Critically-Aware Dynamic Task Scheduling for Heterogeneous Architectures

Kallia Chronaki*, Alejandro Rico*, Rosa M. Badia†, Eduard Ayguadé†, Jesús Labarta†, Mateo Valero‡

†Barcelona Supercomputing Center, Barcelona, Spain
‡Universitat Politècnica de Catalunya - BarcelonaTech, Barcelona, Spain

Copyright 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ABSTRACT
Current and future parallel programming models need to be portable and efficient when moving to heterogeneous multi-core systems. OmpSs is a task-based programming model with dependency tracking and dynamic scheduling. This paper describes the OmpSs approach on scheduling dependent tasks onto the asymmetric cores of a heterogeneous system. The proposed scheduling policy improves performance by prioritizing the newly-created tasks at runtime, detecting the longest path of the dynamic task dependency graph, and assigning critical tasks to fast cores. While previous works use profiling information and are static, this dynamic scheduling approach uses information that is discoverable at runtime which makes it implementable and functional without the need of an oracle or profiling. The evaluation results show that our proposal outperforms a dynamic implementation of Heterogeneous Earliest Finish Time by up to 1.15×, and the default breadth-first OmpSs scheduler by up to 1.3× in an 8-core heterogeneous platform and up to 2.7× in a simulated 128-core chip.

Categories and Subject Descriptors
C.1.3 [Processor Architectures]: Heterogeneous systems; C.1.4 [Processor Architectures]: Mobile processors; D.1.3 [Programming Techniques]: Parallel programming; D.3.2 [Programming Languages]: Parallel languages

Keywords
High performance computing, Scheduling, Heterogeneous systems, Task-based programming models

1. INTRODUCTION
The future of high-performance computing is highly restricted by energy efficiency [18]. The use of heterogeneous multi-core architectures is an approach towards increasing energy efficiency [13, 14]. These architectures feature different types of processing cores that are designed to target different performance and power optimization points.

Load balancing and scheduling are two of the main challenges in utilizing such heterogeneous platforms. The use of task-based programming models with dynamic scheduling is an answer to tackle these challenges. Some of these programming models allow the specification of inter-task dependencies that enable automatic scheduling and synchronization by the runtime system. OmpSs [10, 4] is an example of this type of programming models. It maintains a dynamic directed-acyclic graph of tasks with their current state. Whenever a task’s dependencies are satisfied, it becomes ready to be scheduled to an available core.

The mapping of ready tasks to different types of cores on a heterogeneous system becomes a challenge when considering the reduction of the total execution time. Some task-based applications expose a complex dependency graph in which tasks in the critical path determine the total application duration. This opens an opportunity to accelerate the overall application by running critical tasks on fast cores. Some previous works [15, 9, 25, 20] tackled this issue using static scheduling over the whole dependency graph 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 task dependency graphs (TDGs).

In this paper, we propose a critically-aware dynamic task scheduler that dynamically assigns critical tasks to fast cores to improve performance in a heterogeneous system with fast and slow cores. Compared to previous proposals, this scheduler is based on information discoverable at runtime, is implementable and works without the need of an oracle or profiling. Furthermore, our evaluation is based on a real heterogeneous multi-core platform with real applications and, therefore, using real TDGs.

The contributions of this paper are the following:

- A novel criticality-aware task scheduler (CATS) that dynamically assigns critical tasks to fast cores in a heterogeneous multi-core. Tasks are defined to be critical if they are part of the longest path in the in-flight dynamic state of the dependency graph. The flexibility and work stealing policy of our scheduler are configurable. Flexibility increases the number of tasks...
considered critical. Work stealing may be uni- or bi-directional: only fast cores can steal from slow cores, or slow cores can also steal from fast cores.

- An evaluation of our implementation of CATS in the OmpSs programming model compared to a dynamic implementation of Heterogeneous Earliest Finish Time [25] and the default OmpSs scheduler. We evaluate the effectiveness of CATS on different numbers of cores and shares of fast and slow cores on an Odroid-XU3 development board featuring an eight-core Samsung Exynos 5422 chip with ARM big.LITTLE architecture including four Cortex-A15 and four Cortex-A7 cores. We also evaluate the effectiveness of CATS on different speed ratios between fast and slow cores using simulation of an heterogeneous system with up to 128 cores. The results show that CATS improves overall performance up to 1.3× on the real eight-core platform, and up to 2.7× on a simulated 128-core system.

2. RELATED WORK

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

2.1 Schedulers for Heterogeneous Systems

Previous works on schedulers for heterogeneous systems form four different types of schedulers: listing, clustering, guided-random, and duplication-based schedulers. Listing schedulers [1, 15, 9, 25, 20] have two scheduling stages. In the first stage, each task is given a priority based on the policy defined in each algorithm. In the second stage, tasks are assigned to processors depending on their priorities. Most criticality-aware schedulers fall in this category, and we discuss them in Section 2.3. The scheduler proposed in this paper is also a list scheduler.

Clustering schedulers [26, 27, 15, 17] 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 [17] 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 [28, 19, 21] randomize their schedules by applying policies influenced by other sciences. Genetic algorithms [28] group tasks into generations and schedule them according to a randomized genetic technique. Chemical reaction algorithms [19] mimic molecular interactions to map tasks to processors. Some of these guided-random approaches [28, 19] are designed for heterogeneous systems. The scheduler by Page et al. [21] enables dynamic scheduling of multiple-sized tasks for heterogeneous systems. However, it does not support dependencies between tasks.

Duplication-based schedulers [5, 29, 2] aim to eliminate communication costs between processors by scheduling tasks and their successors on the same processor. If a task has many successors, it is duplicated and executed in multiple cores prior to its successors so all successor tasks get the data from their predecessors with the lowest communication cost. This scheduling potentially introduces redundant duplications of tasks which may lead to bad schedules. The Heterogeneous Economical Duplication scheduler [2] performs task duplication in an economical manner as it removes the redundant duplicates if they do not affect performance.

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

2.2 Schedulers for Compute Accelerators

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

Most heterogeneous systems with compute accelerators nowadays combine general-purpose CPUs and GPU compute accelerators. There is a set of programming models providing abstractions to ease the development of applications on these platforms. OmpSs [10, 4] offers this abstraction by allowing multiple implementations of a given task to be executed on different processing units [23]. The scheduler then assigns the execution of a task to the best resource according to its earliest finish time. Another case is StarPU [3], a library that offers runtime heterogeneity support and provides priority schedulers for task-to-processor allocation. AHP [22] is another framework that generates software pipelines for heterogeneous systems and schedules tasks to their earliest executor, based on profiling information gathered prior to runtime.

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

2.3 Criticality-Aware Schedulers

Several previous works propose scheduling heuristics that focus on the critical path in a TDG to reduce total execution time [15, 9, 25, 20]. 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 [25] 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 [9]. LDCP also statically schedules the first 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 [25] 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.

3. OMPSS PROGRAMMING MODEL

OmpSs is a task-based programming model that offers a high level abstraction to the implementation of parallel applications for various homogeneous and heterogeneous architectures [10, 4]. It enables the annotation of function declarations with the task directive, which declares a task. Every invocation of a function creates a task that is executed concurrently with other tasks or parallel loops. OmpSs also supports task dependencies and it uses the StarsSs [12] 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, responsible for the internal creation and execution of the tasks.

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.

4. CRITICALITY-AWARE SCHEDULER

The proposed scheduling algorithm generally applies to task-based programming models supporting task dependencies, but for simplicity we explain it in the context of the OmpSs programming model.

The Criticality-Aware Task Scheduler (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 depending on the work stealing policy. Currently, we support two work stealing mechanisms: simple work stealing, i.e., fast cores can steal from slow cores; and bi-directional (2DS) work stealing, i.e., both types can steal from the other.

The default policy is simple.

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

To give an overview of the scheduling process, Figure 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
4.2 Task Submission

The purpose of this step is to divide the tasks in two groups: critical and non-critical. Critical tasks are tasks that belong to the longest path of the dynamic TDG. The longest path is the path of the TDG 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 those 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 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:

- Whether the priority of the current ready task is higher than the priority of the last discovered critical task (maxPriority).
- Whether the current ready task is the highest-priority immediate successor of the last discovered critical task.

In the first case, the algorithm detects new longest paths that may have been created by the application throughout the execution of a prior longest path. In this case, the scheduler can either be strict or flexible:

- Strict: marks as critical tasks with priority higher than the priority of the last critical task.
- Flexible: marks as critical tasks with priority higher or equal to the priority of the last critical task.

As a result, the flexible scheduler ends up with more critical tasks than the strict. The flexibility of the scheduler can be set by the programmer through an environment variable.

The task that satisfies the second condition is a task with a lower priority than the maximum but the task belongs to the longest path because it is the highest priority immediate successor of the last detected critical task.

Listing 2 shows a simplified version of the task submission code. 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 if in flexible mode) 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 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. 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 2. For every newly-ready task, the conditions listed before will be evaluated. When task 7 is submitted, it will not be considered as critical because it does not belong to the maxTaskSucc list and its priority is not equal to maxBoLevl = 1. Contrarily, task 8 satisfies both conditions and so the task is inserted in the critical queue.

4.3 Task-to-Core Assignment

Task-to-core assignment takes place dynamically and in parallel to the previous steps. 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 the other group 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 default work-stealing mechanism for CATS lets fast cores steal work from slow cores whenever the critical queue becomes empty. Also, CATS can be configured to perform bi-directional work stealing so slow processors can also steal tasks from the critical queue if the non-critical queue becomes empty. We evaluate these different options and show the results in the next section.

5. EVALUATION

5.1 Methodology

We measure the execution time of four criticality-sensitive applications using CATS, the default OmpSs scheduler and the Heterogeneous Earliest Finish Time scheduler (HEFT) [25] implemented in OmpSs. We evaluate four different CATS configurations based on the options explained in Section 4. The options consist of whether the task submission policy is strict or flexible, and the type of the work stealing mechanism. This results in the following configurations:

- Flexible with simple work stealing (SS FLEX)
- Flexible with bidirectional work stealing (2DS FLEX)
- Strict with simple work stealing (SS STRICT)
- Strict with bidirectional work stealing (2DS STRICT)

The default OmpSs scheduler employs a breadth-first policy (BF) [11]. The BF scheduler 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. Tasks are ordered according to their ready time: the earliest ready task resides at the head of the queue. Since the ready queue is shared, there is no need for

![Diagram](https://via.placeholder.com/150)
work stealing and the load is balanced automatically. BF does not differentiate among core types and assigns tasks in a first-come-first-served basis.

We implemented a dynamic version of the HEFT scheduler (dHEFT) in the OmpSs programming model. The implementation assumes two different types of cores (big and little) and keeps records of the tasks’ execution times in each core. The original HEFT [25] implementation assumes the prior knowledge of the TDG as well as the costs of the tasks. In our case, since the evaluation consists of running real applications, the best way to compare HEFT to our proposal is to keep the scheduling idea of HEFT and transform it from a static to a dynamic scheduler. This means that dHEFT discovers the costs of the tasks at runtime, computes a mean value of the costs for each task-type and parameter-size tuple for each type of core, and then finds the core that will finish the task at the earliest possible time. A difference between the static HEFT and dHEFT is the order in which tasks are being submitted. In HEFT tasks are submitted in decreasing order of their upward rank, which is computed statically with the known task costs. Since dHEFT discovers the costs as tasks execute, there is no priority in task submission. In dHEFT tasks are submitted as soon as they become ready.

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

Our test bed comprises a big.lITTLE processor and a simulated heterogeneous system:

### 5.1.1 ODROID-XU3

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

#### 5.1.2 Simulation

To evaluate CATS on larger multi-core systems we use the heterogeneous multi-core TaskSim simulator [24]. 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 CATS on a total of 80 distinct heterogeneous machine configurations. These comprise systems with the total number of cores ranging from 16 to 128, and the number of fast cores ranging from 1 to 16. For all these configurations, we evaluate the following performance ratios between fast and slow cores: 2×, 2.5×, 3×, 3.5× and 4×.

For both real and simulated platforms, each set-up has a given number of total and big cores. Our metrics are the improvement of CATS over the baseline scheduler, and the speedup over the execution on one little core. Equations 1 and 2 show the improvement and speedup calculations.

\[
\text{Improv. over BF(total, big)} = \frac{\text{Exec. time BF(total, big)}}{\text{Exec. time CATS(total, big)}}
\]

\[
\text{Speedup(total, big)} = \frac{\text{Exec. time(1, 0)}}{\text{Exec. time(total, big)}}
\]

The execution time in these calculations is the average of 10 executions of the application on each machine set-up.

### 5.2 Applications

We use four scientific kernels implemented in the OmpSs programming model: Cholesky factorization, QR factorization, Heat diffusion and Integral Histogram. These benchmarks are accessible in the BSC Application Repository [6]. Their different configurations and characteristics are shown in Table 1.

The applications have different sensitiveness depending on the inter-task dependencies that determine the available parallelism and the percentage of critical tasks, shown in the table. The larger proportion of critical tasks, the larger the potential improvement of CATS, as these are tasks that, once accelerated, reduce the application overall execution time. The percentage of critical tasks in Table 1 is from a single-core execution of CATS. Since CATS is dynamic, the tasks considered critical would be different for different configurations, since the algorithm runs on a partial dependency graph, including only the outstanding tasks. However,

---

### Table 1: Evaluated benchmarks and relevant characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem size</th>
<th>#Tasks</th>
<th>#Task types</th>
<th>Avg task exec. time (µs)</th>
<th>Measured perf. ratio</th>
<th>Critical tasks %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky factorization</td>
<td>8×8 blocks of 1024×1024 floats, 16×16 blocks of 512×512 floats, 32×32 blocks of 512×512 floats</td>
<td>120</td>
<td>4</td>
<td>1.009.786</td>
<td>2.23</td>
<td>17.50</td>
</tr>
<tr>
<td>QR factorization</td>
<td>16×16 blocks of 512×512 doubles</td>
<td>816</td>
<td>4</td>
<td>1.478.12</td>
<td>2.04</td>
<td>5.51</td>
</tr>
<tr>
<td>Heat diffusion</td>
<td>16×16 blocks of 512×512 doubles</td>
<td>5084</td>
<td>4</td>
<td>1.009.786</td>
<td>2.23</td>
<td>17.50</td>
</tr>
<tr>
<td>Integral Histogram</td>
<td>8×8 blocks of 512×512 floats</td>
<td>1.496</td>
<td>4</td>
<td>1.225.254</td>
<td>4.26</td>
<td>4.47</td>
</tr>
</tbody>
</table>
the single-core figures serve as an estimation of the percentage of critical tasks intrinsic to the overall application, that allows us to reason about the results with respect to this application characteristic.

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 get a larger benefit from running on the big cores, than single-precision dominated applications, as shown in Table 1.

The performance ratios in Table 1 are computed for the real machine by comparing the execution time on one little core over the execution time on one big core. Using these performance ratios, we can estimate the ideal speedup over a little core for each application running on all eight cores. Equation 3 is used for the ideal speedup. The ideal speedup considers a fully parallel workload, without any dependencies, overheads or sequential sections, thus unachievable by the dependency-intensive applications in our evaluation.

\[
\text{ideal speedup}(\text{workload}, 8) = 4 \times \frac{\text{perf ratio}(\text{workload})}{4} + 4 \quad (3)
\]

The average overhead per task for each scheduler is negligible compared to the average task execution time shown in Table 1. Specifically, it is observed that the per-task overhead ranges in the order of tens of microseconds for BF and in the order of hundreds of microseconds for dHEFT and CATS when running on 8 cores. The most time consuming phase of CATS is the task prioritization, in which the TDG is traversed to update the bottom levels of the tasks. For dHEFT, the search of the appropriate worker for a task becomes an obstacle in performance. Normally these obstacles in heterogeneous schedulers are paid off by the more effective task execution.

5.2.1 Cholesky Factorization

Cholesky factorization is a dense matrix operation that is used for the efficient solution of linear equations in linear least square systems. The OmpSs implementation of Cholesky blocks the input matrix into square blocks of floats and it then performs the factorization on each block using tasks that call the functions of the Intel MKL library [16].

Figure 3 shows the TDGs for input sizes of (a) 8x8 and (b) 16x16 blocks. Critical tasks are denoted as red nodes. The TDG becomes wider as the number of blocks increases. This reduces the percentage of critical tasks as shown in Table 1. The 8x8 blocks case shows a narrower TDG that makes the application more criticality sensitive than the 16x16 blocks case that exposes more parallelism. We evaluate both configurations to show the impact of scheduling on different criticality sensitiveness of the application configuration.

5.2.2 QR Factorization

QR Factorization is a linear algebra algorithm that is often used to solve the linear least squares problem [8]. We evaluate the performance of a blocked, communication avoiding QR factorization implementation in OmpSs. We used an input matrix of 8192x8192 doubles and a block size of 512x512, that creates 16x16 blocks.

5.2.3 Heat Diffusion

Heat diffusion benchmark uses the Gauss-Seidel method to compute the heat distribution on a matrix A from heat sources represented by x. Heat diffusion implements an iterative solver of the equation that invokes the Gauss-Seidel method until the desired convergence is reached. In the specific implementation [6] the number of iterations can be set as an argument in order to control the workload of the experiments. The code splits the matrix into blocks and creates one task per block for the Gauss-Seidel computation. We use a matrix of 8192x8192 doubles and block size of 512x512. We set the number of iterations to 20.

5.2.4 Integral Histogram

The integral histogram is a method to compute a cumulative histogram for each pixel of an image [7]. The OmpSs implementation performs a blocked cross-weave scan for each block of the image. The horizontal scan processes one image block at a time and transmits the histograms to the block on the right. The vertical scan processes one block and transmits the histograms to the block below the current block. Due to the histograms’ transmissions, the application introduces many task dependencies. We evaluate Integral Histogram with an input image of 4096x4096 pixels and block size of 512x512, resulting in an 8x8 blocked matrix.

5.3 Real Environment Evaluation

Figure 4 shows the results from the experiments on the ODROID-XU3. Figure 4a shows the speedup of CATS (SS FLEX), dHEFT and BF when running the applications on all eight cores, as well as the estimated ideal speedup of the platform for each application. 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. For all cases, CATS scales better than dHEFT and BF. The shortening of the critical path by running all critical tasks on big cores effectively shortens total execution time when running on all cores.

Figure 4b shows the improvement of the CATS configurations and dHEFT over BF, and the speedup obtained with CATS, dHEFT and BF, for Cholesky on an 8x8 blocked matrix. CATS consistently achieves better performance than dHEFT and BF and the improvement over BF increases as the number of cores is increased. Specifically, the improvement is observed to be up to 30% when running on seven and eight cores.

Figure 4c shows the performance on a 16x16 block input matrix, where the improvement of CATS is smaller and ranges from 2 to 9% and all schedulers perform fairly well.

Figure 3: Cholesky factorization task dependency graphs.
In this case the opportunities for enhancement are limited, since, according to Figure 4a, BF performance approaches the ideal speedup. The lower improvement in this case comes from the fact that the application is less sensitive to the critical path. The task graph is wider (as shown in Figure 3) and, accordingly, the percentage of critical tasks is lower. However, CATS still outperforms BF by 7% when using the eight cores in the system and performs as good as dHEFT.

Figure 4d shows the improvement of CATS and dHEFT over BF and their speedup for QR factorization. QR consists of double precision operations which cause the big cores to be 4.26× faster than the little cores. Thus, the ideal speedup of the system for QR is 20.1×. CATS achieves a 15× speedup by shortening the execution of the dynamic longest path in the TDG. Since dHEFT is not focusing on the longest path of the TDG, and it performs some random scheduling at the beginning of the execution in order to determine the task costs, it becomes less effective than CATS but still better than BF as the number of cores increases.

Figure 4e shows the improvement of the different CATS configurations and dHEFT over BF, and the speedup obtained, for heat diffusion. As shown in Table 1 heat diffusion consists of 5124 tasks; 5120 tasks of them are tasks of the same type and parameter size. This limits the effect of dHEFT, that tries to find the earliest executor for each task in comparison to BF in which whenever a core becomes available, it retrieves a task from the ready queue. Thus, dHEFT is better than BF because it uses private ready queues for each core, thing that reduces the congestion of using only one ready queue. CATS consistently improves the scheduling of the asymmetric system from 15% to 22%, since the main criterion of scheduling is the TDG structure. CATS achieves a 13× speedup when using all eight cores, thus getting very close to the ideal 15.3× shown in Figure 4a.

Figure 4f shows the improvement of CATS and dHEFT over BF, and the speedup obtained, for integral histogram. The impact of CATS is again positive for all configurations, since the improvement is at least 5% with the peak be-
Improvement over BF
Number of big cores
Total number of cores
perf 2 perf 2.5 perf 3 perf 3.5 perf 4

An interesting observation is that for single-precision kernels the improvement of CATS is proportional to the percentage of critical tasks. Integral histogram achieves greater improvements in comparison to Cholesky 16×16 since it schedules immediately more tasks to the fast cores of the system, according to the percentage of the critical tasks on Table 1. On the other hand, QR and heat diffusion show the opposite effect: QR has the largest performance ratio and a larger percentage of critical tasks than heat diffusion. However, heat diffusion shows higher overall improvement over the different configurations. We attribute this to the larger sensitivity of heat diffusion to the critical path, which allows CATS to achieve a large improvement even for a configuration of one big and one little core. The dHEFT scheduler outperforms BF but struggles to handle applications with complex TDG. Applications that consist of multiple instances of the same task type benefit less from dHEFT than applications that process tasks with variable costs, such as Cholesky and QR.

In all cases, the SS FLEX configuration of CATS achieves the best performance, since it produces a decent amount of critical tasks for the big cores, fact that shortens the longest path of the TDG. A smaller amount of critical tasks is produced by the SS STRICT policy, which causes a slight imbalance that is fixed through the work stealing mechanism but with lower effectiveness. The 2DS FLEX configuration, produces the same amount of critical tasks as the SS FLEX, but the bi-directional work stealing allows little cores to steal critical tasks, which lengthens the critical path execution and directly increases overall execution time.

5.4 Simulations

To estimate the impact of CATS on larger systems, we run three of the benchmarks using the TaskSim simulator [24]. Integral Histogram is excluded from the simulated evaluation because it does not scale beyond 16 cores. Also, dHEFT is not included in the simulation experiments because the simulation platform does not support informing task execution costs to the scheduler. Simulation allows us to evaluate on larger systems and multiple performance ratios. The results contain a fixed scheduling overhead for all configurations, regardless of the dynamic overheads during execution (e.g., work stealing). The error introduced by this assumption is small because, as stated above, the average scheduling overheads in our applications are negligible compared to the average task execution time. For space purposes, we only show simulation results for the best CATS configuration: SS FLEX.

Figures 5, 6 and 7 show the improvement of CATS over BF in systems with multiple performance ratios (perf x) for Cholesky, QR and heat diffusion respectively. The impact of CATS is larger for systems with higher performance ratio between fast and slow cores. This makes sense, as a faster execution of the critical path directly shortens overall execution time. In addition, CATS utilises fast cores more effectively than BF, which results in larger improvements with larger number of fast cores. In Figure 5, the improvement for 16 cores is comparatively small. This is because the problem size used in this specific experiment is a matrix of $16384 \times 16384$ floats using $512 \times 512$ blocks, which creates a $32 \times 32$ blocked matrix. This is because the other Cholesky configurations do not scale to 128 cores. The small amount of critical tasks in the $32 \times 32$ input, as shown in Table 1, makes the workload less sensitive to critical tasks and limits the improvement of CATS for Cholesky to a maximum of $1.4 \times$, while QR and heat diffusion are improved by factors of $1.6 \times$ and $2.7 \times$. Again, heat diffusion gets the largest improvement due to its larger sensitivity to criticality, as was also shown in the real platform evaluation.
6. CONCLUSION

We introduced the first criticality-aware dynamic scheduling policy for heterogeneous environments. Contrary to previous works on criticality-aware scheduling that use synthetic task graphs and require previous knowledge of profiling information, our proposal works on real platforms with real applications, is implementable and works without the need of an oracle or profiling.

We implemented and evaluated our criticality-aware task scheduler in the runtime system of the OmpSs programming model getting satisfactory results. The implementation shown in this paper will be included in the next stable release of OmpSs. Furthermore, there are no restrictions on applying our policy to other task-based programming models with support for task dependencies.

From our experiments on a real heterogeneous multi-core platform, we found a consistent performance improvement over the default breadth-first scheduling policy and a dynamic implementation of Heterogeneous Earliest Finish Time. The improvement of our proposal, which in most cases ranges from 10 to 20% and reaches up to 30%, is larger as we increase the number of cores. This gives a positive projection for CATS, as it is expected that the number of cores in multi-cores will increase throughout future generation designs.

From our simulation experiments, we found out that the improvement of CATS increases over the baseline with larger differences of performance among fast and slow cores. We explored performance ratios between two and four times faster fast cores over slow cores, with improvements ranging from 30% to 170%.

In conclusion, this paper shows the potential of a criticality-aware scheduler to speed up dependency-intensive applications and take advantage of the asymmetric compute resources. For future work, we aim to extend the scheduling policy to be adaptive so it can dynamically adjust its flexibility and work stealing policy depending on the application characteristics and availability of resources at runtime.

Acknowledgements

The authors would like to thank Paul Carpenter and Alex Ramirez for their help to improve the quality of this paper.

This work has been supported by the grant SEV-2011-00067 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation

7. REFERENCES