [42]. Then, multiple clusters can be interconnected through an on-chip network and share a last-level cache and connection to main memory and peripherals.

In this thesis, we use of one of the commercially available development boards featuring a big.LITTLE architecture: the Hardkernel Odroid-XU3 development board. As shown in Figure 2.1, the Odroid-XU3 includes an 8-core Samsung Exynos 5422 chip with four ARM Cortex-A15 cores and four Cortex-A7 cores. The four Cortex-A15 share a 2 MB 16-way 64-byte-cache-line L2 cache, while the Cortex-A7 cores share a 512 KB L2 cache. A single memory controller provides access to 2 GB of LPDDR3 RAM with dual 32-bit channels at 1866 MT/s. The reason we use this platform instead of the more up-to-date Juno platform [6] is that even if the latter features the more advanced Cortex A53 and Cortex A57 cores, it is limited to six cores instead of the 8 cores in Odroid-XU3.

The Cortex-A7 cores in this SoC support dual-issue of instructions and their pipeline length is between 8 and 10 stages. The L1 instruction cache is 32KB two-way set associative, with virtually indexed and physically tagged cache-lines that can hold up to 8

instructions. The core supports instruction prefetch by predicting the outcome of branches; the prefetch unit can fetch up to a maximum of four instructions per cycle. The L1 data cache is four-way set associative with physically-indexed and physically-tagged cache lines and uses a pseudo-random replacement policy [5]. Dynamic Voltage and Frequency Scaling (DVFS) techniques adjust the frequency of the little cores from 200MHz up to 1.4GHz.

The Cortex-A15 cores in this SoC support triple-issue of instructions and their pipeline length is between 15 and 24 stages [84]. The L1 instruction and data caches of the Cortex-A15 are both 32 KB and 2-way set-associative with 64 byte cache lines. The processor supports speculative instruction execution by maintaining a 2-level global history-based dynamic predictor with a branch target buffer [4]. The instruction decode unit performs register renaming to remove the Write-After-Write and the Write-After-Read hazards, and promote instruction reordering [4]. The instruction dispatch unit analyzes instruction dependences before issuing them for execution. The integer execute unit includes 2 Arithmetic Logical Units with support for operand forwarding. DVFS techniques vary the frequency of the big cores from 200 MHz up to 2 GHz. For the rest of the thesis, we refer to Cortex-A15 cores as *big* and to Cortex-A7 cores as *little*.

All the real machine experiments in this thesis are performed on the Hardkernel Odroid XU3 that features the ARM big.LITTLE architecture. To avoid machine overheating, we make use of the `cpufreq` driver to set big cores at 1.6GHz and little cores at 800MHz.

2.2 Task-Based Parallel Programming Models

Parallel programming models [8, 15, 17, 73], are widely used to facilitate the programming of parallel codes for multi-core systems. These programming models offer an abstraction layer to the programer so that multi-threaded programming of an application becomes easier. They support code annotations that the programmer can add to the application's sequential code and transform it into parallel. These annotations include the specification of parallel loops, atomic operations, critical regions or task clauses.

Our main focus in this thesis is the task annotation with dependency tracking which OpenMP [18] supports since its 4.0 release [67]. A task is a piece of code[1] in the application that can execute simultaneously to other tasks and cooperatively produce results. In a parallel application there can be many tasks that perform the same computations on different data or tasks that perform different computations on the same data. By using task annotations, the programmer decomposes the application into tasks and specifies the input and output data dependencies between them. Parallel programming models typically consist

[1] A piece of code can be a function or a code block.

of two parts; a compiler and a runtime system. The compiler is responsible to parse the code annotations and translate them to 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 well as the availability of resources. This thesis mainly focuses on enhancements of the runtime system in parallel programming models.

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 compiler generated application's code sequentially and creates the tasks as it encounters them. Following task creation, is the analysis of the dependencies of the created task and the insertion of it in the Task Dependency Graph (TDG).

A TDG is a distributed graph structure that connects each task of the application with the rest of the tasks according to their existing input and output data dependencies. A dependency tracking mechanism is responsible for the maintenance of the input and output data of a task. Within this mechanism, the runtime system tracks the memory addresses that tasks write or read and according to this, it manages the tasks that are ready for execution[2]. Tasks that their input data have been produced are marked as *ready* and can start execution, while tasks that are still waiting for input are postponed until their inputs are produced by other executing tasks.

To enable the efficient parallel execution of the tasks, the scheduler of the runtime system keeps track of the available tasks to be executed and manages the task-to-thread allocation. To do this, the scheduler maintains a *ready queue* and when all of a tasks dependencies are satisfied (i.e., the task becomes *ready*) it is inserted in the *ready queue*. All threads have access to this queue which is a first-in-first-out data structure; whenever a thread becomes idle, it pops the next task from the queue and executes it. In this thesis we make use of OmpSs [8], a mainstream task-based programming model and the main influence of the updated OpenMP 4.0 [18].

2.2 OmpSs Programming Model

Kallia: Transform it to OpenMP?? 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 [8, 35]. As a task-based programming model, OmpSs enables the annotation of function declarations with the task directive, which declares a task. Every invocation of a such function creates a task that is

[2]When the input data of a task is produced, the task can start execution.

executed concurrently with other tasks or parallel loops. OmpSs also supports task dependencies and it uses the StarSs [37] dependency tracking mechanisms. OmpSs is built with the support of the Mercurium compiler, responsible for the translation of the OmpSs annotation clauses to source code, and the Nanos++ runtime system [13], responsible for the internal creation and execution of the tasks.

As a task-based parallel programming model, OmpSs enables the annotation of function declarations with the task directive. If a function is declared as a task, then every invocation of this function creates a task that is executed concurrently with other tasks or parallel loops. The accessible data to each task are the arguments of the function. OmpSs uses the StarSs [37] dependency tracking mechanisms and each task may be annotated with the *in*, *out*, *inout* clauses. These clauses allow the specification of scalars, arrays and pointers as input, output or input and output data of a task. The implementation of a barrier is supported under the *taskwait* clause, and it can also be used with the addition of the *on* clause, to declare a barrier for the group of tasks that produce a specific piece of data. These original OmpSs features can now be found in OpenMP 4.0 [18].

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

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 discovered but it is waiting until all its inputs are produced by other 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 multiple threads. When a thread becomes idle, the scheduling policy picks a task from the ready queues for that thread to execute.

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.

2.3 APPLICATIONS

In the evaluation of our contributions we use 13 scientific applications. With the prevalence of many-core processors and the increasing relevance of application domains that do not belong to the traditional HPC field, comes the need for programs representative of current and future parallel workloads. The PARSEC benchmark suite [16, 95] features state-of-the-art, computationally intensive algorithms and very diverse workloads from different areas of computing. In our experiments, we make use of the original PARSEC codes together with a task-based implementation of nine benchmarks of the suite [25]. Additionaly, we evaluate some representative benchmarks from the BSC Application Repository (BAR) [12]. These applications are implemented using the OmpSs programming model.

Table 3.1 describes the benchmarks included in the study along with their respective inputs and parallelization strategy. We are using native inputs, which are real input sets for native execution, except for `dedup`, as the entire input file of 672 MB and the intermediate data structures do not fit in the memory system of our platform. Instead, we reduce the size of the input file to 351 MB.

2.4 THE TASKSIM SIMULATOR

To evaluate our contributions on larger systems we make use of the TaskSim simulator [40, 75]. TaskSim is a trace driven simulator, that supports the specification of homogeneous or heterogeneous systems with many cores. The tracing overhead of the simulator is less than 10% and the simulation is accurate as long as there is no contention in the shared memory resources on a real system [40]. By default, TaskSim allows the specification of the amount of cores and supports up to two core types in the case of heterogeneous asymmetric systems. This is done by specifying the number of cores of each type and their difference in performance between the different types (performance ratio) in the TaskSim configuration file.

Our evaluation consists of experiments on both symmetric and asymmetric platforms with the number of cores varying from 8 to 512. In the case of asymmetric systems, we simulate the behaviour of an ARM big.LITTLE architecture [46].

To simulate our approaches using TaskSim we first run each application/input in the TaskSim trace generation mode. This mode enables the online tracking of task duration and synchronization overheads and stores them in a trace file. To perform the simulation, TaskSim uses the information stored in the trace file and executes the application by providing this information to the runtime system. For our experiments we generate three trace

Table 2.1 Benchmarks used from the PARSEC benchmark suite and their measured performance ratio between big and little cores

Benchmark	**Description**	**Input**	**Parallelization**
Blackscholes	Calculates the prices of a portfolio analytically with the Black-Scholes partial differential equation.	10,000,000 options	data-parallel
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
Canneal	Simulated cache-aware annealing to optimize routing cost of a chip design.	2.5 million elements, 6,000 steps	unstructured
Cholesky Factorization	Dense matrix operation that is used for solving linear equations in linear least square systems.	Kallia: multiple	dependencies
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
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
Ferret	Content-based similarity search of feature-rich data (audio, images, video, etc.)	3,500 queries, 59,695 images database, find top 50 images	pipeline
Fluidanimate	Extended Smoothed Particle Hydrodynamics method to simulate an incompressible fluid for interactive animations.	500 frames, 500,000 particles	data-parallel
Heat diffusion	Computes the heat distribution on a matrix from x heat sources using the Gaus-Seidel method.	16×16 blocks of 512×512 doubles	data-parallel
Integral Histogram	A method to compute a cumulative histogram for each pixel of an image represented as a Cartesian data space in constant time.	8×8 blocks of 12×512 floats	dependencies
QR Factorization	A linear algebra algorithm that is used to solve the linear least squares problem [21].	Kallia: multiple	dependencies
Streamcluster	Solves the online clustering problem.	200K points per block, 5 block	data-parallel
Swaptions	Intel RMS workload; uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions.	128 swaptions, 1 million simulations	data-parallel

files for each application/input combination on a Genuine Intel 16-core machine running at 2.60GHz.

CHAPTER 3

STUDY OF ASYMMETRIC SYSTEMS

3.1 INTRODUCTION

Energy efficiency has become the main challenge for future processor designs, motivating prolific research to face the *power wall*. Using heterogeneous processing elements is one of the approaches to increase energy efficiency [61, 91]. Asymmetric multi-core (AMC) systems is an interesting case of heterogeneous systems to utilize for energy efficiency. These systems maintain different types of cores that support the same instruction-set architecture. The different core types are designed to target different (performance or power) optimization points [10, 54, 89].

AMCs have been mainly deployed for the mobile market. Mobile processors are also utilized in HPC platforms aiming to energy savings [56]. Asymmetric mobile SoCs combine low-power simple cores (*little*) with fast out-of-order cores (*big*) to achieve high performance while keeping power dissipation low. Another area where AMCs have been successful is the supercomputing market. The Sunway TaihuLight supercomputer topped the Top500 list in 2016 using AMCs. In this setup, big cores, that offer support for speculation to exploit Instruction-Level Parallelism (ILP), run the master tasks such as the OS and runtime system. Little cores are equipped with wide Single Instruction Multiple Data (SIMD) units and lean pipeline structures for energy efficient execution of compute-intensive code.

Like in other heterogeneous systems, load balancing and scheduling are fundamental challenges that must be addressed to effectively exploit all the resources in AMC platforms [39, 41, 48, 49, 72, 80]. Mobile applications rely on multi-programmed workloads to balance the load in the system, while supercomputer applications rely on hand-tuned code to extract maximum performance. However, these approaches are not always suitable for

general-purpose parallel applications.

In this chapter, we evaluate several execution models on an AMC using the PARSEC benchmark suite [95]. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We quantify the performance loss of executing the applications *as-is* on all cores in the system. These applications were developed on homogeneous platforms and are bound to suffer from load imbalance on parallel regions that statically distribute the work evenly across cores without considering their performance disparity.

To overcome this matter, we consider two possible solutions at the OS and runtime levels to exploit AMCs effectively. The first solution delegates scheduling to the OS. We evaluate the built-in heterogeneity-aware OS scheduler currently used in existing mobile platforms that automatically assigns threads to different core types based on CPU utilization.

The second solution is to transfer the responsibility to the runtime system so it dynamically schedules work to different core types based on work progress and core availability. We evaluate the impact of using an inherently load-balanced execution model such that of task-based programming models. Recent examples [9, 14, 34, 35, 66, 74, 82, 87, 88] include clauses to specify inter-task dependencies and remove most barriers which are the major source of load imbalance on AMCs. Another approach of scheduling in the runtime system is to change the existing statically-scheduled work-sharing constructs for the applications implemented in OpenMP to use dynamic scheduling.

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

The rest of this chapter is organized as follows: Section 4.3 provides information on scheduling at the OS and runtime system levels while Section 5.4 describes the experimental framework. Section 5.5 shows the performance and energy results and associated insights.Finally, Section 5.6 discusses related work and Section 5.7 concludes this work.

3.2 SCHEDULING IN ASYMMETRIC MULTI-CORES

Scheduling a set of processes on an AMC system is more challenging than the traditional process scheduling on SMCs. An efficient OS scheduler has to take into account the different characteristics of the cores and act accordingly [22]. There have been three mainstream OS schedulers for ARM big.LITTLE systems: *cluster switching*, *in-kernel switch* and *global task scheduling*, described in the next sections. In the case of parallel applications, *dynamic scheduling at the runtime system level* can be exploited to balance the workload among the different cores and is described in section 3.2.3.

3.2 CLUSTER SWITCHING AND IN-KERNEL SWITCH

In the Cluster Switching (CS) approach [28], only one of the clusters is active at any given time: either the cluster with little cores or the cluster with big cores executes. Thus, the OS scheduler operates on a *de-facto* symmetric multi-core with only four cores, namely the cores of the current active cluster. The policy to change the operating cluster is based on CPU utilization. When idle, background processes are executed on the little cores. When CPU utilization surpasses a threshold, all processes (foreground and background) are migrated to the big cluster. When running on the big cluster, if CPU utilization decreases below a given lower threshold, the entire workload is moved to the little cluster.

In the In-Kernel Switch (IKS) approach [63], each little core is paired with a big core and it is seen as a single core. On idle, background processes are run on little cores. When the CPU utilization on a given little core surpasses a threshold, the execution on that core is migrated to the big core. When the CPU utilization decreases on that big core below a given threshold, the execution migrates to the associated little core. Thus, at the same time, little and big cores may co-execute, but only one of each pair is active at a given point in time, effectively exploiting just half of the cores concurrently. For both CS and IKS, an enhanced `cpufreq` driver manages the switching within each core pair.

3.2 GLOBAL TASK SCHEDULING

The Global Task Scheduling (GTS) [28] allows running applications on all cores in the asymmetric multi-core. In GTS, all cores are available and visible to the OS scheduler, and this scheduler is aware of the characteristics of the core types. Each process is assigned to a core type depending on its CPU utilization: high CPU utilization processes are scheduled to big cores and low CPU utilization processes to little cores. GTS also migrates processes between big and little cores when their CPU utilization changes. As a result, cores are active

depending on the characteristics of the workload.

The key benefit of GTS is that it can use all the cores simultaneously, providing higher peak performance and more flexibility to manage the workload. In GTS tasks are directly migrated to cores without needing the intervention of the `cpufreq` daemon, reducing response time and minimizing the overhead of context switches. As a consequence, Samsung reported 20% improvement in performance over CS for mobile benchmarks [28]. Also, GTS supports clusters with different number of cores (e.g. with 2 big cores and 4 little cores), while IKS requires to have the same number of cores per cluster.

3.2 Dynamic Scheduling in the Runtime

Current programming models for shared memory systems such as OpenMP rely on a runtime system to manage the execution of the parallel application. In this work, we make use of two types of programming models: loop- and task-based. Loop-based scheduling distributes the iterations of a loop among the threads available in the system, following a traditional *fork-join* model. OpenMP supports loop-based scheduling through its *parallel for* directives. This clause implies a barrier synchronization at the end of the loop[1], and supports either static or dynamic loop scheduling.

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

Recent advances in programming models recover the use of task-based programming models to simplify parallel programming of multi-cores [14, 35, 66, 88, 98]. In these models the programmer splits the code in sequential pieces of work (tasks) and specifies the data dependencies among them. With this information the runtime system schedules tasks and manages synchronization. These models ease programmability [14, 35, 66, 87, 88, 98], and also increase performance by avoiding global synchronization points.

To evaluate this approach we make use of OpenMP tasking support [66]. OpenMP allows expressing tasks and data dependences between them using equivalent code annotations. It conceives the parallel execution as a *task dependence graph* (TDG), where nodes are sequential pieces of code (tasks) and the edges are control or data dependences between them. The runtime system builds this TDG at execution time and dynamically schedules tasks to the available cores. Tasks become ready as soon as their input dependencies are sat-

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

Table 3.1 Benchmarks used from the PARSEC benchmark suite and their measured performance ratio between big and little cores

Benchmark	Input	Parallelization	Perf ratio
blackscholes	10,000,000 options	data-parallel	2.18
bodytrack	4 cameras, 261 frames, 4,000 particles, 5 annealing layers	pipeline	4.16
canneal	2.5 million elements, 6,000 steps	unstructured	1.73
dedup	351 MB data	pipeline	2.67
facesim	100 frames, 372,126 tetrahedra	data-parallel	3.40
ferret	3,500 queries, 59,695 images database, find top 50 images	pipeline	3.59
fluidanimate	500 frames, 500,000 particles	data-parallel	3.32
streamcluster	200K points per block, 5 block	data-parallel	3.48
swaptions	128 swaptions, 1 million simulations	data-parallel	2.78

isfied. The scheduling of the ready tasks is done in a first-come-first-served manner, using a FIFO scheduler. Even though this scheduler is not aware of the task computational requirements or the core type and its performance and power characteristics, it can balance the load as long as there are ready tasks available thanks to the lack of global synchronization.

3.3 Experimental Methodology

3.3 Metrics

All the experiments in this paper are performed on the Hardkernel Odroid XU3 described in Section 5.2. To avoid machine overheating, we make use of the `cpufreq` driver to set big cores at 1.6GHz and little cores at 800MHz.

We evaluate seven configurations with different numbers of *little* (L) and *big* (B) cores, denoted `L+B`. For each configuration and benchmark, we report the average performance of five executions in the application parallel region. Then, we report the application speedup over its execution time on one little core. Equation 4.1 shows the formula to compute this speedup.

$$\text{Speedup(L, B, method)} = \frac{\text{Exec. time(1, 0, method)}}{\text{Exec. time(L, B, method)}} \tag{3.1}$$

In this platform, there are four separated current sensors to measure, in real time, the power consumption of the A15 cluster, the A7 cluster, the GPU and DRAM. To gather power and energy measurements, a background daemon reads the machine power sensors

periodically during the application execution with negligible overhead. Sensors are read at their refresh rate, every 270ms, and the values of A7 and A15 clusters' sensors are collected. With the help of timestamps, we correlate the power measurements with the application parallel region in a *post-mortem* process[2]. The reported power consumption is the average power tracked during five executions of each configuration, considering the application parallel region only. We then report average power in Watts along the execution.

Finally, in terms of energy and Energy Delay Product (EDP), we report the total energy and EDP of the benchmarks region of interest normalized to the run on four little cores with static threading. Equations 3.2 and 3.3 show the formulas for these calculations.

$$\text{Normalized Energy(L, B, method)} = \frac{\text{Energy(L, B, method)}}{\text{Energy(4, 0, static-threading)}} \tag{3.2}$$

$$\text{Normalized EDP(L, B, method)} = \frac{\text{EDP(L, B, method)}}{\text{EDP(4, 0, static-threading)}} \tag{3.3}$$

3.3 APPLICATIONS

With the prevalence of many-core processors and the increasing relevance of application domains that do not belong to the traditional HPC field, comes the need for programs representative of current and future parallel workloads. The PARSEC benchmark suite [16, 95] features state-of-the-art, computationally intensive algorithms and very diverse workloads from different areas of computing. In our experiments, we make use of the original PARSEC codes together with a task-based implementation of nine benchmarks of the suite [25].

Table 3.1 describes the benchmarks included in the study along with their respective inputs, parallelization strategy and performance ratio between big and little cores per application. We are using native inputs, which are real input sets for native execution, except for `dedup`, as the entire input file of 672 MB and the intermediate data structures do not fit in the memory system of our platform. Instead, we reduce the size of the input file to 351 MB.

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

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

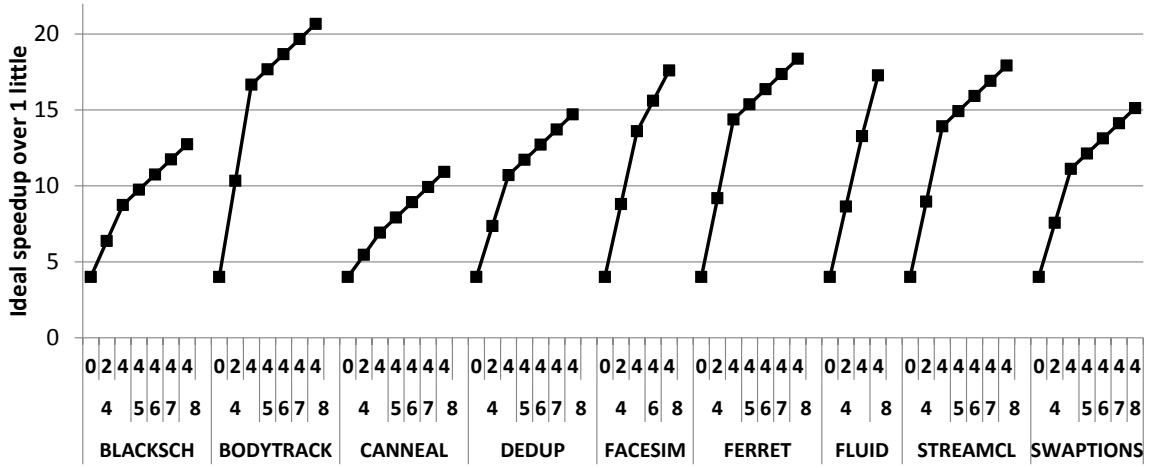


Figure 3.1 Ideal speedup over 1 little core according to Equation 3.4. Numbers at the bottom of x axis show the total number of cores, numbers above it show the number of big cores

When running on the big.LITTLE processor, each benchmark exhibits different performance ratios between big and little cores. These ratios tell us how many times faster a big core is compared to a little core. We measure the performance ratio of each application by executing it first on one big core and then on one little core, which corresponds to Speedup(0, 1, task-based) in Equation 4.1. Table 3.1 also includes the observed performance ratio for each application. Bodytrack is the application that benefits the most from running on the big core with a performance ratio of 4.16×. The out-of-order execution of the big core together with an increased number of in-flight instructions significantly improves the performance of this application. In contrast, canneal is the benchmark with the lowest performance ratio, 1.73×, as this is a memory-intensive benchmark that does not benefit as much from the extra computation power of the big core. In general, performance ratios are above 2.5× for seven out of nine benchmarks, reaching 3.03× on average.

Taking into account these performance ratios, we can estimate the ideal speedup of the platform for each workload assuming a perfect parallelization strategy. Equation 3.4 shows the equation for the ideal speedup over 1 little core computation according to the number of big (B) and little (L) cores.

$$\text{Ideal speedup(workload, B, L)} = \text{B} \times \text{Perf\_ratio(workload)} + \text{L} \tag{3.4}$$

Figure 3.1 shows the ideal speedup of the system for each application for the varying numbers of cores. This speedup assumes that the applications are fully parallel with no barriers or other synchronization points. Thus, these speedups are an upper bound of the achievable application performance.

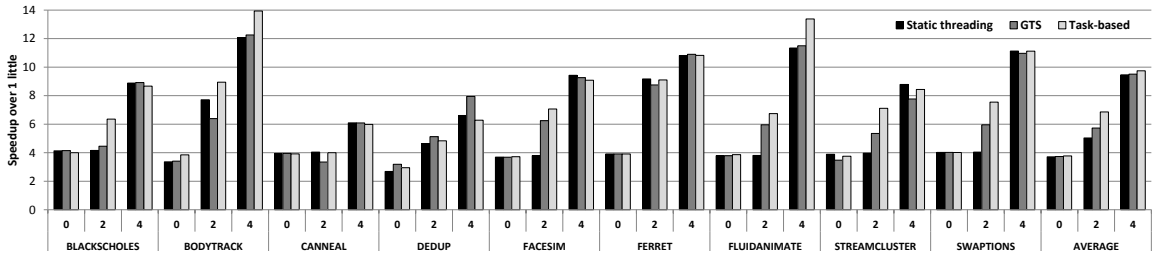


Figure 3.2 Execution time speedup over 1 little core for systems that consist of 4 cores in total with 0, 2 and 4 big cores. Different schedulers at the application (*static threading*), OS (*GTS*) and runtime (*task-based*) levels are considered.

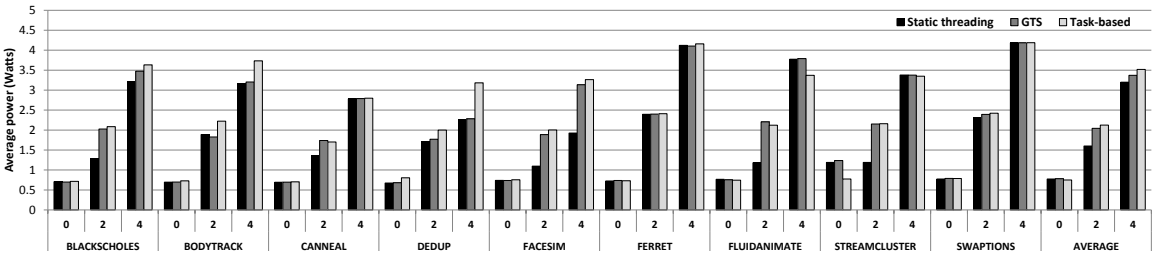


Figure 3.3 Average power measurements on a 4-core system with 0, 2, and 4 big cores.

3.4 EVALUATION

We measure execution time, power, energy and EDP of nine applications from the PARSEC benchmark suite [16]. We compare these metrics for three different scheduling approaches:

- *Static threading*: scheduling decisions are made at the application level. The OS is not allowed to migrate threads between the clusters of big and little cores.
- *GTS*[3]: dynamic coarse-grained OS scheduling using the GTS scheduler integrated in the Linux kernel [28, 46] using the default PARSEC benchmarks.
- *Task-based*: dynamic fine-grained scheduling at the runtime level with the task-based implementations of the benchmarks provided in PARSECSs [25].

3.4 EXPLOITING PARALLELISM IN AMCS

This section examines the opportunities and challenges that current AMCs offer to emerging parallel applications. With this objective, we first evaluate a system with a constant number of four cores, changing the level of asymmetry to evaluate the characteristics of each configuration. In these experiments, all applications run with the original parallelization

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

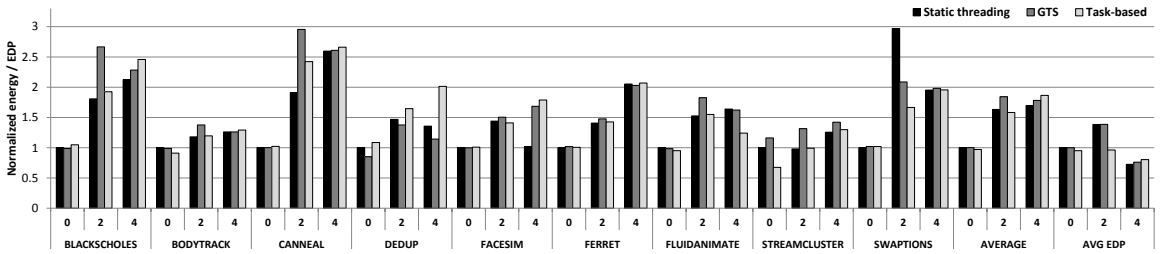


Figure 3.4 Normalized energy consumption and average EDP on a 4-core system with 0, 2, and 4 big cores. Static threading on 4 little cores is the baseline in both cases.

strategy that relies on the user to balance the application (*Static threading*). We also evaluate the OS-based dynamic scheduling (*GTS*) and the task-based runtime dynamic scheduling (*Task-based*) for the same applications. The system configurations evaluated in this section are: i) Four little cores (`0+4`); ii) Two big and two little cores (`2+2`); and iii) Four big cores (`4+0`)

For these configurations, Figure 3.2 shows the speedup of the PARSEC benchmarks with respect to running on a single little core. Figure 3.3 reports the average power dissipated on the evaluated platform. Finally, Figure 3.4 shows the total energy consumed per application for the same configurations. Energy results are normalized to the energy measured with four little cores (higher values imply higher energy consumptions). Average EDP results are also included in this figure.

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

Contrarily, in terms of power and energy, the most efficient configuration is running with four little cores, as the performance ratio between the different cores is inversely proportional to the power ratio [41]. On average, all the approaches reach a power dissipation of 0.75W for the `0+4` configuration, while *Task-based* reaches 3.5W for the `4+0` configuration which is the one with the highest average power dissipation. In configuration `2+2`, average energy values for *Static threading* and *Task-based* are nearly the same, as the increase in power from 1.6W to 2.1W is compensated by a significant improvement in performance of 30%.

Finally, in terms of EDP using the four big cores is the optimal, as the performance improvements compensate the increase in total energy. In configuration `2+2`, *Task-based*

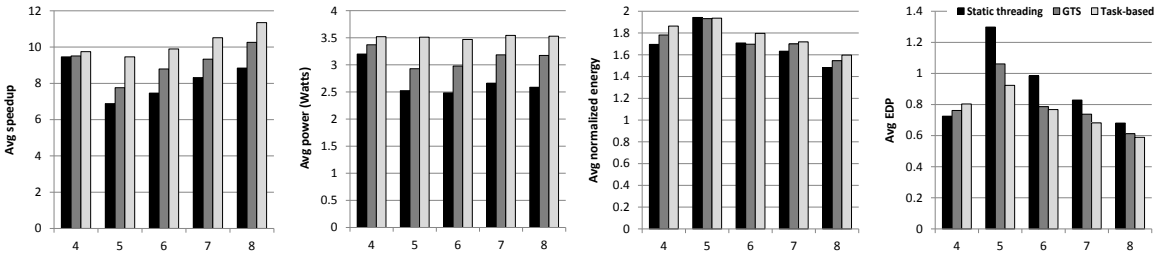


Figure 3.5 Average results when running on 4 to 8 cores with 4 of them big. Speedup is over 1 little core. Static threading on 4 little cores is the baseline of energy consumption and EDP

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

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

The three remaining applications are parallelized using a pipeline model (bodytrack, dedup, and ferret) with queues for the data-exchange between pipeline stages and application-specific load balancing mechanisms designed by the programmer. As a result, *Static scheduling* with these applications benefits from the extra computational power of the big cores in the configuration 2+2. These mechanisms are not needed in the *Task-based* code; in this approach the code of the application is simplified and the runtime automatically allows the overlapping of the different pipeline stages. Thus, on the asymmetric configuration, *Task-based* further improves the obtained performance, reaching a 13% average improvement over *GTS*. Clearly, these applications benefit in performance by the increased number of big cores, while power and energy are increasing since the big cores are effectively utilized.

Generally, relying on the programmer to statically schedule asymmetric configurations

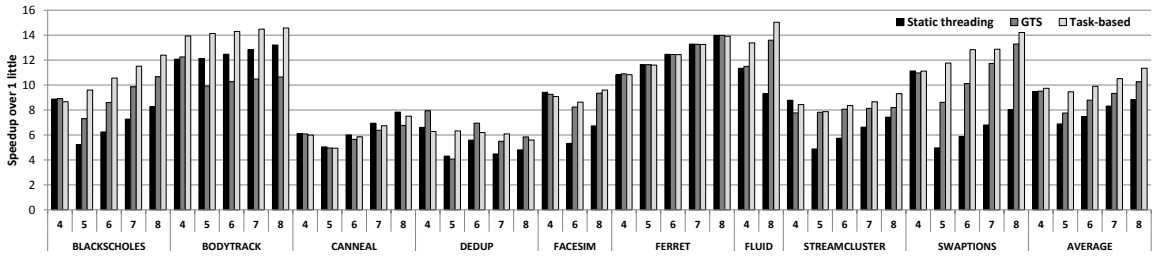


Figure 3.6 Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big

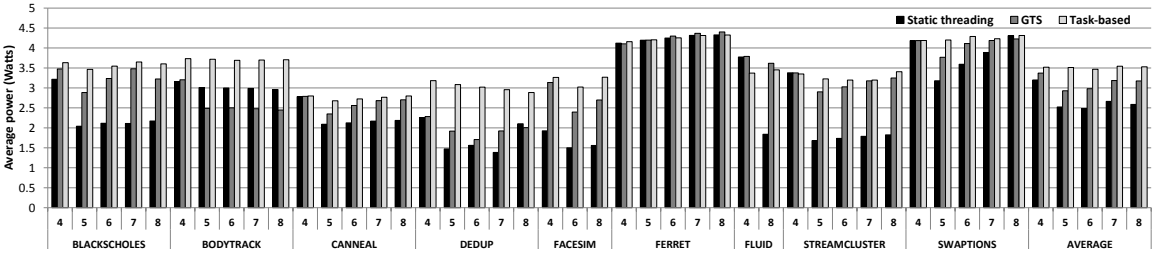


Figure 3.7 Average power when running on 4 to 8 cores and 4 of them are big

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

3.4 Adding Little Cores to an SMC

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

To evaluate this scenario, we explore configurations `4+0`, `4+1`, `4+2`, `4+3` and `4+4`. In these experiments, the number of big cores remains constant (four), while the number of little cores increases from 0 to 4. First we focus on the average results of speedup, power, energy and EDP, shown in Figure 3.5.

The speedup chart of Figure 3.5 shows that *Static threading* does not benefit from adding little cores to the system. In fact, this approach brings an average 6% slowdown when adding four little cores for execution (`4+4`). This is a result of the static thread scheduling; because

the same amount of work is assigned to each core, when the big cores finish the execution of their part, they become idle and under-utilized. GTS achieves a limited speedup of 8% with the addition of four little cores to the `4+0` configuration. The addition of a single little core brings a 22% slowdown (from `4+0` to `4+1`) and requires three additional little cores to reach the performance of the symmetric configuration (`4+3`). Finally, the *Task-based* approach always benefits from the extra computational power as the runtime automatically deals with load imbalance. Performance improvements keep growing with the additional little cores, reaching an average improvement of 15% over the symmetric configuration when 4 extra cores are added.

The power chart of Figure 3.5 shows oppositional benefits among the three approaches. We can see that *Static threading* and *GTS* benefit from asymmetry, effectively reducing average power consumption. *Static threading* reduces power consumption when moving from the `4+0` to the `4+4` system by 23% while *GTS* does so by 6.2%. On the other hand, the *task-based* approach keeps the big cores busy for most of the time so it maintains the average power nearly constant.

The reduction in power, results to reduced average energy in the case of *Static threading* in configuration `4+4`, as shown on the energy chart of Figure 3.5. As discussed in Section 3.4.1, little cores are more energy efficient than big cores, at the cost of reduced performance. In all the approaches, at least two extra little cores are needed to reduce energy. In configuration `4+4`, energy is reduced by 14% for *Static threading*, 15% for *GTS*, and 16% for *Task-based*. Consequently, we can state that asymmetry reduces overall energy consumption.

To see the impact on both performance and energy efficiency we plot the average EDP on the rightmost chart of Figure 3.5. In this chart the lower values are the better. The *task-based* approach is the one that has the best performance-energy combination for the asymmetric configurations since it maintains the lowest EDP for all cases. *Static threading* manages to reduce the average EDP by 6% while *GTS* and *task based* approaches do so by 24% and 36% respectively.

Figure 3.6 shows a more detailed exploration of the performance results. As Table 3.1 shows, the applications with barrier synchronization are blackscholes, facesim, fluidanimate, streamcluster and swaptions. For these applications the most efficient system configuration with the *Static threading* approach is the `4+0`. Little cores increase execution time due to load imbalance effects. Since the big cores reach barriers earlier, power is reduced for these applications, as shown in Figure 3.7. Energy reduction is less significant with a few extra little cores as the performance degradation is higher, but as the number of little cores increases, energy is reduced.

Applications with more advanced load balancing techniques like pipelined parallelism (bodytrack, dedup and ferret), benefit of the asymmetric hardware and balance the load among all the cores. As a result, performance improves as we increase the number of little cores. In the case of bodytrack, *GTS* reduces performance by 15% when adding four little cores. We attribute this to the cost of the thread migration from one core to the other in contrast to the *Static threading* approach that does not add such overheads. In the case of dedup, results show more variability. This benchmark is very I/O intensive and, depending on the type of core that executes these I/O operations, performance drastically changes. In order to deal with this problem, a smarter dynamic scheduling mechanism would be required. Finally, canneal does not scale according to its ideal speedup reported on Figure 3.1 as it has a memory intensive pattern that limits performance.

Figure 3.7 shows the average power. The barrier-synchronized applications (blackscholes, facesim, fluidanimate, streamcluster and swaptions) reduce power because of their imbalance; since big cores have long idle times with the *Static threading* approach, they do not spend the same power as *GTS* and *Task-based*. For pipeline-parallel applications, both bodytrack and ferret maintain nearly the same power levels among the configurations for each scheduling approach. Dedup is an exception, as the results highly depend on the core that executes the aforementioned I/O operations. Yet, the effect of the lower power for *Static threading* is observed in all the benchmarks and is because the big cores are under-utilized.

This section proves that adding little cores to an SMC with big cores presents significant challenges for the application, OS and runtime developers. Little cores increase load imbalance and can degrade performance as a result. Relying on the programmer to deal with this asymmetry is complex, but a dynamic OS scheduler such as *GTS* helps in mitigating these problems, providing an average performance increase of 10%. However, the optimal performance results are obtained with the *Task-based* approach, as they improve static threading by 23% on average. In terms of power and energy, the AMC provides significant benefits, although the SMC with little cores remains the most energy-efficient configuration. The answer to the question of which system configuration provides the best power-performance balance, can be found on the average EDP chart of Figures 3.4 and 3.5, and is the use of the entire 8-core system with the *Task based* approach.

3.4 Programming Models for AMCs

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

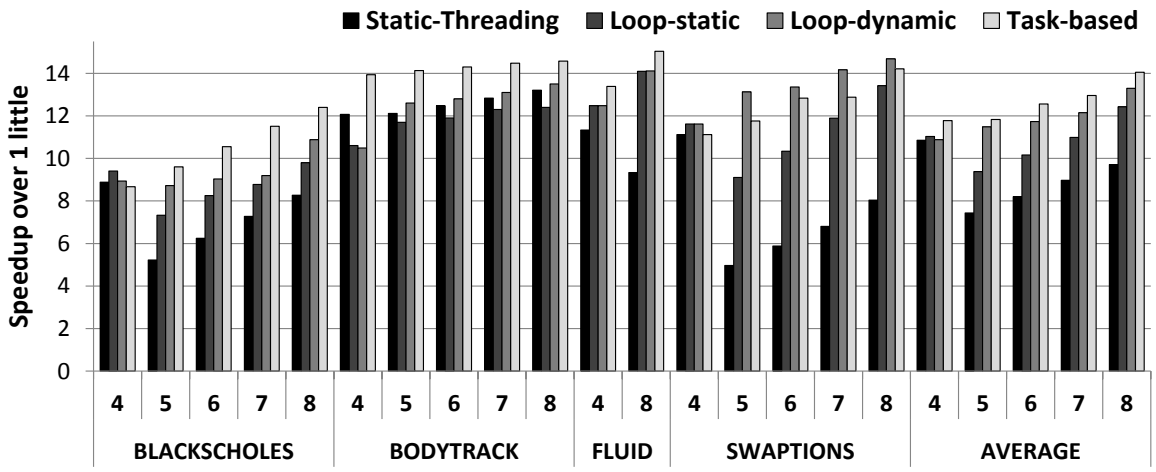


Figure 3.8 Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big. Four different programming models are considered: Static threading using `pthreads`, parallel loops with static scheduling (loop static), parallel loops with dynamic scheduling (loop dynamic), and a task-based solution with dynamic scheduling (task-based).

Implementing advanced load balancing schemes, such as work pools, in `pthreads` requires a significant development effort.

As an alternative, many parallel applications are implemented using loop-based scheduling with the OpenMP *parallel for* directives. In this case, the runtime library is in charge of scheduling work to the available threads in the system, either statically or dynamically, as described in Section 3.2.3.

We compare these solutions to the task-based approach evaluated in the previous sections. Figure 3.8 shows the results obtained from running blackscholes, bodytrack, fluidanimate and swaptions on all the scheduling models: static threading, static loop scheduling, dynamic loop scheduling and task-based scheduling. We chose these applications as they are the only ones implemented using the OpenMP loop directives.

Looking at the average results in Figure 3.8, we can observe that the task-based solution achieves the best results when the system is asymmetric. Task-based improves the static threading by up to 59% on 5 cores, while dynamic loop scheduling improves by up to 54%. The OpenMP version with static scheduling reaches an average 26% improvement over the static-threading approach with pthreads.

Taking a closer look to the results we observe that for bodytrack, an application with sophisticated parallelization techniques, static-threading achieves better results than loop-static. This is because the static-threading implementation contains specific parallelization techniques that cannot be completely expressed using the loop-static method. The loop-dynamic method improves performance for bodytrack by up to 4% due to the runtime

decisions of the iteration execution, but the optimal solution is offered by the task-based approach that achieves up to 16% improvement over static-threading, due to the flexibility in expressing irregular parallelization strategies.

Blackscholes, fluidanimate and swaptions, consist of independent tasks and are a good fit for loop parallelism. The first observation is that both applications benefit from the loop-static approach on an SMC with 4 big cores. Moreover, the task-based approach is still the optimal for blackscholes and fluidanimate, reaching up to 83% improvement over static threading for 5 cores, while for swaptions loop-dynamic is the most appropriate, improving the baseline by up to 2.6×. The difference in the benefits of these applications relies on the task granularity; blackscholes consists of 6400 tasks that are about a hundred times smaller than each one of the 128 tasks of swaptions. This shows that loop-dynamic is more efficient on coarse-grained applications. Finally, fluidanimate, that is also a fine-grained application that consists of 128 500 tasks, also benefits from the task-based approach. For this benchmark, static and dynamic loop scheduling achieve similar performance; this is due to the limited parallelism per parallel region, as the loop-based implementation consists of multiple barriers between small parallel regions, fact that diminishes the effect of dynamic vs static scheduling.

3.5 CONCLUSIONS

In this extensive evaluation of highly parallel applications on an ARM big.LITTLE AMC system we showed that current implementations of parallel applications using pthreads are not ready to fully utilize an AMC. Implementing highly sophisticated parallelization strategies such as parallel pipelines (ferret) to exploit AMCs at the application level requires a significant programming effort and is not applicable to all workloads. The built-in GTS heterogeneity-aware OS scheduler only partially mitigates the slowdown of static threading when using both big and little cores. Both dynamically-scheduled loop- and task-based versions achieve higher performance with increased utilization which results in increased power. This leads to similar energy consumption as static threading and GTS, which ends up with better results in EDP.

Overall, GTS and static threading are not suitable solutions to run intensive multi-threaded applications on AMCs. Dynamic scheduling is essential to distribute the load across different core types. A loop-based implementation with dynamic scheduling is appropriate when the parallel work granularity is large and the potential imbalance at the tail of the loop is insignificant compared to the overall parallel region duration. A task-based implementation with inter-task dependencies allows removing barriers, which is the preferred

solution, especially when the granularity of parallel regions is small.

CHAPTER 4

TASK-BASED SCHEDULING SOLUTIONS

4.1 INTRODUCTION

The use of asymmetric multi-core architectures forms an appealing solution in high-performance computing to tackle the power wall. These architectures increase energy efficiency [23, 39, 41] by featuring different types of processing cores designed to target performance or power optimization.

To effectively utilize such systems taking into account their heterogeneity, load balancing and scheduling become two of the main challenges [57]. An approach towards these challenges is the use of task-based programming models [7, 18, 35, 37]. The modern task-based programming models schedule tasks dynamically according to the availability of resources. They also allow the specification of dependencies between tasks, enabling the runtime system to automatically perform scheduling and synchronization decisions.

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. Some previous works [31, 43, 60, 83] tackled this issue using static scheduling over the whole TDG to statically map tasks to processors on a heterogeneous system. However, they required the knowledge of profiling information and most of them were evaluated on synthetic

randomly-generated TDGs.

In this paper, we propose two novel dynamic task schedulers that detect the critical path of the in-flight dynamic snapshot of the TDG. Moreover, we make a study of the potential of the proposed dynamic scheduling techniques compared to existing dynamic heterogeneous schedulers [27, 83]. Specifically we compare our approaches to the the criticality aware task scheduler (CATS) [27] as well as a dynamic implementation of the heterogeneous earliest finish time scheduler (HEFT) [83]. We implement these scheduling policies in the OmpSs [8, 35] programming model that supports dynamic scheduling and dependency tracking.

Compared to previous works, all the scheduling policies described and evaluated in this paper are based on information discoverable at runtime, are implementable and work on a real asymmetric multi-core platform with real applications and therefore, using real TDGs. The contributions of this paper are the following:

- The Critical Path scheduler (CPATH) that dynamically assigns the tasks that belong to the critical path of the TDG to the fast cores of the system. To do so, CPATH tracks the execution time of the tasks, assigns cost-based priorities and, according to these priorities it detects the critical tasks.

- The Hybrid Criticality scheduler (HYBRID) that incorporates the features of CPATH and CATS [27] by assigning to the fast cores tasks that belong either to the critical path or to the longest path of the TDG, depending on the runtime circumstances. HYBRID uses mixed priorities that are cost-based or level-based. This technique also keeps track of the task costs but if this information is not available it uses the mechanisms of CATS that dynamically detects the longest dependency chain of the in-flight dynamic state of the TDG

- An evaluation of the proposed CPATH and HYBRID schedulers compared to the state of the art heterogeneous schedulers CATS [27] and HEFT [83], all of them implemented in the OmpSs programming model. Moreover we evaluate these approaches next to the default FIFO scheduler that serves as our baseline. The results show that all heterogeneous schedulers improve overall performance reaching up to 45% improvement. Furthermore, we describe their features such as the high per-task overheads of CPATH, the inability of dHEFT to improve performance when the task number increases as well as the benefit of HYBRID scheduler compared to CATS when task cost variability increases.

Table 4.1 shows the acronyms that we use in the next sections.

Table 4.1 Acronyms used in the paper

Acronym	Meaning
TDG	Task Dependency Graph
CATS	Criticality-Aware Task Scheduler
CPATH	Critical Path-Aware Scheduler
HEFT	Heterogeneous Earliest Finish Time
dHEFT	Dynamic Heterogeneous Earliest Finish Time
BF	Breadth-First
HYBRID	Hybrid Criticality-Aware Scheduler
FIFO	First-In First-Out
`plist`	Predecessors' List
`slist`	Successors' List
tt-is	Task Type - Input Size

4.2 BACKGROUND

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 [8, 35]. It enables the annotation of 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 [37]. OmpSs is built with the support of the Mercurium compiler, responsible for the translation of the OmpSs annotation clauses to source code, 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 discovered 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) [36] 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.

4.3 HETEROGENEOUS SCHEDULING

The efficient scheduling problem has been intensively studied for asymmetric systems. In this section we describe four scheduling approaches that target such systems. The first three are based on separating the tasks into groups of critical and non-critical tasks and assign each group to one core type: the critical tasks to the fast cores and the non-critical tasks to the slow cores. The difference between these three approaches is the way of considering a task critical. First is the Criticality-Aware scheduler (CATS)[27], which detects the critical tasks based on their *bottom level*. Secondly, the Critical Path scheduler (CPATH), proposed in this paper, that detects the critical path of the dynamic (TDG) with the help of *bottom cost* based priorities. The Hybrid Criticality scheduler (HYBRID), proposed in this paper, uses both bottom level and bottom cost based priorities. Last, we describe a dynamic implementation of HEFT scheduler (dHEFT) [83], that for every task it detects the processor that finishes its execution at the earliest possible time. All of the described schedulers operate at runtime on the dynamic snapshots of the TDG. CPATH, HYBRID and dHEFT perform on-line profiling of the task execution time without considering inter-task communication costs, given the uncertainty of data movement latency that hides in the cache hierarchy of an asymmetric multi-core system with prefetching.

4.3 CRITICALITY-AWARE TASK SCHEDULER

The Criticality-Aware Task Scheduling generally applies to task-based programming models supporting task dependencies, but for simplicity we explain it in the context of the OmpSs programming model. CATS uses bottom-level longest-path priorities and consists of three steps:

Task prioritization: when a task is created and added to the TDG, 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 according on the work stealing policy.

These steps are performed dynamically and potentially in parallel in different cores. Thus, 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 4.1 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 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 detail.

Task Prioritization

Each task in the TDG has a list to include its predecessors (*plist*). Every time an edge is added into the TDG on the creation of a new task, the corresponding predecessor of the dependency is added in the *plist* of its successor. For example, in Figure 4.1, when the dependency between tasks 2 and 5 occurs, the task number 2 is inserted into the *plist* of the

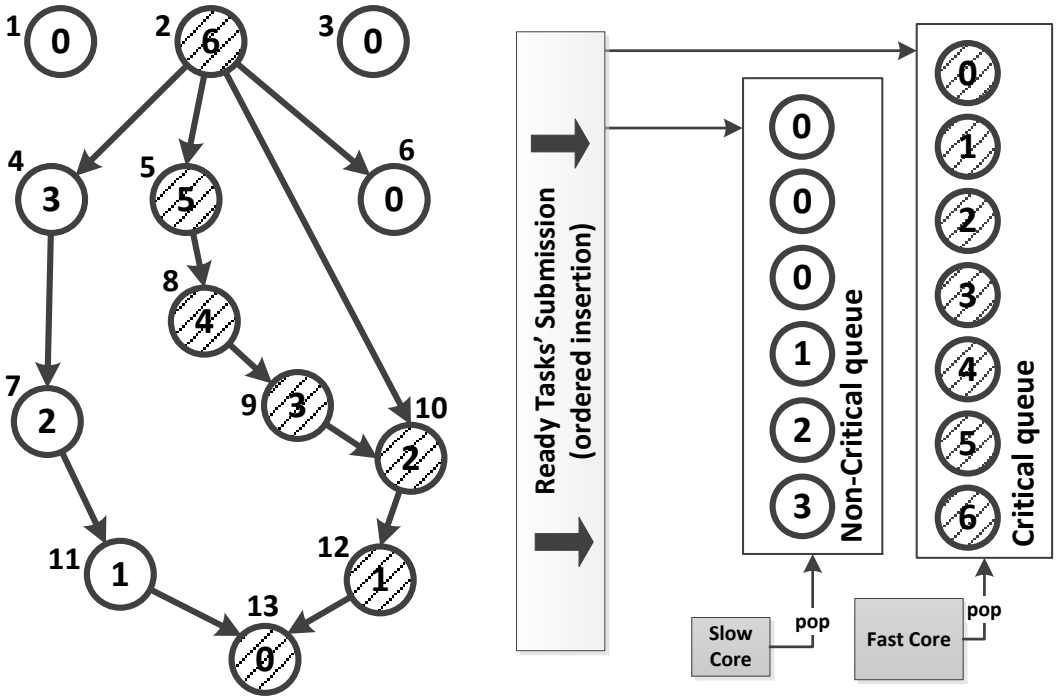


Figure 4.1 Task submission with CATS. Nodes are marked with the *bottom level* of each task. Pattern-filled nodes mark the critical tasks.

task number 5. Thus, the *plist* of task number 5 becomes $\{2\}$. Accordingly, the *plist* of task number 10 will be $\{2, 9\}$ when the edge 9→10 is inserted to the TDG.

The priority given to a task is the *bottom level* of the task. The *bottom level* is computed by traversing the TDG upwards starting from the successor that the currently created edge is pointing to. The priority of this successor is 0 because it is a leaf node of the graph, as it is the last created task. Then, using *plist* for each task, the algorithm navigates to the upper levels of the TDG and updates the priority on each visited node. This way not all the graph is updated, but only the tasks that are predecessors in the paths to the new edge. The algorithm also stops going up through a path, when it finds a priority larger than the one it would be updated to.

Listing 4.1 shows the algorithm for task prioritization. The complexity of this is $O(n^2)$, n being the number of tasks. This function is called on the creation of a new edge with the successor as argument. The algorithm traverses the *plist* of the successor task (line 5) and if the priority of the current predecessor is lower than the bottom level of the successor plus one, it updates the current predecessor's priority to that value (lines 7-8). If the updated predecessor task is ready (i.e., it sits in one of the ready queues), the scheduler reorders the ready queue so it remains ordered considering the updated priority (lines 9-10). Then, the same actions are performed recursively for each predecessor of the *plist* to update all the possible upward paths from the successor.

```
1 void prioritize_task(task *succ) {
2  int blev = succ->priority;
3  list plist = plistOf(succ);
4  task *currPred;
5  while( not isEmpty(plist) ) {
6    currPred = plist.next();
7    if(priorityOf(currPred) < blev+1) {
8     currPred->priority = blev+1;
9     if(isReady(currPred))
10     readyQueueOf(currPred)->reorder();
11    prioritize_task(currPred);
12   }
13 }
14}
```

Listing 4.1 Pseudo-code task prioritization with CATS.

The terminate conditions for the TDG navigation are two: *(a)* if the *plist* of the current task (`currPred`) is empty, so either we reach an entry node or the predecessors of the task have finished execution; or *(b)* if the priority of the current task (`currPred`) remains unchanged, which means that the successor task (`succ`) does not belong to the longest path because its predecessor already has a higher priority.

Task Submission

The purpose of this step is to divide the tasks into two groups: *critical* and *non-critical*. Critical tasks are tasks that belong to the longest path of the dynamic TDG, namely the path with the maximum number of tasks (or nodes). Thus, the longest path starts from the task with the maximum bottom level. At runtime, the longest path changes as tasks complete execution and new tasks are created. CATS manages to detect these changes and dynamically decide if the submitted task belongs to the longest path of the TDG.

When a task's dependencies are satisfied, the task becomes ready for execution and is to be inserted in the *ready queues*. Ready queues are priority queues that keep tasks in a decreasing order of task priorities, i.e., the task with the maximum priority resides on the head of the queue. Critical tasks are inserted in the critical queue and non-critical tasks in the non-critical queue. The pattern-filled nodes in Figure 4.1 represent the critical tasks in that graph.

To determine the criticality of a task, CATS keeps track of the last discovered critical task. Then, for each task that becomes ready, CATS checks the following conditions: *(a)* if the priority of the current ready task is higher or equal to the priority of the last discovered critical task and, *(b)* if the current ready task is the highest-priority immediate successor of the last discovered critical task.

The task that satisfies the second condition is a task with a lower priority than the max-

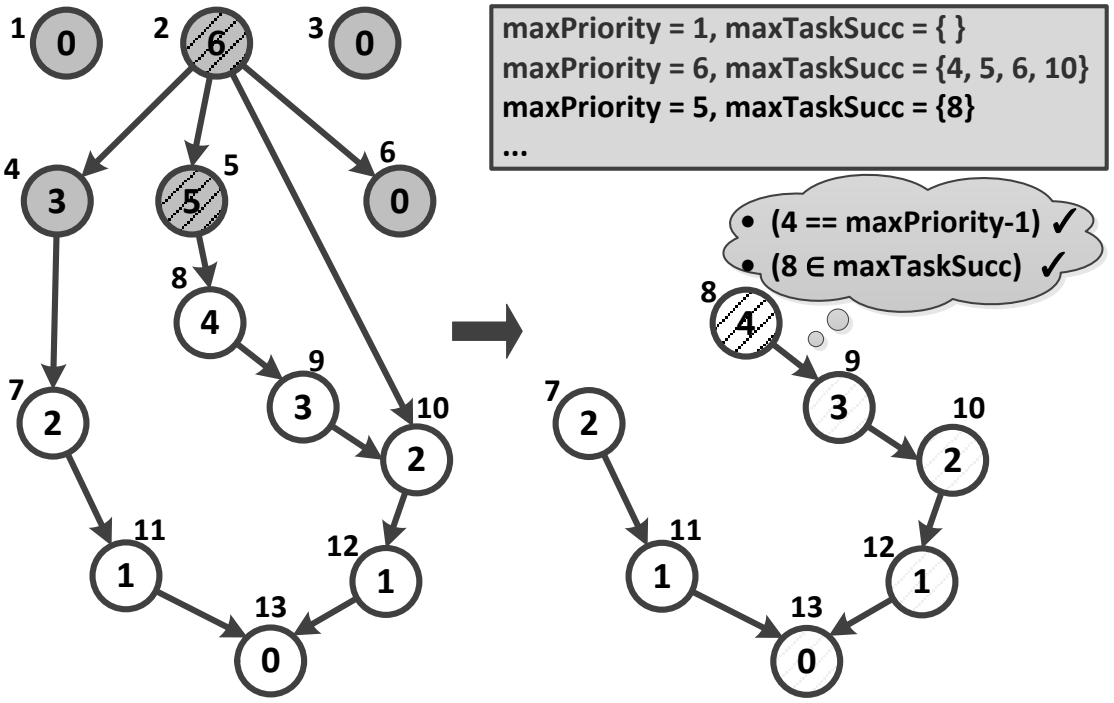


Figure 4.2 Task submission. Gray nodes indicate finished tasks and pattern-filled nodes indicate critical tasks.

imum but the task belongs to the longest path because it is the highest priority immediate successor of the last detected critical task.

Listing 4.2 shows a simplified version of the task submission code, that is of complexity *O(n)* (*n* is the number of tasks). The variable `maxPriority` (line 1) is used to store the priority of the last critical task, and `maxPriorityTask` (line 2) is used to store the last critical task. Initially, `maxPriority` is set to 1 and `maxPriorityTask` is set to `NULL`. This avoids the scheduling of independent tasks (i.e., tasks with zero priority) to fast processors at the start of the execution. On the first ready task, if its priority is higher or equal than 1 (line 5) , it is considered to be the first task of the longest path. Therefore, it is inserted in the critical queue and the variables `maxPriority` and `maxPriorityTask` are updated accordingly (lines 9-11) to determine correctly the criticality of the next submitted task.

If the priority of the submitted task is equal to `maxPriority - 1`, we check if it also belongs to the successors of the task with the maximum priority (lines 6-7) and therefore to the longest path. If these two conditions are met, the task is determined to be critical, it is inserted in the critical queue and, as before, the variables `maxPriority` and `maxPriorityTask` are updated (lines 9-11). In the rest of the cases the task is not considered critical and it is inserted in the non-critical queue.

Figure 4.2 shows an example of a TDG during task submission. The gray nodes in the graph are tasks that have finished execution and the pattern-filled nodes are critical tasks.

```
1 int maxPriority = 1;
2 task *maxPriorityTask = NULL;
3
4 void submit_task(task *t) {
5  if( t->priority >= maxPriority or
6     (t->priority == maxPriority-1 and
7      t $\in$ succListOf(maxPriorityTask)) )
8  { //the task is critical
9    critical_queue.push(t);
10   maxPriority = priorityOf(t);
11   maxPriorityTask = t;
12   return;
13 }
14 //the task is non-critical
15 non_critical_queue.push(t);
16}
```

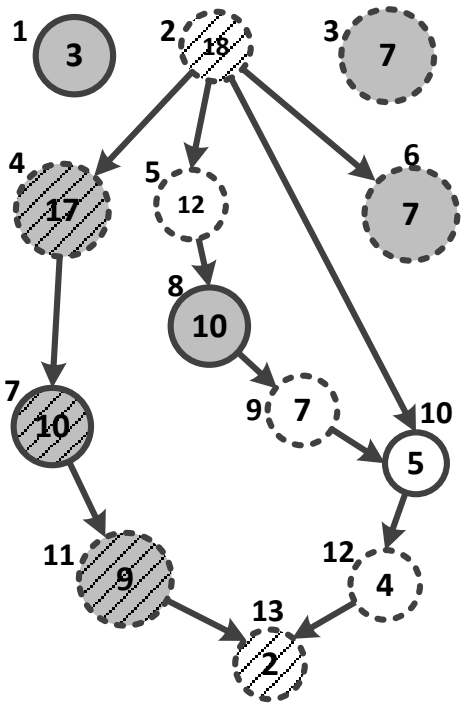
Listing 4.2 Pseudo-code for task submission with CATS.

The numbers inside the nodes indicate their priority and the numbers outside the nodes show the task id, which is assigned in task creation order. The variable `maxPriority` corresponds to the priority of the last critical task and the `maxTaskSucc` is the list of the successors of the last critical task, filled with the task ids of the successors. Initially, `maxPriority` is set to 1 and `maxTaskSucc` is empty. When task 2 is about to be submitted, it is inserted in the critical queue because its priority is higher than the maximum, which at the beginning is 1. Then, the value of `maxPriority` is set to 6 (priority of task 2), and the `maxTaskSucc` list is updated with the successors of task 2. At the point where all the gray tasks have finished execution, the values of `maxPriority` and `maxTaskSucc` are updated as shown in Figure 4.2. For every newly-ready task, the conditions listed above are evaluated. When task 7 is submitted, it is not considered as critical because it does not belong to the `maxTaskSucc` list and its priority is not equal to `maxPriority-1`. Contrarily, task 8 satisfies both conditions and so the task is inserted in the critical queue.

Task-to-Core Assignment

Task-to-core assignment takes place dynamically and in parallel to the previous steps and its time complexity is *O(n)*, *n* being the number of tasks. When a core becomes idle, it checks the corresponding ready queue (depending on the core type) to get a task to execute. Fast cores retrieve critical tasks from the critical queue, while slow cores retrieve non-critical tasks from the non-critical queue. Each ready queue is shared among the cores of the same type so there is no need for work stealing among cores of the same type.

If tasks in an application are imbalanced, i.e., the majority are non-critical and only a few tasks are critical, or vice versa, one of the types of processors would be overloaded and



	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

Figure 4.3 Priority assignment taking into account the task costs. Task costs are assumed known and are shown in the tables.

the other would starve for work. This can happen in applications with wide graphs and a large amount of tasks, where the ratio between critical tasks and the total amount of tasks may be small. To leverage the resources, the work-stealing mechanism for CATS lets fast cores steal work from slow cores whenever the critical queue becomes empty.

4.3 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. 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, introducing the cost of $O(2n^2)$, where n

is the number of tasks.

Task submission: when a task becomes *ready*, it is submitted to a *ready queue*. At 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 and complexity of *O(n)*.

Task-to-core assignment: this step is identical to CATS.

Task Prioritization

Each task of the TDG keeps a list with its successors (*slist*). This list is being built when an edge (dependency between two tasks) is added in the TDG. So when a task dependency occurs, the corresponding successor task is added in the *slist* of its predecessor. For example, on Figure 4.3, when the dependency between tasks 2 and 4 occurs, the *slist* of task number 2 becomes {4}. This goes on for all the added edges of the TDG, therefore when the edge 2→5 is inserted in the TDG, the task number 5 is inserted in the *slist* of task number 2; so the *slist* of task number 2 becomes {4, 5}.

The goal of this step is to assign priorities based on the *bottom cost* of the tasks of the TDG. 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. So the main difference between the *bottom level* and the *bottom cost* is the consideration of the estimated time.

Figure 4.3 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 4.3 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 and the numbers outside the nodes show their task ID.

The task prioritization step takes place every time a task finishes execution. CPATH uses a vector to store task costs and keeps one entry per tt-is. Because CPATH needs to discover the unbiased critical path of the TDG, it uses one of the core types as reference to track the task costs. In our experiments we chose to use as reference the fast cores since this way the learning phase (that is, the phase where CPATH discovers the task costs) becomes shorter. To avoid wrong task cost prediction of future tasks, CPATH ignores the first execution of each tt-is because usually it takes more time.

Listings 4.3 and 4.4 show how the critical path scheduler performs task prioritization.

```
1 void taskExit (task* finished) {
2  if( stateOf(finished) == init ) {
3    finished->state = in_progress;
4    return;
5  }
6  if( stateOf(finished) == in_progress ) {
7    timesSet[finished] = finished->execTime;
8    finished->state = tracked;
9  }
10 task* succ;
11 for( succ in finished->successors )
12   if( numPredecessorsOf(succ) == 1 ) {
13     lock();
14     if( succ $\notin$ entryNodes )
15       entryNodes->push(succ);
16       unlock();
17   }
18 list<task>* updatedList = new list<task>();
19 for( node in entryNodes)
20   updatePriorities(node, updatedList);
21 for( node in updatedList )
22    node->unsetUpdated();
23}
```

Listing 4.3 Pseudo-code for taskExit, the function called by the cores used as reference for tracking the task costs

Whenever a task finishes execution on one of the cores used as reference (here: fast cores) the runtime makes a call to the `taskExit` routine shown in Listing 4.3. At this point, the runtime is aware of the execution time of the finished task. This function has the responsibility to update the known task costs and also perform the prioritization of the tasks on the TDG. The prioritization is done by the `updatePriorities` function of Listing 4.4. This function is responsible for TDG traversal.

The `taskExit` function in Listing 4.3 takes as an argument the task that has just finished. In order to keep track of whether the execution time of the tt-is has been discovered we implement a small finite state machine within this stage. Every tt-is has three possible states. The initial state is the `init` state; this means that the specific tt-is has not yet been executed so its execution time is totally unknown. When a tt-is is executed for the first time its state changes from `init` to `in_progress`. This means that a task of this tt-is has been executed once, but CPATH ignores this cost because the first instance may not be representative due to cold start effects and one sample may not be enough history for prediction. While the tt-is of a node is in `init` or `in_progress` state its execution time is considered to be 1. After the second execution of a tt-is the state of it becomes `tracked` meaning that the execution time has been tracked and can be used for the computation of the priorities.

After the first checks of the tt-is state (lines 2-9 of Listing 4.3) the algorithm traverses

```
1 int updatePriorities (task* currT, list* updated) {
2   if( currT == NULL )  return 0;
3   if( isVisited(currT) )
4     return priorityOf(currT);
5   successors = currT->successors;
6   int maxSucc = -1;
7   bool succVisited = true;
8
9   for(succ in successors) {
10    int succPriority;
11    //Avoid double update
12    if( !isUpdated(succ) || !isVisited(succ) ) {
13      succPriority = updatePriorities(succ, updated);
14      succ->setUpdated();
15      updated->push(succ);
16    }
17    else
18      succPriority = priorityOf(succ);
19    if(succPriority > maxSucc)
20       maxSucc = succPriority;
21    succVisited = succVisited && isVisited(succ);
22  }
23  if( timeIsTracked(currT) ) {
24    currT->priority = (maxSucc + timesSet[currT]);
25    if(succVisited && groupOf(currT) < twDetected)
26      currT->setVisited();
27  }
28  else
29    currT->priority = maxSucc + 1;
30
31  return priorityOf(currT);
32}
```

Listing 4.4 Pseudo-code for task prioritization with CPATH

the *slist* of the finished task and searches for the successors that become ready by the end of the execution of this task. This is identified by the fact that the ready-to-be successors have one unique (remaining) predecessor (e.g. the just finished task). These successors are inserted in the `entryNodes` list (lines 11-16 of Listing 4.3). For each one of the entry nodes the `updatePriorities` function is called (line 19 of Listing 4.3); this performs a top to bottom traversal of the TDG and updates the priorities.

Due to the properties of the top-to-bottom TDG traversal, the algorithm has to make sure that every node is prioritized only once per `updatePriorities` call. This is controlled by checking the `updated` flag of each node of the TDG. To visualize this situation let us assume that task number 2 of the TDG on Figure 4.3 finishes. Then the `entryNodes` list contains three tasks that will start the update: {4, 5, 6}. The update that starts from task number 4 marks tasks 4, 7, 11 and 13 as updated. Then, during the update of task number 5, the algorithm knows that task 13 has already been prioritized during the same update so

there is no need to apply the algorithm at this node again. This example does not show too much optimization because in this case the update of only one node is saved, but in real applications this node could have numerous successors for whom the priority update would be a large overhead.

The raising of the `updated` flag is something temporal and is only used for helping the prioritization of a single update. There are cases when CPATH needs to raise a permanent flag in order to mark that the priority of the task will not change again in the future, e.g. it is the final priority. This happens when the execution times of all the tt-is that appear on the TDG have been discovered, for the tasks that their priorities are up to date. To mark these tasks CPATH uses the `visited` flag. If a task is `visited`, there is no need to get prioritized again. To clarify this, let us assume that in Figure 4.3 the task costs of the tt-is TaskA-Input2 and TaskB-Input2 are known. During the next prioritization, tasks 11 (TaskB-Input2), 12 (TaskA-Input2) and 13 (TaskA-Input2) in the TDG will be set as visited, because their priorities consist of the sum of known task execution times and they do not have any successors (with unknown execution times). So, an additional priority update in cases like this is redundant.

Listing 4.4 shows what happens during the the update of one entry node. The arguments of this function are `currT`, that is the entry node being updated, and `updated`, that is the list with the updated nodes. This list is being filled throughout the priority update in order to unset the updated flag later. The lines 2-4 of Listing 4.4 perform the checks that would cause the traversal to finish. If the node is not visited, then the algorithm traverses its successors. Note that, at this point, there is no check for updated flag, since tasks in the `entryNodes` are unlikely to be updated. Updated nodes can only be discovered through recursive calls and this check is performed later. If a successor is updated or visited, the priority update is skipped for the reasons explained above. Otherwise, the `updatePriorities` is called recursively for the current successor. This happens until we detect a node that is updated, visited or is a leaf node (node with no successors) of the TDG. When the algorithm reaches a node ready for update it calculates its priority by summing the highest priority of its successors to the execution time, if known, of the current node (lines 24, 29). Finally, the visited flag of the task is being updated.

There are three conditions that mark a task as visited: *(a)* if its execution time is known (line 23), *(b)* if **all** of its successors are visited (line 25) or *(c)* if we have encountered a taskwait (barrier) after the creation of this task (line 25). The last condition confirms that it is safe to mark this task as visited as there will be no future successors of this task on the current TDG. To track this information we use an atomic variable, `twDetected`, which is increased every time a taskwait is encountered. At creation time, each task is assigned a

group ID which is the value of the `twDetected` at that moment. If the group ID of a task is less than the current `twDetected` value then this means that a taskwait has occurred after the creation of this task.

Task Submission and Task-to-Core Assignment

The task submission is implemented using the same `critical` and `non-critical` ready queues as in CATS. Listing 4.2 can be used to describe the task submission of CPATH. The only modification needed is in the condition of the lines 6 and 7 of Listing 4.2. In addition to the `maxPriority`, CPATH keeps track of the `maxExecTime` which is the cost of the last discovered critical task. CPATH extends the condition of the critical task consideration by checking whether the priority of the current task is equal to `maxPriority - 1` or if it is equal to `maxPriority - maxExecTime`. Moreover, the value of `maxExecTime` is updated accordingly to the `maxPriority`.

Finally, task-to-core assignment is identical to CATS as described in Section 4.3.1. According to this, fast cores are responsible for the execution of the tasks in the critical queue and slow cores for the tasks in the non-critical queue.

4.3 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 4.3.1 and 4.3.2. 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 tt-is pair execution time on task creation, as some task type may have not been executed yet at the time all tasks are created.

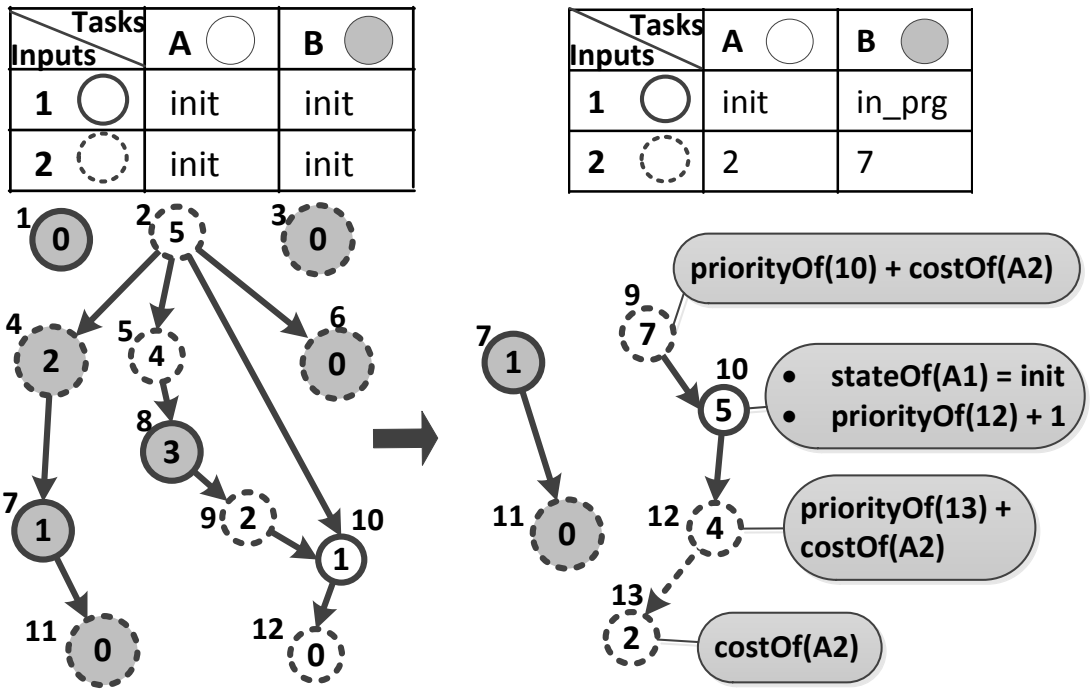


Figure 4.4 Priority assignment with HYBRID scheduler. Priority update when the edge between tasks 12 and 13 is created

HYBRID tracks task execution time on task completion and stores this information in a vector. This means that it also implements the `taskExit` function of CPATH that is called on task completion but, in the case of HYBRID, `taskExit` is only responsible of recording the execution time of the exiting task. This functionality is represented in lines 2-9 of Listing 4.3 and, after this code, the function returns. The priority assignment, taking place on task creation, remains similar to CATS [1] with the only difference that task cost is used for priority computation only if known and, otherwise, the cost is assigned to 1 and priority is increased according to CATS (lines 7 and 8 of Listing 4.1).

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.

Figure 4.4 shows an example of task prioritization with HYBRID. The tables show the

[1] All of the HYBRID scheduling steps have the same time complexity as CATS

state (or exec. time) of the tt-is pairs that appear on the TDG. Gray or white nodes indicate different task types (A or B respectively) and solid or dashed node outlines indicate task input size (1 or 2 respectively). The numbers inside the nodes show task priorities and the numbers outside the nodes show the task id.

On the leftmost TDG, the algorithm has no information about any of the tt-is costs. As the leftmost table shows, for all the possible tt-is the state is `init` meaning no task has been executed yet. Since the tasks of the TDG have been created, they have been prioritized using the CATS priority assignment method and the bottom level based priorities. On the rightmost TDG, tasks 1, 2, 3, 4, 5, 6 and 8 have been executed and a new task has appeared on the TDG: task number 13. When the edge 12→13 is created, tasks begin to be prioritized. Initially, the priority of the new task 13 is the cost of this task's tt-is, i.e., type A and input 2 (TaskA-Input2). Since there are no successors of this task, this becomes its initial priority. Then, the *plist* of task 13 is traversed and the priority of task 12 changes to priorityOf(13)+costOf(TaskA-Input2) since task 12 is corresponding to the TaskA-Input2 tt-is. Moving to the upper levels, task 10 is of tt-is TaskA-Input1 that is on the `init` state, thus unknown cost. This translates to the use of bottom level based prioritization so the priority of task 10 becomes priorityOf(12)+1. Finally, task 9 is prioritized using the cost of the TaskA-Input2 tt-is and the TDG navigation stops since there are no other predecessors.

4.3 Dynamic Heterogeneous Earliest Finish Time Scheduler

The Heterogeneous Earliest Finish Time (HEFT) algorithm [83] is a static scheduling approach for asymmetric systems. HEFT consists of two compile-time phases that use profiling information: the *task prioritizing phase* and the *processor selection phase*. In the first phase, the algorithm assigns priorities to the tasks based on their *upward rank*, that is, the length of the critical path from a given task to the exit task including task computation and communication costs [83]. When task prioritizing is done, the tasks are sorted according to their priorities. In the *processor selection phase* the algorithm searches for each task the appropriate processor to execute it. By keeping communication and computation costs, HEFT assigns each task to the processor that will finish its execution at the earliest possible time. Topcuoglu et al. [83] present their results based on evaluation on synthetic TDGs and assume known task execution and communication times at compile time. The scheduling is static, so all the decisions are taken before execution.

In this paper, since the evaluation consists of running real applications with unknown task costs, the best way to compare HEFT to our proposal is by using a dynamic version of HEFT algorithm (dHEFT). The dHEFT is implemented in the OmpSs programming model and is based on the implementation used in the evaluation of CATS [27]. This version

assumes two different types of cores (fast and slow) and keeps records of the task costs in each core. DHEFT discovers the task costs at runtime, computes the mean cost of each tt-is for each core type 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 a task becomes ready, dHEFT first inserts it in the ordered ready queue; then the task with the highest upward rank is selected and dHEFT checks if there are execution time records for this task. If the number of records is sufficient (we require a minimum of three records) then the estimated cost of the task is considered stable. Using that estimated execution time, the task is scheduled to the earliest executor by consulting the wlist of all 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. [2]

The initial dHEFT version presented in previous work [27] lacks the *task prioritizing phase* of the original HEFT algorithm. This paper, uses an improved version of dHEFT that adds this functionality by prioritizing tasks according to their *upward rank*. The implementation of this is similar to the CPATH prioritization step. When the prioritized tasks become ready, they are inserted in a sorted ready queue in decreasing order of their priorities. The algorithm then accesses the tasks in the order of their priorities to find the earliest executor for each of them.

4.4 EVALUATION

4.4 METHODOLOGY

We measure the execution time of five applications using CATS, CPATH, HYBRID, dHEFT and the default BF scheduler. The execution time corresponds to the average of 10 executions of the application on each machine set-up. Our test bed comprises a real big.LITTLE processor and a simulated heterogeneous system.

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 clocked at 1.6GHz and four Cortex-A7 cores at 800MHz.

[2]The time complexity of the task submission step is *O(nN)* and the task-to-core assignment is *O(n)*, where *n* is the number of tasks and *N* is the number of cores.

Table 4.2 Evaluated benchmarks and relevant characteristics

Application	Problem size	#Tasks	#Task types	Avg task exec. time (μs)	Per task ove	
					CATS	CP
Cholesky factorization	8×8 blocks of 1024×1024 floats	120	4	10 314 660	81.19	115
	16×16 blocks of 512×512 floats	816		1 551 322	104.76	238
	32×32 blocks of 512×512 floats	5984		1 551 322	104.76	238
QR factorization	16×16 blocks of 512×512 doubles	1 496	4	11 651 079	1 419.33	2 58
Heat diffusion	16×16 blocks of 512×512 doubles	5 124	3	93 198	145.17	748
Int. Histogram	8×8 blocks of 512×512 floats	2 048	2	514 096	217.45	62.
Bodytrack	native input (851MB)	408 525	6	41 869	93.90	120

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 remainder of the paper, we refer to Cortex-A15 cores as *big* and to Cortex-A7 cores as *little*.

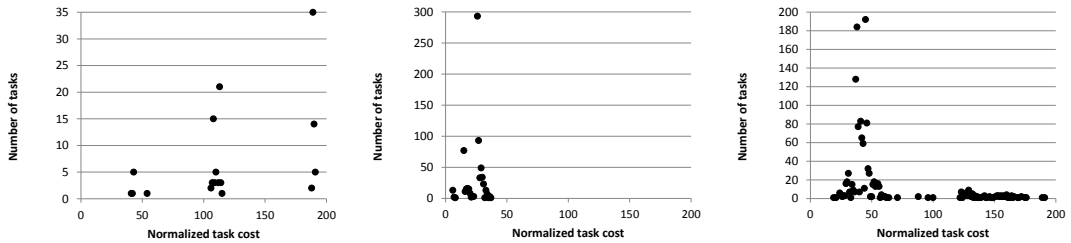
To evaluate heterogeneous scheduling on larger multi-core systems we use the heterogeneous multi-core TaskSim simulator [75]. TaskSim allows the specification of a heterogeneous system with two different types of cores: fast and slow. We can configure the amount of cores of each type and the difference in performance between the different types (performance ratio) in the TaskSim configuration file. In our experiments, we evaluate the effectiveness of the schedulers on 8 distinct heterogeneous machine configurations. These comprise systems with 16 or 32 total number of cores, and the number of fast cores ranging from 1 to 16. We set the performance ratio between fast and slow cores to 4.5× because this is the average performance ratio observed on the real machine for the benchmarks of this evaluation.

For both real and simulated platforms, each set-up has a given number of *total* and *big* cores. For all the scheduling approaches we present their speedup over the execution on one *little* core, shown in Equation 4.1.

$$\text{Speedup(total, big)} = \frac{\text{Exec. time}(1, 0)}{\text{Exec. time(total, big)}} \tag{4.1}$$

4.4 APPLICATIONS

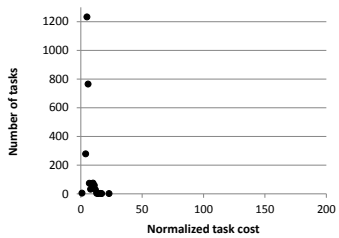
We use five scientific applications implemented in the OmpSs programming model: Cholesky factorization, QR factorization, Heat diffusion, Integral Histogram and Bodytrack. These



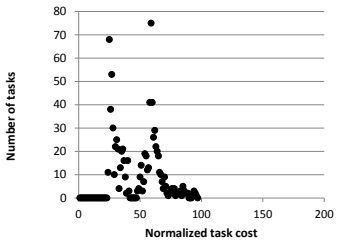
(a) Cholesky 8×8

(b) Cholesky 16×16

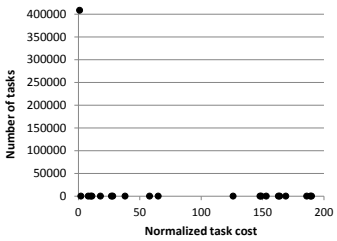
(c) QR factorization



(d) Heat diffusion



(e) Integral Histogram



(f) Bodytrack

Figure 4.5 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.

benchmarks are accessible in the BSC Application Repository [12] and in the PARSECSs library [25].

Kallia: Added in the background

Bodytrack is an application that tracks a marker-less human body using multiple cameras through an image sequence. The OmpSs version implements a two-stage parallel pipeline for the image processing. The two stages are synchronized through the OmpSs dataflow annotations. We use the native input of the benchmark suite [25].

Table 5.1 shows the different configurations and characteristics of the applications. The performance ratio between big and little cores depends on the application. For example, the difference between the issue rate and throughput of double-precision floating point units of both types of cores is larger than the difference for single-precision floating point instructions. Therefore, applications with heavy double-precision operation (e.g. QR) get a larger benefit from running on the big cores, than single-precision dominated applications (e.g integral histogram), as shown in Table 5.1.

The average per task overhead for each scheduler is negligible compared to the average task execution time shown in Table 5.1. Specifically, CATS has the lowest per task overheads. Next is HYBRID and the least efficient is CPATH. This is because of the complexity of the CPATH algorithm that takes place whenever the TDG needs to be updated. On the

other hand, CATS and HYBRID have negligible overheads caused by the task prioritization. For dHEFT, the search of the appropriate worker for a task becomes an obstacle in performance. Table 5.1 lacks the per task overheads of dHEFT because they appear to be too high due to the fact that the most intensive computations of dHEFT take place during the cores' idle time. Thus, the natural idle time of cores is also encountered as scheduling overhead and could not be separated, so it is unfair to present such results for comparison. Normally these obstacles in heterogeneous schedulers are paid off by the more effective task execution.

To more precisely characterise the benchmarks, we plot the task cost variability for each benchmark on Figure 4.5. 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 in the next section to classify how heterogeneous each application is and explain the behaviour of the heterogeneous schedulers that take into account the execution time.

4.4 Real Environment Evaluation

Figure 4.6 shows the speedup of CATS, CPATH, HYBRID, dHEFT and BF when running the applications on all eight cores of the Odroid-XU3. Cholesky and Integral Histogram operate on single-precision data, while QR and Heat Diffusion operate on double-precision. Double-precision applications get larger speedups over one little core because they benefit from a larger performance ratio when running on a big core. In the case of Bodytrack, the out-of-order processing power of the big cores helps on the efficient execution and creates a high performance ratio between big and little cores. For most of the cases, CATS scales better than the rest of the schedulers. The shortening of the critical path by running all critical tasks on big cores effectively reduces total execution time when running on all cores. CPATH scheduler does not achieve as high speedup as the other heterogeneous scheduling approaches but it still outperforms the baseline (BF) approach.

Figure 4.7 shows the average speedup obtained for each scheduler and machine setup. Overall, the heterogeneous schedulers outperform the platform-unaware BF scheduler. Specifically, CATS and HYBRID achieve a higher speedup by detecting critical tasks. We observe that their performance is approximately the same and this is due to the fact that HYBRID exploits the same CATS criticality in case the execution time of the task is not yet resolved. CPATH is less effective due to the additional overheads of the top-to-bottom TDG traversal. Since the evaluated dHEFT version is improved from previous studies [27], it shows better performance, although it still does not reach the efficiency of CATS and HYBRID because of its task criticality agnosticism.

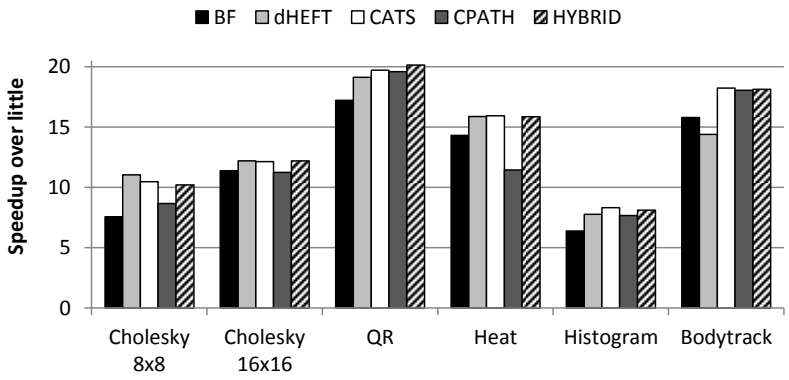


Figure 4.6 Speedup of CATS, CPATH, HYBRID, dHEFT and BF on 8 cores compared to the ideal

Moving in more detail, Figure 4.8 shows the speedup obtained for each application, scheduler and machine set-up. 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 4.5d. 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 shown on Table 5.1 and the homogeneity of the benchmark, CPATH scheduler cannot outperform 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. These tasks are considered non-critical, which 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

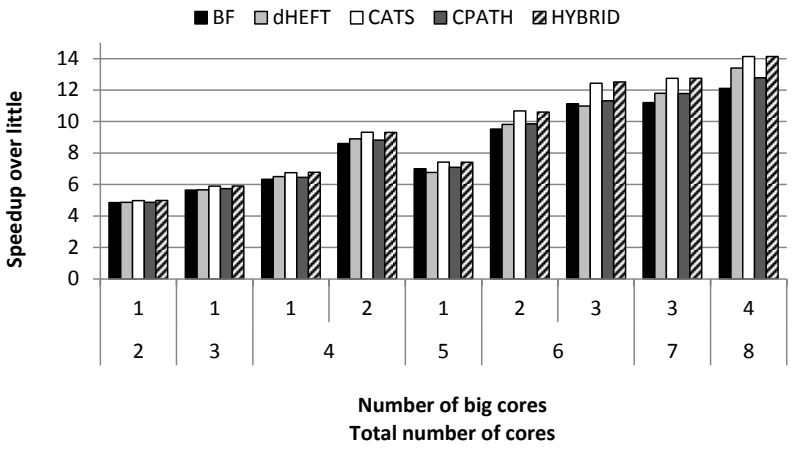


Figure 4.7 Average speedups obtained for each scheduler

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 4.5c. This is the reason why HYBRID gradually outperforms CATS as we increase the number of cores. With a small additional overhead, as Table 5.1 shows, HYBRID manages to 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

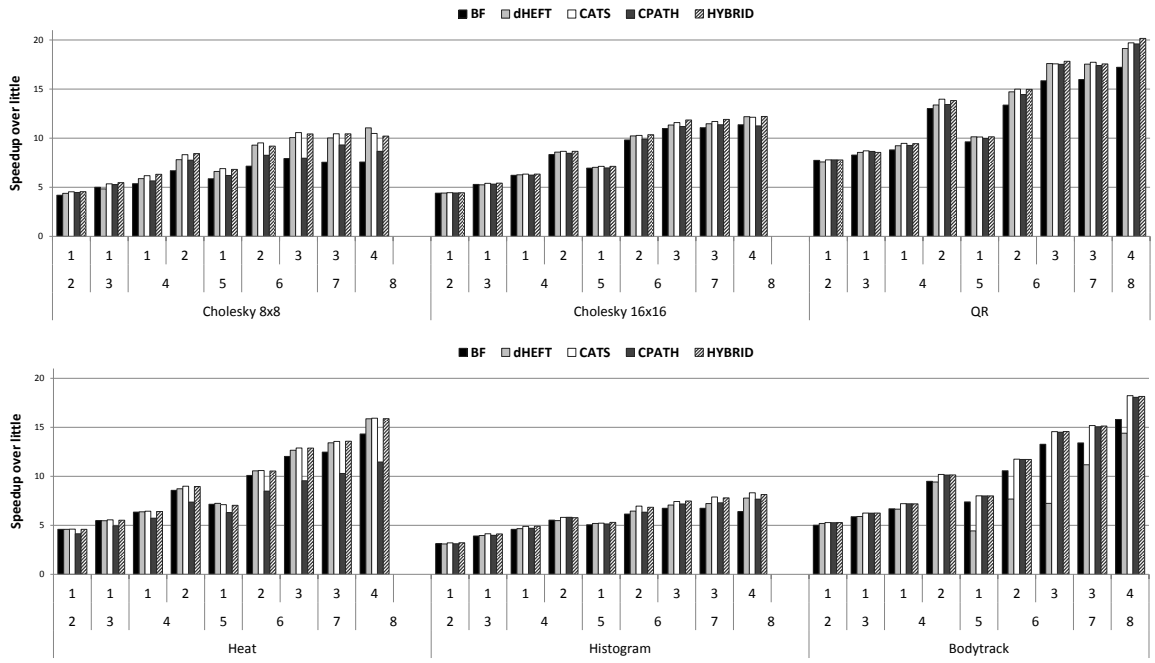
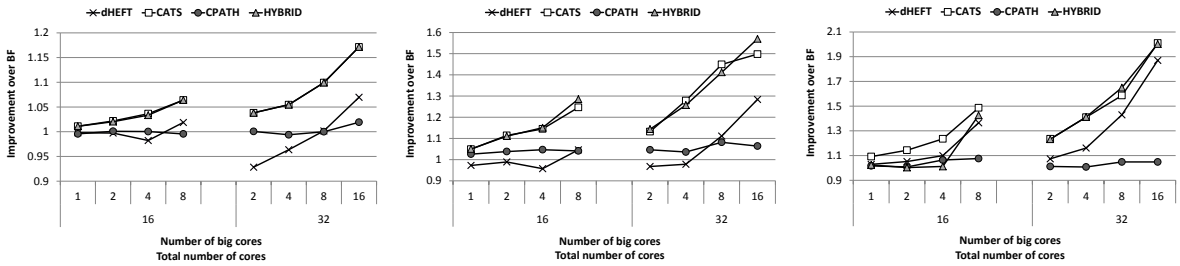


Figure 4.8 Speedups obtained for each scheduler and each application

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 tt-is. When the TDG has been completely created, and as soon as the cost of every tt-is 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 when it comes at a low cost.



(a) Cholesky 32×32 (b) QR (c) Heat diffusion

4.4 SIMULATIONS

To estimate the impact of the heterogeneity-aware schedulers on larger systems, we run three benchmarks using the TaskSim simulator [75]. The results contain a fixed scheduling overhead for all configurations, regardless of the dynamic overheads during execution (e.g., work stealing). We simulate Cholesky, QR and Heat diffusion. These applications feature different levels of task cost variability and have a proper amount of tasks so that the error introduced by the static overhead assumption remains negligible (e.g., bodytrack that creates 408 525 tasks should not be compared to a 5 000 task benchmark and static overhead). For Cholesky, we use an input matrix of 16384×16384 floats creating 512×512 blocks, which results in a 32×32 blocked matrix. This is because the other Cholesky configurations do not scale to 32 cores due to the limited task number. However, the task cost variability is similar to the 16× 16 input since the task size is not modified. Integral Histogram is excluded from the simulated evaluation because it does not scale beyond 16 cores.

Figures 4.9a, 4.9b and 4.9c show the improvement of dHEFT, CATS, HYBRID and CPATH over BF in systems with 16 and 32 cores for Cholesky, QR and heat respectively. In these experiments, the performance ratio between fast and slow cores is set to 4.5, which is the average performance ratio among the benchmarks. The heterogeneous schedulers utilize fast cores more effectively than BF, which results in larger improvements with higher number of fast cores.

Figure 4.9a shows the improvement of the schedulers over the baseline for Cholesky. The improvement for 16 cores is comparatively small. This is due to the increased problem size used in this experiment. This benchmark creates a small amount of critical tasks in the 32×32 input, which makes the workload less sensitive to critical tasks and limits the improvement of CATS and HYBRID to a maximum of 17%, while CPATH and dHEFT outperform BF by up to 10%.

Figure 4.9b shows that the best option for QR, the application with the highest task cost variability, on systems with 16 or 32 cores is the HYBRID scheduler, as was also shown in

the real platform evaluation, bringing improvements of 30 and 56%. CATS also performs well but CPATH falls short in detecting an appropriate amount of critical tasks which makes the little cores overloaded and the big cores waste their resources in work stealing.

For heat diffusion, Figure 4.9c shows that CATS achieves the best results outperforming BF by a factor of 2×. Moreover, HYBRID achieves similar results as it performs similar schedules as CATS. However, CPATH fails to achieve optimal results because it overloads the big cores during the learning phase while the little cores remain under-utilized.

4.5 Related Work

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

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, 31, 43, 58, 60, 83, 96] 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 7.0.3. The scheduler proposed in this paper is also a list scheduler.

Clustering schedulers [43, 45, 90, 93] 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 [45] 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 their cost, (i.e. tasks with the highest cost have the highest priority). The task-to processor assignment is done in decreasing order of priority.

Guided-random schedulers randomize their schedules by applying policies influenced by other sciences. Genetic algorithms [94] group tasks into generations and schedule them according to a randomized genetic technique. Chemical reaction algorithms [59, 92] mimic molecular interactions to map tasks to processors. Some of these guided-random approaches

are designed for heterogeneous systems [59, 94]. The scheduler by Page et al. [68] enables dynamic scheduling of multiple-sized tasks for heterogeneous systems, but it lacks support of inter-task dependencies.

Duplication-based schedulers [2, 11, 97] 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 to reduce communication costs. This scheduling may introduce redundant task duplications tasks which may lead to bad schedules. The Heterogeneous Economical Duplication scheduler [2] performs task duplication cautiously 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.

Criticality-Aware Schedulers: Several previous works propose scheduling heuristics that focus on the critical path of a TDG to reduce total execution time [31, 43, 60, 65, 83]. To identify the tasks on 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 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 core types.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [83] 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 [31]. 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 costs discovered in the processor to which it was assigned.

The Critical-Path-on-a-Processor (CPOP) algorithm [83] 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 TDG, so they do not apply to dynamically scheduled applications where only a part of the TDG is available at any given time, and (c) most of them use synthetic TDGs that are not necessarily representative of the dependencies in real workloads.

4.6 CONCLUSIONS

We introduced the first critical-path-aware dynamic scheduler for heterogeneous systems as well as the first hybrid criticality-aware scheduler. Like CATS and contrary to previous works on criticality-aware scheduling that use synthetic TDGs and require prior knowledge of profiling information, our proposals work on real platforms with real applications and do not require off-line profiling.

We implemented and evaluated our scheduling proposals in the runtime system of the OmpSs programming model. We showed that even if the accuracy of CPATH is higher in terms of task criticality identification, it does not always increase performance. Factors like the number of tasks and task cost variability play an important role on choosing the most appropriate scheduling policy and improve the performance of task-based applications. The implementations shown in this paper will be included in the next stable release of the OmpSs programming model. Furthermore, the described policies are expected to be applicable to other task-based programming models with support for task dependencies.

In conclusion, this paper shows the potential of different heterogeneous schedulers to speed up dependency-intensive applications and take advantage of the asymmetric compute resources. As future work, we aim to provide a single smart scheduler that dynamically adapts the most appropriate scheduling policy depending to the application's characteristics and availability of resources, with the possibility of tracking the task costs on all the core types to cover the case when a core type is not always faster, and potentially using off-line profiling to alleviate the overhead of task cost tracking at runtime. In addition, these schedulers could be extended to assume more than two core types. This can be done by applying multiple levels of criticality to the tasks, and assign each task to the corresponding core type depending on its performance.

CHAPTER 5

RUNTIME OVERHEADS MIGRATION

5.1 INTRODUCTION

Since the end of Dennard scaling [33] and the subsequent stagnation of CPU clock frequencies, computer architects and programmers rely on multi-core designs to achieve the desired performance levels. While multi-core architectures constitute a solution to the CPU clock stagnation problem, they bring important challenges both from the hardware and software perspectives. On the hardware side, multi-core architectures require sophisticated mechanisms in terms of coherence protocols, consistency models or deep memory hierarchies. Such requirements complicate the hardware design process. On the software side, multi-core designs significantly complicate the programming burden compared to their single-core predecessors. The different CPUs are exposed to the programmer, who has to make sure to use all of them efficiently, as well as using the memory hierarchy properly by exploiting both temporal and spatial locality. This increasing programming complexity, also known as the Programmability Wall [24], has motivated the advent of sophisticated programming paradigms and runtime system software to support them.

Task-based parallelism [8, 15, 17, 73] has been proposed as a solution to the Programmability Wall and, indeed, the most relevant shared memory programming standards, like OpenMP, support tasking constructs [67]. The task based model requires the programmer to split the code into several sequential pieces, called tasks, as well as explicitly specifying their input and output dependencies. The task-based execution model (or runtime system) consists of a master thread and several worker threads. The master thread goes over the code of the application and creates tasks once it encounters source code annotations identifying them. The runtime system manages the pool of all created tasks and schedules them

across the threads once their input dependencies are satisfied. To carry out the task management process, the parallel runtime system creates and maintains a Task Dependency Graph (TDG). In this graph nodes represent tasks and edges are dependencies between them. Once a new task is created, a new node is added to the TDG. The connectivity of this new node is defined by the data dependencies of the task it represents, which are explicitly specified in the application's source code. When the execution of a task finalizes, its corresponding node is removed from the TDG, as well as its data dependencies.

This task-based runtime system constitutes of a software layer that enables parallel programmers to decouple the parallel code from the underlying parallel architecture where it is supposed to run on. As long as the application can be decomposed into tasks, the task-based execution model is able to properly manage it across homogeneous many-core architectures or heterogeneous designs with different core types. A common practice in the high performance domain is to map a single thread per core, which enables the tasks running on that thread to fully use the core capacity. Finally, another important asset of task-based parallelism is the possibility of automatically managing executions on accelerators with different address spaces. Since the input and output dependencies of tasks are specified, the runtime system can automatically offload a task and its dependencies to an accelerator device (e.g., GPU) without the need for specific programmer intervention [20]. Additional optimizations in terms of software pre-fetching [69] or more efficient coherence protocols [62] can also be enabled by the task-based paradigm.

Despite their advantages, task-based programming models also induce computational costs. For example, the process of task creation requires the traversal of several indexed tables to update the status of the parallel run by adding the new dependencies the recently created tasks bring, which produces a certain overhead. Such overhead constitutes a significant burden, especially on architectures with several 10's or 100's of cores where tasks need to be created at a very fast rate to feed all of them. This paper proposes the Task Generation Express (TaskGenX) approach. Our proposal suggests that the software and hardware are designed to eliminate the most important bottlenecks of task-based parallelism without hurting their multiple advantages. This paper focuses on the software part of this proposal and draws the requirements of the hardware design to achieve significant results. In particular, this paper makes the following contributions beyond the state-of-the-art:

- A new parallel task-based runtime system that decouples the most costly routines from the other runtime activities and thus enables them to be off-loaded to specific-purpose helper cores.

- A detailed study of the requirements of a specific-purpose helper core able to acceler-

```
1       ...
2    //task_clause
3    memalloc(&task, args, size);
4    createTask(deps, task, parent, taskData);
5       ...
```

Listing 5.1 Compiler generated pseudo-code equivalence for task annotation.

```
1  void createTask(DepList dList, Task t,
2                  Task parent, Data args) {
3    initAndSetupTask(task1, parent, args);
4    insertToTDG(dList, task1);
5  }
```

Listing 5.2 Pseudo-code for task creation.

ate the most time consuming runtime system activities.

- A complete evaluation via trace-driven simulation considering 11 parallel OpenMP codes and 25 different system configurations, including homogeneous and heterogeneous systems. Our evaluation demonstrates how TaskGenX achieves average speedups of 3.1× when compared against currently use state-of-the-art approaches.

The rest of this document is organized as follows: Section 5.2 describes the task-based execution model and its main bottlenecks. Section 5.3 describes the new task-based runtime system this paper proposes as well as the specialized hardware that accelerates the most time-consuming runtime routines. Section 5.4 contains the experimental set-up of this paper. Section 5.5 describes the evaluation of TaskGenX via trace-driven simulation. Finally, Section 5.6 discusses related work and Section 5.7 concludes this work.

5.2 Background and Motivation

5.2 Task-based Programming Models

Task-based parallel programming models [8, 15, 17, 73], 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. One type of these annotations is the task annotations with dependency tracking which OpenMP [18] supports since its 4.0 release [67]. 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 well as the availability of resources.

When the compiler encounters a task annotation in the code, it transforms it to the pseudo-code shown in Listing 5.1. `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 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 5.1 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 5.2 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. The TDG is a distributed and dynamic graph structure that the runtime uses to keep the information about the current tasks of the application. The insertion of a task in the TDG 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. Listing 5.3 shows the pseudo-code for the TDG insertion. If for a task the `dList` is empty (line 2), this means that there are no memory addresses that need to be tracked during the execution; thus, the task is marked as *ready* by pushing it to the *ready queue* (line 3). Each entry of `dList` contains the actual memory address as well as the access type (read, write or read-write). The runtime

keeps a distributed unified dependency tracking structure, the `depMap` where it stores all the tracked memory addresses together with their writer and reader tasks. For each item in the `dList` the runtime checks if there is an existing representation inside the `depMap` (line 8). If the memory address of an entry of the `dList` is not represented in the `depMap`, it is being added as shown in line 9. If the address of a `dList` item exists in the `depMap`, this means that a prior task has already referred to this memory location, exhibiting a data dependency. According to the access type of `d`, the readers and the writers of the specific address are updated in the `depMap` (lines 10-15).

To reduce the lookup into the `depMap` calls, every time the contents of a memory address are modified, the tasks keep track of their *successors* as well as the number of *predecessors*. The *successors* of a task are all the tasks with inputs depending 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, as shown on line 20 of Listing 5.2.

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 goes to the *finished* state, the runtime has to perform some actions in order to prepare the successor tasks for execution. These actions are described in Listing 5.4. The runtime first updates the `depMap` to remove the possible references of the task as reader or writer (line 2). Then, if the task does not have any successors, it can safely be deleted (line 3). If the task has successors, the runtime traverses the successor list and for each successor task it decreases its predecessor counter (lines 5-6). If for a successor task its predecessor counter reaches zero, then this task becomes *ready* and it is inserted in the *ready queue* (lines 7-8). The runtime activity 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*. At this point the runtime prepares all the appropriate task and dependency tracking data structures as well as inserts the task into the TDG. The second change occurs when a task from being *created* it becomes *ready*; this implies that the input dependencies of this task are satisfied so the runtime schedules and inserts the task into the ready queue. The third change occurs when a running task finishes execution. In this case, following our task states, the task from being *ready* it becomes *finished*; this is followed by the runtime updating the dependency tracking data structures and scheduling possible successor tasks that become ready. For the rest of the paper we will refer to the first state change runtime activity as the task creation overheads (*Create*). For the runtime activity that takes place for the following two state changes (and includes scheduling and dependence analysis) we will use the term runtime overheads (*Runtime*).

```
1  void insertToTDG(DepList dList, Task t) {
2   if( dList is empty ) {
3     readyQ->push(t);
4     return;
5   }
6   Dependency entry;
7   for( d in dList ) {
8     entry = depMap[d.address()];
9     if(entry==NULL) depMap.add(entry, t);
10    if(d.accessType() == "write")
11      entry.addLastWriter(t);
12    if(d.accessType() == "read") {
13      entry.addReader(t);
14      entry.lastWriter()->addSuccessor(t);
15    }
16  }
17 }
```

Listing 5.3 Pseudo-code for TDG insertion

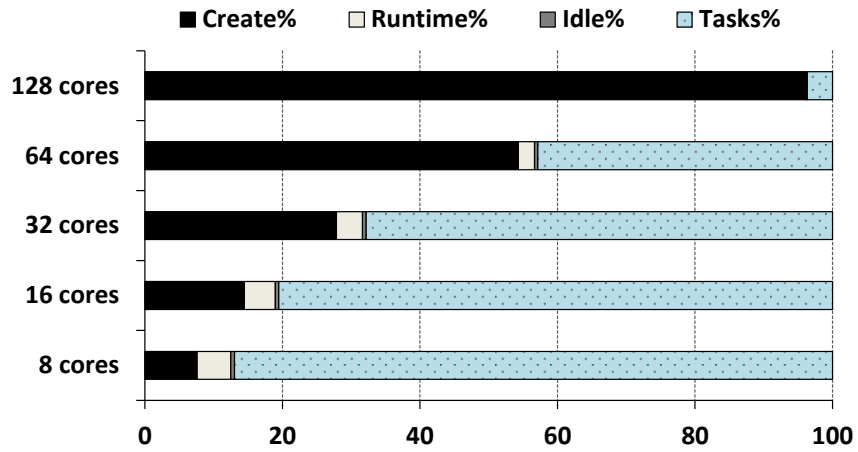


Figure 5.1 Master thread activity for Cholesky as we increase the number of cores.

5.2 MOTIVATION

Figure 5.1 shows the runtime activity of the master thread during the execution of the Cholesky[1] benchmark on 8, 16, 32, 64 and 128 cores[2]. The execution time represented here is the wall clock time during the parallel region of the benchmark. Each one of the series represents a different runtime overhead from the ones described above. The percentage of time spent on task creation is increasing as we increase the number of cores. This is be-

[1]Details about the benchmarks used are in Section 5.4
[2]The experimental set-up is explained in Section 5.4

```
1  void task_finish(Task *t) {
2    depMap.removeReaderWriter(t);
3    if(t->successors.empty()) delete t;
4    else {
5      for( succ in t->successors ) {
6        succ.decreasePredecessors();
7        if(succ.numPredecessors == 0)
8          readyQ->push(succ);
9      }
10   }
11 }
```

Listing 5.4 Pseudo-code for task_finish runtime activity.

cause the creation overhead is invariant of core count: the more we reduce the application's execution time by adding resources the more important this step becomes in terms of execution time. In contrast, the task execution time percentage is decreased as we increase the number of cores because the computational activity is being shared among more resources. One way to reduce the task creation overhead is by introducing nested parallelism. In this programming technique, every worker thread is able to generate tasks thus the task creation is spread among cores and its overhead is reduced. However, not all applications can be implemented with this parallelization technique and there are very few applications using this scheme. *Runtime* decreases as we increase the number of cores because this activity is also shared among the resources. This is because this part of the runtime takes place once the tasks finish execution and new tasks are being scheduled. So the more the resources, the less the runtime activity per thread, therefore less activity for the master thread.

Our motivation for this work is the bottleneck introduced by task creation as shown in Figure 5.1. Our runtime proposal decouples this piece of the runtime and accelerates it on a specialized hardware resulting in higher performance.

5.3 Task Generation Express

In this paper we propose a semi-centralized runtime system that dynamically separates the most computationally intensive parts of the runtime system and accelerates them on specialized hardware. To develop the TaskGenX we use the OpenMP programming model [18], [67]. The base of our implementation is the Nanos++ runtime system responsible for the parallel execution and it is used in this paper as a replacement of the entire OpenMP's default runtime.

Nanos++ [13] is a distributed runtime system that uses dynamic scheduling. As most

task-based programming models, Nanos++ consists of the master and the worker threads. The master thread is launching the parallel region and creates the tasks that have been defined by the programmer[3]. The scheduler of Nanos++ consists of a *ready queue* (*TaskQ*) that is shared for reading and writing among threads and is used to keep the tasks that are ready for execution. All threads have access to the *TaskQ* and once they become available they try to pop a task from the *TaskQ*. When a thread finishes a task, it performs all the essential steps described in Section 5.2.1 to keep the data dependency structures consistent. Moreover, it pushes the tasks that become ready to the *TaskQ*.

5.3 IMPLEMENTATION

TaskGenX relieves the master and worker threads from the intensive work of task creation by offloading it on the specialized hardware. Our runtime, apart from the master and the worker threads, introduces the Special Runtime Thread (SRT). When the runtime system starts, it creates the SRT and binds it to the task creation accelerator, keeping its thread identifier 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. Instead of querying the ready queue for tasks, the SRT looks for runtime activity requests in the Runtime Requests Queue (*RRQ*) and if there are requests, it executes them.

Figure 5.2a shows the communication infrastructure between threads within TaskGenX. 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 pending 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 executes them on the accelerator.

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 5.2 and described in Section 5.2.1. TaskGenX decouples the execution of `createTask` from the master thread. To do so, TaskGenX implements a wrapper function that is invoked instead of `createTask`. In this function, the runtime system checks if the SRT is enabled; if not then the default behaviour takes place, that is, to perform the creation of the task. If the SRT is enabled, a *Create* request is generated and inserted in the *RRQ*. The *Create* runtime request includes the appropriate info to execute the code described in Listing 5.2. That is, the dependence analysis data, the address of the allocated task, its parent and its arguments.

[3]Nanos++ also supports nested parallelism so any of the worker threads can potentially create tasks. However the majority of the existing parallel applications are not implemented using nested parallelism.

```
1 void SRTloop() {
2   while( true ) {
3     while(RRQ is not empty)
4       executeRequest( RRQ.pop() );
5     if( runtime.SRTstop() ) break;
6   }
7   return;
8 }
```

Listing 5.5 Pseudo-code for the SRT loop.

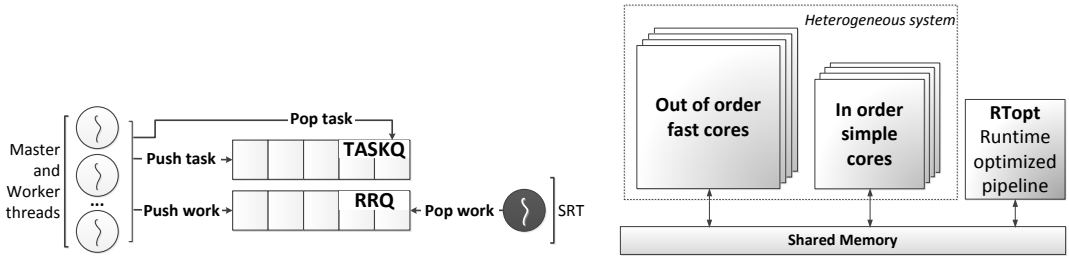
While the master and worker threads are executing tasks, the SRT is looking for *Create* requests in the *RRQ* to execute. Listing 5.5 shows the code that the SRT is executing until the end of the parallel execution. The special runtime thread continuously checks whether there are requests in the *RRQ* (line 3). If there is a pending creation request, the SRT calls the `executeRequest` (line 4), which extracts the appropriate task creation data from the creation request and performs the task creation by calling the `createTask` described in Listing 5.2. When the parallel region is over, the runtime system informs the SRT to stop execution. This is when the SRT exits and the execution finishes (line 5).

5.3 Hardware Requirements

As described in the previous section, TaskGenX assumes the existence of specialized hardware that accelerates the task creation step. The goal of this paper is not to propose a detailed micro-architecture of the specialized hardware; instead we sketch the high-level hardware requirements for the TaskGenX set-up, in the hope to be an insightful and useful influence for hardware designers. The SRT is bound to the task creation accelerator and executes the requests in the RRQ. Previous studies have proposed custom accelerators for the runtime activity [30, 38, 47, 55, 79, 81]. These proposals significantly accelerate (up to three orders of magnitude) different bottlenecks of the runtime system[4]. These special purpose designs can only execute runtime system activity.

As an alternative, in our envisioned architecture we propose to have a general purpose core that has been optimized to run the runtime system activity more efficiently. The runtime optimized (RTopt) core can be combined with both homogeneous or heterogeneous systems and accelerate the runtime activity. Figure 5.2b shows the envisioned architecture when RTopt is combined with an asymmetric heterogeneous system. This architecture has three core types that consist of simple in-order cores, fast out-of-order cores and an RTopt core for the SRT. RTopt can optimize its architecture, having a different cache hierarchy, pipeline

[4]Section 5.6 further describes these proposals.



(a) Communication mechanism between master/workers and SRT threads.

(b) SoC architecture including three types of cores: out of order, in-order and RTopt.

Figure 5.2

configuration and specialized hardware structures to hold and process the SRT. As a result, the RTopt executes the SRT much faster than the other cores. The RTopt can also execute tasks, but will achieve limited performance compared to the other cores as its hardware structures have been optimized for a specific software (the SRT).

To evaluate our approach we study the requirements of the RTopt in order to provide enough performance for TaskGenX. Based on the analysis by Etsion et al. [38], there is a certain *task decode rate* that leads to optimal utilization of the multi-core system. This rule can be applied in the case of TaskGenX for the *task creation rate*, i.e., the frequency of task generation of the runtime system. If the *task creation rate* is higher than the *task execution rate*, then for a highly parallel application the resources will always have tasks to execute and they will not remain idle. To achieve a high *task creation rate*, we can accelerate the task creation cost. Equation 5.1 shows the maximum optimal task creation cost, $C_{opt}(x)$ in order to keep x cores busy, without starving due to task creation.

$$C_{opt}(x) = avg.\ task\ duration/x \tag{5.1}$$

If C_{gp} is the cost of task creation when it is performed on a general purpose core, then the RTopt has to achieve a speedup of $r = C_{gp}/C_{opt}(x)$ to achieve full utilization of the system. Section **??** performs an analysis based on these requirements for the evaluated applications. As we will see in Section **??**, a modest and implementable value of $r = 16\times$ is enough to significantly accelerate execution on a 512-core system.

Finally, if TaskGenX executes on a regular processor without the RTopt core, the SRT is bound to a regular core without any further modification. In this scenario, applications will not significantly benefit from having a separate SRT.

Table 5.1 Evaluated benchmarks and relevant characteristics

Application	Problem size	#Tasks	Avg task CPU cycles (thousands)	Per task overheads (CPU cycles)			Measured perf. ratio	r	Parallel model
				Create	All	Deps + Sched			
Cholesky factorization	32K 256 32K 128	357 762 2829058	753 110	15221 17992	73286 58820	58065 40828	3.5	10.34 83.74	dependencies
QR factorization	16K 512 16K 128	11 442 707 265	518 570 3 558	17595 21642	63008 60777	45413 39135	6.8	0.01 3.11	dependencies
Blackscholes	native	488 202	348	29141	85438	56297	2.3	42.87	data-parallel
Bodytrack	native	329 123	383	9 505	18979	9474	4.2	12.70	pipeline
Canneal	native	3 072 002	67	25781	50094	24313	2.0	197.01	unstructured
Dedup	native	20 248	1 532	1294	9647	8353	2.7	0.43	pipeline
Ferret	native×2	84 002	29 088	38913	98457	59544	3.6	0.68	pipeline
Fluidanimate	native	128 502	16 734	30210	94079	64079	3.3	0.91	data-parallel
Streamcluster	native	3 184 654	161	6892	13693	6801	3.5	21.91	data-parallel

5.4 Experimental Methodology

5.4 Applications

Table 5.1 shows the evaluated applications, the input sizes used, and their characteristics. All applications are implemented using the OpenMP programming model [67]. We obtain Cholesky and QR from the BAR repository [12] and we use the implementations of the rest of the benchmarks from the PARSECSs suite [25]. More information about these applications can be found in [25] and [27]. As the number of cores in SoCs is increasing, so does the need of available task parallelism [77]. We choose the input sizes of the applications so that they create enough fine-grained tasks to feed up to 512 cores. The number of tasks per application and input as well as the average per-task CPU cycles can be found on Table 5.1.

5.4 Simulation

To evaluate TaskGenX we make use of the TaskSim trace-driven simulator [40, 75] as explained in Section 2.4.

We evaluate both symmetric and asymmetric systes with the number of cores varying from 8 to 512. To set the correct performance ratio between big and little cores for the asymetric systems, we measure the sequential execution time of each application on a real ARM big.LITTLE platform when running on a little and on a big core. We use the Hardkernel Odroid XU3 board that includes a Samsung Exynos 5422 chip with ARM big.LITTLE. The big cores run at 1.6GHz and the little cores at 800MHz. Table 5.1 shows the measured performance ratio for each case. The average performance ratio among our 11 workloads

is 3.8. Thus in the specification of the asymmetric systems we use as performance ratio the value 4.

We modify TaskSim so that it features one extra hardware accelerator (per multi-core) responsible for the fast task creation (the RTopt). Apart from the task duration time, our modified simulator tracks the duration of the runtime overheads. These overheads include: (a) task creation, (b) dependencies resolution, and (c) scheduling. The RTopt core is optimized to execute task creation faster than the general purpose cores; to determine how much faster a task creation job is executed we use the analysis performed in Section 5.3.2.

Using Equation 5.1, we compute the $C_{opt}(x)$ for each application according to their average task CPU cycles from Table 5.1 for $x = 512$ cores. C_{gp} is the cost of task creation when it is performed on a general purpose core, namely the *Create* column shown on Table 5.1. To have optimal results for each application on systems up to 512 cores, C_{gp} needs to be reduced to $C_{opt}(512)$. Thus the specialized hardware accelerator needs to perform task creation with a ratio $r = C_{gp}/C_{opt}(512)\times$ faster than a general purpose core.

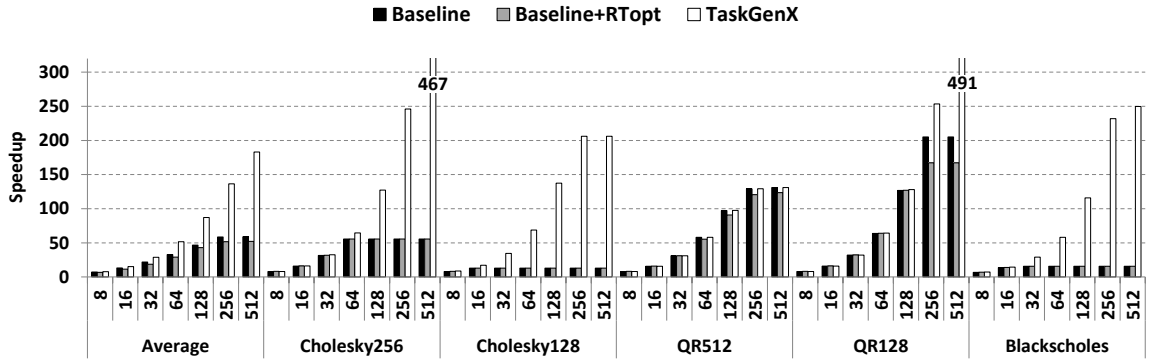
We compute r for each application shown on Table 5.1. We observe that for the applications with a large number of per-task CPU cycles and relatively small *Create* cycles (QR512, Dedup, Ferret, Fluidanimate), r is very close to zero, meaning that the task creation cost (C_{gp}) is already small enough for optimal task creation without the need of a faster hardware accelerator. For the rest of the applications, more powerful hardware is needed. For these applications r ranges from $3\times$ to $197\times$. Comparing r to the measured performance ratio of each application we can see that in most cases accelerating the task creation on a big core would not be sufficient for achieving higher task creation rate. In our experimental evaluation we accelerate task creation in the RTopt and we use the ratio of $16\times$ which is a relatively small value within this range that we consider realistic to implement in hardware. The results obtained show the average results among three different traces for each application-input.

5.5 EVALUATION

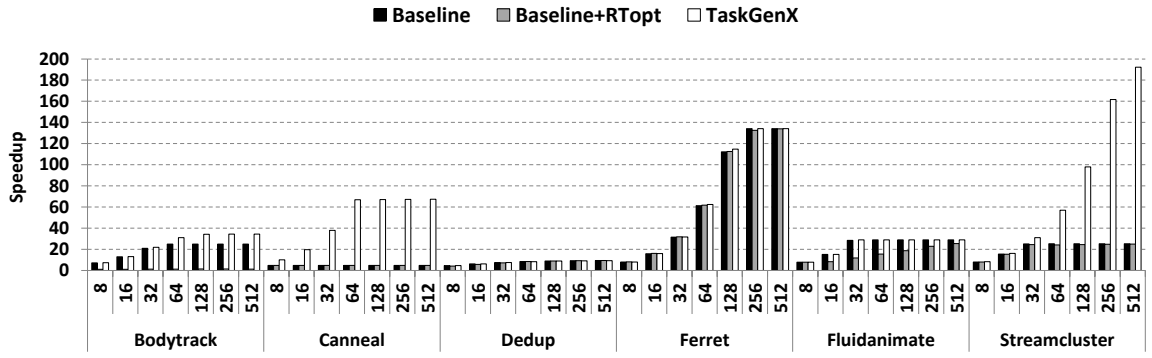
5.5 HOMOGENEOUS MULTICORE SYSTEMS

Figures 5.3a and 5.3b show the speedup over one core of three different scenarios:

- *Baseline*: the Nanos++ runtime system, which is the default runtime without using any external hardware support
- *Baseline+RTopt*: the Nanos++ runtime system that uses the external hardware as if it is a general purpose core



(a)



(b)

Figure 5.3 Speedup of TaskGenX compared to the speedup of Baseline and Baseline+RTopt for each application for systems with 8 up to 512 cores. The average results of (a) show the average among all workloads shown on (a) and (b)

- *TaskGenX*: our proposed runtime system that takes advantage of the optimized hardware

We evaluate these approaches with the TaskSim simulator for systems of 8 up to 512 cores. In the case of Baseline+RTopt the specialized hardware acts as a slow general purpose core that is additional to the number of cores shown on the x axis. If this core executes a task creation job, it executes it 16× faster, but as it is specialized for this, we assume that when a task is executed on this core it is executed 4× slower than in a general purpose core. The runtime system in this case does not include our modifications that automatically decouple the task creation step for each task. The comparison against the Baseline+RTopt is used only to show that the baseline runtime is not capable of effectively utilizing the accelerator. In most of the cases having this additional hardware without the appropriate runtime support results in slowdown as the tasks are being executed slower on the special hardware.

Focusing on the average results first, we can observe that TaskGenX constantly improves the baseline and the improvement is increasing as the number of cores is increased, reaching up to 3.1× improved performance on 512 cores. This is because as we increase the number of cores, the task creation overhead becomes more critical part of the execution time and affects performance even more. So, this becomes the main bottleneck due to which the performance of many applications saturates. TaskGenX overcomes it by automatically detecting and moving task creation on the specialized hardware.

Looking in more detail, we can see that for all applications the baseline has a saturation point in speedup. For example Cholesky256 saturates on 64 cores, while QR512 on 256 cores. In most cases this saturation in performance comes due to the sequential task creation that is taking place for an important percentage of the execution time (as shown in Figure 5.1). TaskGenX solves this as it efficiently decouples the task creation code and accelerates it leading to higher speedups.

TaskGenX is effective as it either improves performance or it performs as fast as the baseline (there are no slowdowns). The applications that do not benefit (QR512, Ferret, Fluidanimate) are the ones with the highest average per task CPU cycles as shown on Table 5.1. Dedup also does not benefit as the per task creation cycles are very low compared to its average task size. Even if these applications consist of many tasks, the task creation overhead is considered negligible compared to the task cost, so accelerating it does not help much.

This can be verified by the results shown for QR128 workload. In this case, we use the same input size as QR512 (which is 16K) but we modify the block size, which results in more and smaller tasks. This not only increases the speedup of the baseline, but also shows even higher speedup when running with TaskGenX reaching very close to the ideal speedup and improving the baseline by 2.3×. Modifying the block size for Cholesky, shows the same effect in terms of TaskGenX over baseline improvement. However, for this application, using the bigger block size of 256 is more efficient as a whole. Nevertheless, TaskGenX improves the cases that performance saturates and reaches up to 8.5× improvement for the 256 block-size, and up to 16× for the 128 block-size.

Blackscholes and Canneal, are applications with very high task creation overheads compared to the task size as shown on Table 5.1. This makes them very sensitive to performance degradation due to task creation. As a result their performance saturates even with limited core counts of 8 or 16 cores. These are the ideal cases for using TaskGenX as such bottlenecks are eliminated and performance is improved by 15.9× and 13.9× respectively. However, for Canneal for which the task creation lasts a bit less than half of the task execution time, accelerating it by 16 times is not enough and soon performance saturates at 64

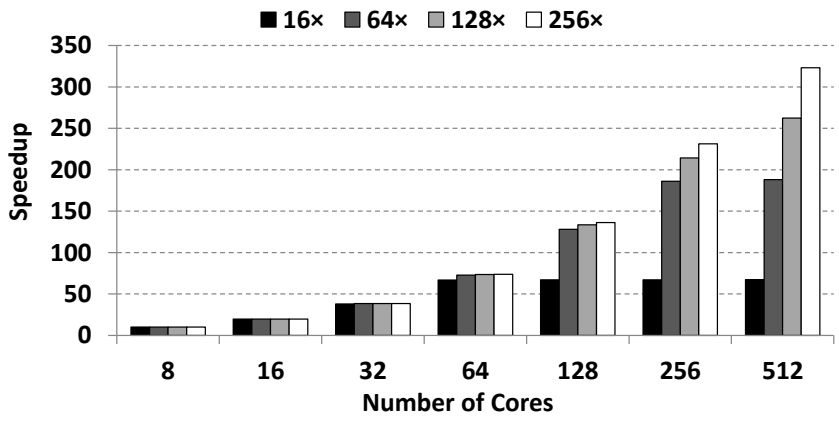


Figure 5.4 Canneal performance as we modify *r*; *x*-axis shows the number of cores.

cores. In this case, a more powerful hardware would improve things even more. Figure 5.4 shows how the performance of Canneal is affected when modifying the task creation performance ratio, *r* between the specialized hardware and general purpose. Using hardware that performs task creation close to 256× faster than the general purpose core leads to higher improvements.

Streamcluster has also relatively high task creation overhead compared to the average task cost so improvements are increased as the number of cores is increasing. TaskGenX reaches up to 7.6× improvement in this case.

The performance of Bodytrack saturates on 64 cores for the baseline. However, it does not approach the ideal speedup as its pipelined parallelization technique introduces significant task dependencies that limit parallelism. TaskGenX still improves the baseline by up to 37%. This improvement is low compared to other benchmarks, firstly because of the nature of the application and secondly because Bodytrack introduces nested parallelism. With nested parallelism task creation is being spread among cores so it is not becoming a sequential overhead as happens in most of the cases. Thus, in this case task creation is not as critical to achieve better results.

5.5 Heterogeneous Multicore Systems

At this stage of the evaluation our system supports two types of general purpose processors, simulating an asymmetric multi-core processor. The asymmetric system is influenced by the ARM big.LITTLE architecture [46] that consists of big and little cores. In our simulations, we consider that the big cores are four times faster than the little cores of the system. This is based on the average measured performance ratio, shown on Table 5.1, among the 11 workloads used in this evaluation.

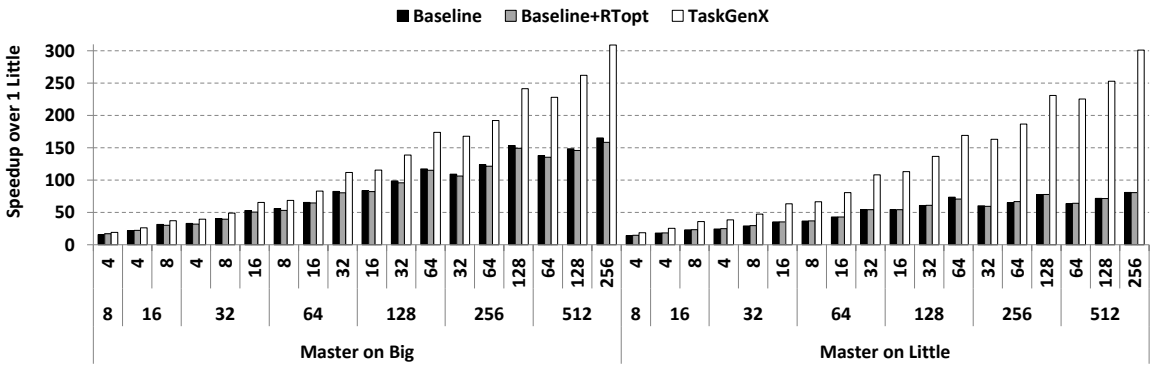


Figure 5.5 Average speedup among all 11 workloads on heterogeneous simulated systems. The numbers at the bottom of x axis show the total number of cores and the numbers above them show the number of big cores. Results are separated depending on the type of core that executes the master thread: a big or little core.

In this set-up there are two different ways of executing a task-based application. The first way is to start the application's execution on a big core of the system and the second way is to start the execution on a little core of the system. If we use a big core to load the application, then this implies that the master thread of the runtime system (the thread that performs the task creation when running with the baseline) runs on a fast core, thus tasks are created faster than when using a slow core as a starting point. We evaluate both approaches and compare the results of the baseline runtime and TaskGenX.

Figure 5.5 plots the average speedup over one little core obtained among all 11 workloads for the Baseline, Baseline+RTopt and TaskGenX. The chart shows two categories of results on the x axis, separating the cases of the master thread's execution. The numbers at the bottom of x axis show the total number of cores and the numbers above show the number of big cores.

The results show that moving the master thread from a big to a little core degrades performance of the baseline. This is because the task creation becomes even slower so the rest of the cores spend more idle time waiting for the tasks to become ready. TaskGenX improves performance in both cases. Specifically when master runs on big, the average improvement of TaskGenX reaches 86%. When the master thread runs on a little core, TaskGenX improves performance by up to 3.7×. This is mainly due to the slowdown caused by the migration of master thread on a little core. Using TaskGenX on asymmetric systems achieves approximately similar performance regardless of the type of core that the master thread is running. This makes our proposal more portable for asymmetric systems as the programmer does not have to be concerned about the type of core that the master thread

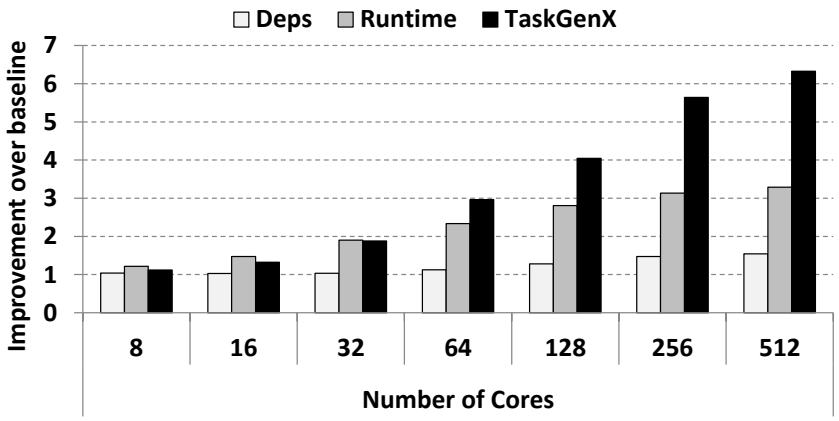


Figure 5.6 Average improvement over baseline; *x*-axis shows the number of cores.

migrates.

5.5 Comparison to Other Approaches

As we saw earlier, TaskGenX improves the baseline scheduler by up to 6.3× for 512 cores. In this section we compare TaskGenX with other approaches. To do so, we consider the proposals of Carbon [55], Task Superscalar [38], Picos++ [81] and Nexus# [30]. We group these proposals based on the part of the runtime activity they are offloading from the CPU. Carbon and Task Superscalar are runtime-driven meaning that they both accelerate all the runtime and scheduling parts. The task creation, dependence analysis as well as the scheduling, namely the ready queue manipulation, are transferred to the RTopt with these approaches. These overheads are represented on Table 5.1 under ALL. For the evaluation of these approaches one RTopt is used optimized to accelerate all the runtime activities. The second group of related designs that we compare against is the dependencies-driven, which includes approaches like Picos++ and Nexus#. These approaches aim to accelerate only the dependence analysis part of the runtime as well as the scheduling that occurs when a dependency is satisfied. The RTopt in this case is optimized to accelerate these activities. For example, when a task finishes execution, and it has produced input for another task, the dependency tracking mechanism is updating the appropriate counters of the reader task and if the task becomes ready, the task is inserted in the ready queue. The insertion into the ready queue is the scheduling that occurs with the dependence analysis. These overheads are represented on Table 5.1 under *Deps+Sched*.

Figure 5.6 shows the average improvement in performance for each core count over the performance of the baseline scheduler on the same core count. *Runtime* represents the runtime driven approaches and the *Deps* represents the dependencies driven approaches as

described above. X-axis shows the number of general purpose cores; for every core count one additional RTopt core is used.

Accelerating the scheduling with *Runtime*-driven is as efficient as TaskGenX for a limited number of cores, up to 32. This is because they both accelerate task creation which is an important bottleneck. *Deps*-driven approaches on the other hand are not as efficient since in this case the task creation step takes place on the master thread.

Increasing the number of cores, we observe that the improvement of the *Runtime*-driven over the baseline is reduced and stabilized close to 3.2× while TaskGenX continues to speedup the execution. Transferring all parts of the runtime to RTopt with the *Runtime*-driven approaches, leads to the serialization of the runtime. Therefore, all scheduling operations (such as enqueue, dequeue of tasks, dependence analysis etc) that typically occur in parallel during runtime are executed sequentially on the RTopt. Even if RTopt executes these operations faster than a general purpose core, serializing them potentially creates a bottleneck as we increase the number of cores. TaskGenX does not transfer other runtime activities than the task creation, so it allows scheduling and dependence analysis operations to be performed in a distributed manner.

Deps driven approaches go through the same issue of the serialization of the dependency tracking and the scheduling that occurs at the dependence analysis stage. The reason for the limited performance of *Deps* compared to *Runtime* is that *Deps* does not accelerate any part of the task creation. Improvement over the baseline is still significant as performance with *Deps* is improved by up to 1.5×.

TaskGenX is the most efficient software-hardware co-design approach when it comes to highly parallel applications. On average, it improves the baseline by up to 3.1× for homogeneous systems and up to 3.7× for heterogeneous systems. Compared to other state of the art approaches, TaskGenX is more effective on a large number of cores showing higher performance by 54% over *Runtime* driven approaches and by 70% over *Deps* driven approaches.

5.6 Related Work

Our approach is a new task-based runtime system design that enables the acceleration of task creation to overcome important bottlenecks in performance. Task-based runtime systems have intensively been studied. State of the art task-based runtime systems include the OpenMP [18], OmpSs [35], StarPU [7] and Swan [88]. All these models support tasks and maintain a TDG specifying the inter-task dependencies. This means that the runtime system is responsible for the task creation, the dependence analysis as well as the scheduling of the

tasks. However, none of these runtime systems offers automatic offloading of task creation.

The fact that task-based programming models are so widely spread makes approaches like ours very important and also gives importance to studies that focus on adding hardware support to boost performance of task-based runtime systems. Even if their work focuses more on the hardware part of the design, their contributions are very relative to our study as we can distinguish which parts of the hardware is more beneficial to be accelerated.

Carbon [55] accelerates the scheduling of tasks by implementing hardware ready queues. Carbon maintains one hardware queue per core and accelerates all possible scheduling overheads by using these queues. Nexus# [30] is also a distributed hardware accelerator capable of executing the *in*, *out*, *inout*, *taskwait* and *taskwait on* pragmas, namely the task dependencies. Unlike Carbon and Nexus, TaskGenX accelerates only task creation. Moreover, ADM [77] is another distributed approach that proposes hardware support for the inter-thread communication to avoid going through the memory hierarchy. This aims to provide a more flexible design as the scheduling policy can be freely implemented in software. These designs require the implementation of a hardware component for each core of an SoC. Our proposal assumes a centralized hardware unit that is capable of operating without the need to change the SoC.

Task Superscalar [38] and Picos++ [81] use a single hardware component to accelerate parts of the runtime system. In the case of Task superscalar, all the parts of the runtime system are transferred to the accelerator. Picos++ [81] is a hardware-software co-design that supports nested tasks. This design enables the acceleration of the inter-task dependencies on a special hardware. Swarm [47] performs speculative task execution. Instead of accelerating parts of the runtime system, Swarm uses hardware support to accelerate speculation. This is different than our design that decouples only task creation.

Our work diverges to prior studies for two main reasons:

- The implementation of prior studies requires changes in hardware of the SoC. This means that they need an expensive design where each core of the chip has an extra component. Our proposal offers a much cheaper solution by requiring only a single specialized core that, according to our experiments, can manage the task creation for 512-core SoCs.

- None of the previous studies is aiming at accelerating exclusively task creation overheads. According to our study task creation becomes the main bottleneck as we increase the number of cores and our study is the first that takes this into account.

5.7 CONCLUSIONS

This paper presented TaskGenX, the first software-hardware co-design that decouples task creation and accelerates it on a runtime optimized hardware. In contrast to previous studies, our paper makes the observation that task creation is a significant bottleneck in parallel runtimes. Based on this we implemented TaskGenX on top of the OpenMP programming model. On the hardware side, our paper sets the requirements for the RTopt in order to achieve optimal results and proposes an asymmetric architecture that combines it with general purpose cores.

Based on this analysis we evaluate the performance of 11 real workloads using our approach with TaskSim simulator. Accelerating task creation, TaskGenX achieves up to 15.8× improvement (Cholesky128) over the baseline for homogeneous systems and up to 16× (Blackscholes) on asymmetric systems when the application is launched on a little core. Using TaskGenX on asymmetric systems offers a portable solution, as the task creation is not affected by the type of core that the master thread is bound to.

We further showed that for some cases like Canneal where task creation needs to be accelerated as much as 197× in order to steadily provide enough created tasks for execution. However, even by using a realistic and implementable hardware approach that offers 16× speedup of task creation, achieves satisfactory results as it improves the baseline up to 14×.

Comparing TaskGenX against other approaches such as Carbon, Nexus, Picos++ or TaskSuperscalar that manage to transfer different parts of the runtime to the RTopt proves that TaskGenX is the most minimalistic and effective approach. Even if TaskGenX transfers the least possible runtime activity to the RTopt hardware it achieves better results. This implies that TaskGenX requires a less complicated hardware accelerator, as it is specialized for only a small part of the runtime, unlike the other approaches that need specialization for task creation, dependency tracking and scheduling.

We expect that combining TaskGenX with an asymmetry-aware task scheduler will achieve even better results, as asymmetry introduces load imbalance.

CHAPTER 6

REAL-TIME SCHEDULING

CHAPTER 7

RELATED WORK

There has been a lot of studies on AMC systems. Some works focus on the system design, while other works explore the challenges that appear in efficiently utilizing such a heterogeneous system. Kumar et al [53] present the idea of an AMC system and proposed a feedback-based way to dynamically migrate processes among the different cores. To determine the core that most effectively executed a workload, Kumar et al [54] proposed the use of sampling. This method minimizes the execution time of each single thread and increases performance. Other studies focused on the pipeline design of such AMCs and the area that should be devoted to each component in the system [10, 64]. Other works on AMCs focus on hardware support for critical section detection [80] or bottleneck detection [48, 49]. These approaches are orthogonal to the ones evaluated in this paper and could benefit from them to further improve the final performance of the system.

Process scheduling on AMCs is one of the most challenging topics in this area of study. Bias scheduling [51] is an OS scheduler that characterizes the running threads according to their memory or execution intensity. It then schedules the computation intensive threads to the big cores of the system while the memory intensive threads to the little cores of the system. The experimental evaluation is done on Intel Xeon processors and the heterogeneous system is emulated by changing the configuration of three out of the four cores of the processor. Cong et al propose the Energy-Efficient [29] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [26] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et al. [85] propose the fairness-aware OS scheduler that focuses on AMC architectures. The performance impact estimation (PIE) scheduler [86] 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 [76] propose a thread scheduling technique that estimates power and performance when deciding to assign a thread to a specific core of the heterogeneous system. Finally, Energy-Aware Scheduling (EAS) is an on-going effort in the Linux community to introduce the energy factor in the OS scheduler [44, Anderson]. It is based on performance and power profiling to set performance and power capacities and let the Linux completely fair scheduler assign slots to processes considering the different core capacities. EAS is not yet part of the Linux kernel and, therefore, GTS is the most sophisticated state of the art scheduling method in production on current big.LITTLE processors.

Similar to OS scheduling approaches there have been many task scheduling approaches that are directed for utilizing AMCs. The Levelized Min Time [45] 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 [78] 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 cores but it then removes the redundant duplicates if they do not affect the makespan. CATS scheduler [27] is designed for AMCs like big.LITTLE and dynamically schedules the *critical* tasks to the big cores of the system to increase performance. Topcuoglu et al proposed the Heterogeneous Earliest Finish Time (HEFT) scheduler that statically assigns each task to the processor that will finish it at the earliest possible time. To do so, it keeps records with the task costs for each processor type. They also proposed the Critical Path on a Processor (CPOP) algorithm [83] 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 [31] identifies the tasks that belong to the critical path and schedules them with higher priority.

All these works reflect the remarkable research that is taking place on AMCs. However we consider that their experimental evaluation is limited for three main reasons: i) The evaluation is done through a simulator or emulation of an AMC [2, 10, 45, 48, 49, 51, 53, 64, 76, 78, 80, 85, 86]; ii) The evaluated applications are either random task dependency graph generators or scientific kernels and micro-benchmarks [31, 78, 83]. iii) Their evaluation does not focus on power and energy consumption [27, 45, 54, 85, 86].

This paper includes a unique evaluation of performance, power and energy on a real AMC of real parallel applications. This paper also reflects the impact of using different big and little core counts which is not present in previous works [29].

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 ef-

ficiently utilizing such a heterogeneous system. Kumar et al [53] 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 [54] 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 [10, 64]. Other works on asymmetric systems focus on hardware support for critical section detection [80] or bottleneck detection [48, 49]. 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 [51] 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 [29] OS scheduler based on energy estimation. The evaluation is performed on the Intel QuickIA [26] platform that integrates an Intel Xeon with an Atom processor. Van Craeynest et al propose the fairness-aware OS scheduler [85] that focuses on asymmetric multi-core architectures. The performance impact estimation (PIE) scheduler [86] 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 [76] 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 [45] 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 [78] 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 [83] 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 [31] 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, 10, 45, 48, 49, 51, 53, 54, 64, 76, 78, 80, 85, 86].
- The evaluated applications are either random task dependency graph generators or scientific kernels and micro-benchmarks [31, 78, 83].
- Their evaluation does not focus on power and energy consumption [45, 54, 78, 85, 86].

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.

7.0 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, 31, 43, 60, 83] 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 7.0.3. The scheduler proposed in this paper is also a list scheduler.

Clustering schedulers [43, 45, 90, 93] 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 [45] 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 [59, 68, 94] randomize their schedules by applying policies influenced by other sciences. Genetic algorithms [94] group tasks into generations and schedule them according to a randomized genetic technique. Chemical reaction algorithms [59] mimic molecular interactions to map tasks to processors. Some of these guided-random approaches are designed for heterogeneous systems [59, 94]. The scheduler by Page et al. [68] enables dynamic scheduling of multiple-sized tasks for heterogeneous systems. However, it does not support dependencies between tasks.

Duplication-based schedulers [2, 11, 97] 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.

7.0 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 [8, 35] offers this abstraction by allowing multiple implementations of a given task to be executed on different processing units [71]. The scheduler then assigns the execution of a task to the best resource according to its earliest finish time. Another case is StarPU [7], a library that offers runtime heterogeneity support and provides priority schedulers for task-to-processor allocation. AHP [70] 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.

7.0 CRITICALITY-AWARE SCHEDULERS

Several previous works propose scheduling heuristics that focus on the critical path in a TDG to reduce total execution time [31, 43, 60, 83]. 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 [83] 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 [31]. 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 [83] 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.

CHAPTER 8

CONCLUSIONS

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.

CHAPTER 9

FUTURE DIRECTIONS

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).

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

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

[4] ARM (2011a). Cortex-A15 technical reference manual, revision: r2p0.

[5] ARM (2011b). Cortex-A7 MPCore, revision: r0p3.

[6] ARM (Last accessed Oct 12, 2016). Juno ARM Development Platform.

[7] 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).

[8] 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).

[9] 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 TPDS*, 20(3):404–418.

[10] 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.

[11] Bansal, S., Kumar, P., and Singh, K. (2003). An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems. *IEEE TPDS*, 14(6).

[12] Barcelona Supercomputing Center. Barcelona Application Repository.

[13] Barcelona Supercomputing Center. Nanos++.

[14] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012a). Legion: Expressing locality and independence with logical regions. In *SC*.

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

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

[17] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216.

[18] Board, O. A. R. (2015). OpenMP Specification. 4.5.

[19] Bolaria, J. (2012). Cortex-A57 extends ARM's reach.

[20] Bueno, J., Planas, J., Duran, A., Badia, R. M., Martorell, X., Ayguadé, E., and Labarta, J. (2012). Productive programming of GPU clusters with OmpSs. In *IPDPS*, pages 557–568.

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

[22] Cao, T., Huang, W., He, Y., and Kondo, M. (2017). Cooling-aware job scheduling and node allocation for overprovisioned hpc systems. In *IPDPS'17*, pages 728–737.

[23] Casas, M., Moreto, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O., Cristal, A., Ayguade, E., Labarta, J., and Valero, M. (2015). *Runtime-Aware Architectures*, pages 16–27. Springer Berlin Heidelberg, Berlin, Heidelberg.

[24] Chapman, B. (2007). *The Multicore Programming Challenge*, pages 3–3. Springer Berlin Heidelberg, Berlin, Heidelberg.

[25] 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. *TACO*.

[26] 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.

[27] 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.

[28] 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.

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

[30] Dallou, T., Engelhardt, N., Elhossini, A., and Juurlink, B. (2015). Nexus#: A distributed hardware task manager for task-based programming models. In *IPDPS*, pages 1129–1138.

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

[32] Demler, M. (2015). Cortex-A72 takes big step forward.

[33] Dennard, R., Gaensslen, F., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. In *Solid-State Circuits, IEEE Journal of*, volume 9.

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

[35] 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.

[36] Duran, A., Corbalán, J., and Ayguadé, E. (2008a). Evaluation of OpenMP Task Scheduling Strategies. IWOMP'08.

[37] Duran, A., Perez, J. M., Ayguadé, E., Badia, R. M., and Labarta, J. (2008b). Extending the OpenMP Tasking Model to Allow Dependent Tasks. IWOMP'08.

[38] Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguade, E., Labarta, J., and Valero, M. (2010). Task superscalar: An out-of-order task pipeline. In *MICRO*, pages 89–100.

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

[40] Grass, T., Allande, C., Armejach, A., Rico, A., Ayguadé, E., Labarta, J., Valero, M., Casas, M., and Moreto, M. (2016). Musa: A multi-level simulation approach for next-generation hpc machines. In *SC16*, pages 526–537.

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

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

[43] Hakem, M. and Butelle, F. (2005). Dynamic Critical Path Scheduling Parallel Programs onto Multiprocessors. In *IPDPS*.

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

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

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

[47] Jeffrey, M. C., Subramanian, S., Yan, C., Emer, J., and Sanchez, D. (2015). A scalable architecture for ordered parallelism. In *MICRO*, pages 228–241.

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

[49] 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.

[50] 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.

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

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

[53] 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.

[54] 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.

[55] Kumar, S., Hughes, C. J., and Nguyen, A. (2007). Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, pages 162–173.

[56] Laurenzano, M. A., Tiwari, A., Cauble-Chantrenne, A., Jundt, A., Ward, W. A., Campbell, R., and Carrington, L. (2016). Characterization and bottleneck analysis of a 64-bit armv8 platform. In *ISPASS'16*, pages 36–45.

[57] Li, K., Tang, X., and Li, K. (2014). Energy-efficient stochastic task scheduling on heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2867–2876.

[58] Li, K., Tang, X., Veeravalli, B., and Li, K. (2015). Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *IEEE Transactions on Computers*, 64(1):191–204.

[59] Li, K., Zhang, Z., Xu, Y., Gao, B., and He, L. (2012). Chemical Reaction Optimization for Heterogeneous Computing Environments. In *ISPA*.

[60] 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 *ICPADS*.

[61] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F., and Mahlke, S. (2012). Composite cores: Pushing heterogeneity into a core. In *MICRO*, pages 317–328.

[62] Manivannan, M. and Stenström, P. (2014). Runtime-guided cache coherence optimizations in multi-core architectures. In *IPDPS*.

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

[64] 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.

[65] Moschakis, I. A. and Karatza, H. D. (2015). A meta-heuristic optimization approach to the scheduling of bag-of-tasks applications on heterogeneous clouds with multi-level arrivals and critical jobs. *Simulation Modelling Practice and Theory*, 57:1–25.

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

[67] OpenMP Architecture Review Board (2015). OpenMP architecture review board: Application programming interface 4.5.

[68] 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*.

[69] Papaefstathiou, V., Katevenis, M. G., Nikolopoulos, D. S., and Pnevmatikatos, D. (2013). Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 325–334, New York, NY, USA. ACM.

[70] Pienaar, J. A., Chakradhar, S., and Raghunathan, A. (2012). Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems. In *SC*.

[71] Planas, J., Badia, R., Ayguade, E., and Labarta, J. (2013). Self-Adaptive OmpSs Tasks in Heterogeneous Environments. pages 138–149.

[72] 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 *SC*.

[73] Reinders, J. (2007). *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly.

[74] Ren, B., Krishnamoorthy, S., Agrawal, K., and Kulkarni, M. (2017). Exploiting vector and multicore parallelism for recursive, data- and task-parallel programs. In *PPoPP '17*, pages 117–130, New York, NY, USA.

[75] 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 TACO*, 8(4).

[76] 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.

[77] Sanchez, D., Yoo, R. M., and Kozyrakis, C. (2010). Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322.

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

[79] Själander, M., Terechko, A., and Duranton, M. (2008). A look-ahead task management unit for embedded multi-core architectures. In *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 149–157.

[80] 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.

[81] Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2017). General purpose task-dependence management hardware for task-based dataflow programming models. In *IPDPS*, pages 244–253.

[82] Tang, X., Pattnaik, A., Jiang, H., Kayiran, O., Jog, A., Pai, S., Ibrahim, M., Kandemir, M. T., and Das, C. R. (2017). Controlled kernel launch for dynamic parallelism in gpus. In *HPCA'17*, pages 649–660.

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

[84] Turley, J. (2011). Cortex-A15 eagle flies the coop.

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

[86] 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.

[87] Vandierendonck, H., Chronaki, K., and Nikolopoulos, D. S. (2013). Deterministic scale-free pipeline parallelism with hyperqueues. In *SC*.

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

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

[90] Wu, M.-Y. and Gajski, D. (1990). Hypertool: a Programming Aid for Message-Passing Systems. *IEEE TPDS*, 1(3).

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

[92] Xu, Y., Li, K., He, L., Zhang, L., and Li, K. (2015). A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3208–3222.

[93] Yang, T. and Gerasoulis, A. (1994). DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE TPDS*, 5(9).

[94] Yu, H. (2007). A Hybrid GA-based Scheduling Algorithm for Heterogeneous Computing Environments. In *SCIS*.

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

[96] Zhang, F., Cao, J., Li, K., Khan, S. U., and Hwang, K. (2014). Multi-objective scheduling of many tasks in cloud platforms. *Future Generation Computer Systems*, 37:309 – 320.

[97] 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. *IEEE Trans. Comput.*, 60(3).

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