Task Scheduling Techniques for Asymmetric Multi-core Systems

Kallia Chronaki, Alejandro Rico, Marc Casas, Miquel Moretó,
Rosa M. Badia, Eduard Ayguadé, Jesus Labarta, and Mateo Valero

Abstract—As performance and energy efficiency have become the main challenges for next-generation high-performance computing, asymmetric multi-core architectures can provide solutions to tackle these issues. Parallel programming models need to be able to suit the needs of such systems and keep on increasing the application's portability and efficiency. This paper proposes two task scheduling approaches that target asymmetric systems. These dynamic scheduling policies reduce total execution time either by detecting the longest or the critical path of the dynamic task dependency graph of the application, or by finding the earliest executor of a task. They use dynamic scheduling and information discoverable during execution, fact that makes them implementable and functional without the need of off-line profiling. In our evaluation we compare these scheduling approaches with two existing state-of the art heterogeneous schedulers and we track their improvement over a FIFO baseline scheduler. We show that the heterogeneous schedulers improve the baseline by up to 1.45× in a real 8-core asymmetric system and up to 2.1× in a simulated 32-core asymmetric chip.

Index Terms—Scheduling, Heterogeneous, Multi-core

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 [1], [2], [3] 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 [4]. An approach towards these challenges is the use of task-based programming models [5], [6], [7], [8]. 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 [9], [10], [11], [12] 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 [11], [13]. Specifically we compare our approaches to the the criticality aware task scheduler (CATS) [13] as well as a dynamic implementation of the heterogeneous earliest finish time scheduler (HEFT) [11]. We implement these scheduling policies in the OmpSs [5], [14] 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 [13] by assigning to the fast cores tasks that belong either to the critical path or to the longest path of the TDG,

- *Kallia Chronaki, Eduard Ayguade, Jesus Labarta and Mateo Valero are with Barcelona Supercomputing Center and Universitat Polytéchnica de Catalunya. Email: first.last@bsc.es*
- *Marc Casas, Miquel Moreto, are with Barcelona Supercomputing Center. Email: first.last@bsc.es*
- *Alejandro Rico is with ARM. Email: alejandro.rico@arm.com*
- *Rosa M. Badia is with Barcelona Supercomputing Center and Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC). Email: rosa.m.badia@bsc.es*

Manuscript received April 11, 2016;

© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/TPDS.2016.2633347

TABLE 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

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 [13] and HEFT [11], 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 1 shows the acronyms that we use in the next sections.

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 [5], [14]. 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 [7]. 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) [15] 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.

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) [13], 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) [11], 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.

3.1 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

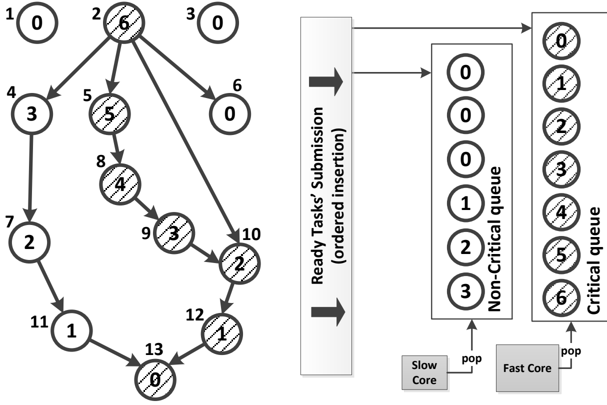


Fig. 1: Task submission with CATS. Nodes are marked with the *bottom level* of each task. Pattern-filled nodes mark the critical tasks.

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 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.

3.1.1 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 1, when the dependency between tasks 2 and 5 occurs, the task number 2 is inserted into the *plist* of the 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.

```
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 1: Pseudo-code task prioritization with CATS.

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 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.

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.

3.1.2 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*

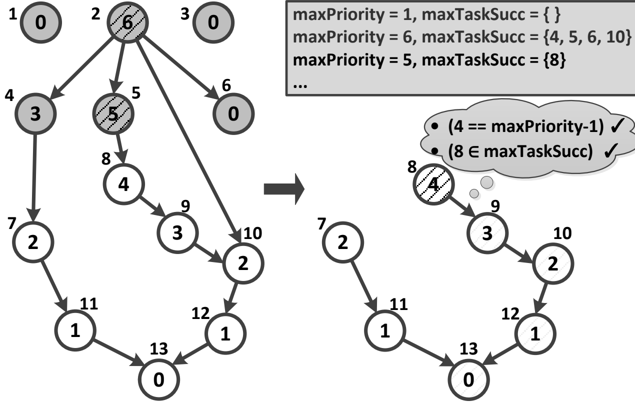


Fig. 2: Task submission. Gray nodes indicate finished tasks and pattern-filled nodes indicate critical tasks.

```
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 ∈ 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 2: Pseudo-code for task submission with CATS.

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: *(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 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, 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 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 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.

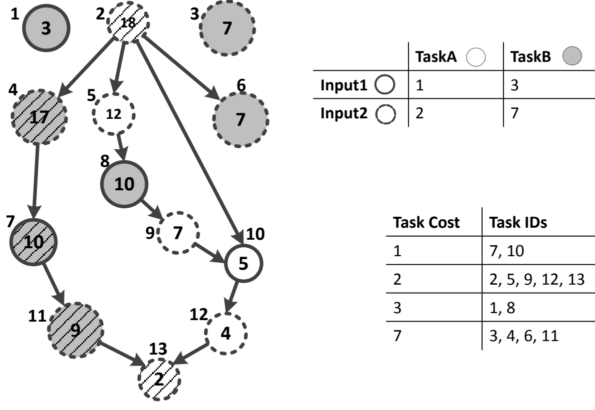
3.1.3 *Task-to-Core Assignment*

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 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.

3.2 Critical Path Scheduler

The Critical Path scheduler (CPATH) dynamically detects the critical path of the TDG. Like CATS, CPATH separates tasks into two groups: critical and non-critical tasks. The detected critical tasks are executed by the fast cores in the



	TaskA	TaskB
Input1	1	3
Input2	2	7

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

Fig. 3: Priority assignment taking into account the task costs. Task costs are assumed known and are shown in the tables.

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.

3.2.1 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 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 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 3 indicates the execution time of the tasks according

```
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 ∉ 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 3: Pseudo-code for taskExit, the function called by the cores used as reference for tracking the task costs

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 3 and 4 show how the critical path scheduler performs task prioritization. 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 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. This function is responsible for TDG traversal.

The `taskExit` function in Listing 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

```
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: Pseudo-code for task prioritization with CPATH

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 3) the algorithm traverses 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 3). For each one of the entry nodes the `updatePriorities` function is called (line 19 of Listing 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 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 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 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 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.

3.2.2 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 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 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 3.1.3. 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.

3.3 Hybrid Criticality Scheduler

The Hybrid Criticality Scheduler (HYBRID) is a combination of the CATS and CPATH scheduling policies. HYBRID keeps the simplicity of the implementation of CATS and introduces the task execution time only if available. This results in an efficient low-overhead scheduler that computes the critical path of a TDG more faithfully than CATS and with lower overheads than CPATH. This section describes HYBRID through its relation to CATS and CPATH described in Sections 3.1 and 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.

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 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 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.

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

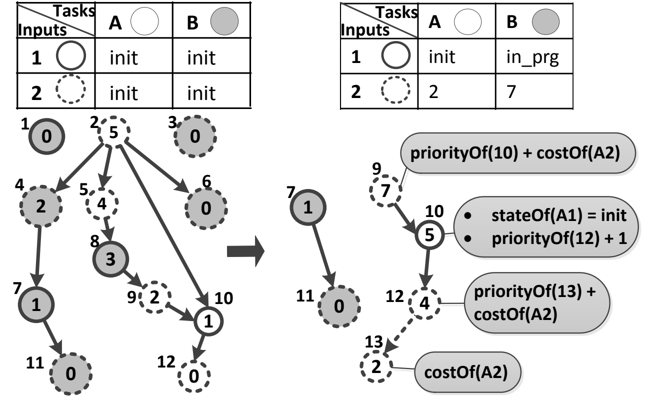


Fig. 4: Priority assignment with HYBRID scheduler. Priority update when the edge between tasks 12 and 13 is created

Figure 4 shows an example of task prioritization with HYBRID. The tables show the 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.

3.4 Dynamic Heterogeneous Earliest Finish Time Scheduler

The Heterogeneous Earliest Finish Time (HEFT) algorithm [11] 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 [11]. 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. [11] 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 [13]. 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 [13] 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 Evaluation

4.1 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. 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 [16]. 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 1.

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

4.2 Applications

We use five scientific applications implemented in the OmpSs programming model: Cholesky factorization, QR factorization, Heat diffusion, Integral Histogram and Bodytrack. These benchmarks are accessible in the BSC Application Repository [17] and in the PARSECSs library [18].

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

QR Factorization is a linear algebra algorithm that is used to solve the linear least squares problem [19]. We evaluate the performance of a blocked, communication avoiding QR implementation in OmpSs. We use an input blocked matrix of 8192×8192 doubles forming 16×16 blocks.

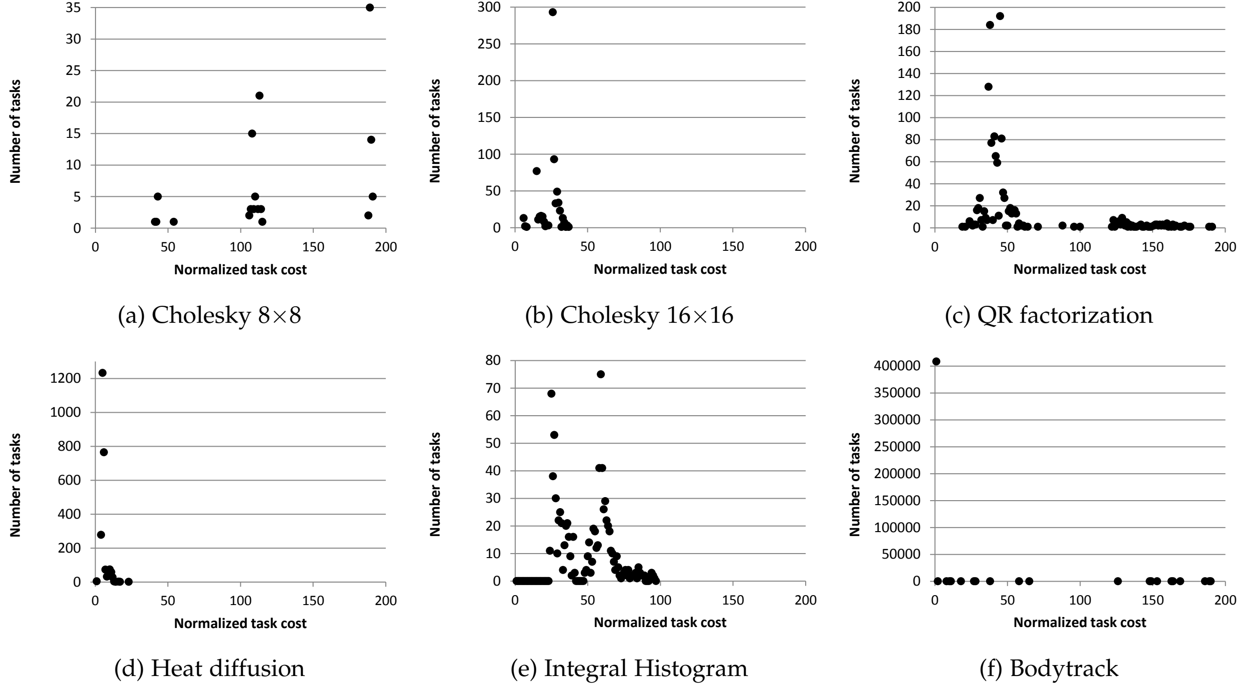
Heat diffusion uses the Gauss-Seidel method to compute the heat distribution on a matrix from x heat sources. Heat diffusion implements an iterative solver of the equation that invokes the Gauss-Seidel method until the desired convergence is reached. We use a matrix of 8192×8192 doubles and block size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

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

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 2: Evaluated benchmarks and relevant characteristics

Application	Problem size	#Tasks	#Task types	Avg task exec. time (μs)	Per task overheads (μs)			Measured perf. ratio
					CATS	CPATH	HYBRID	
Cholesky factorization	8×8 blocks of 1024×1024 floats	120	4	10 314 660	81.19	115.29	112.41	3.48
	16×16 blocks of 512×512 floats	816		1 551 322	104.76	238.02	194.28	
	32×32 blocks of 512×512 floats	5984		1 551 322	104.76	238.02	194.28	
QR factorization	16×16 blocks of 512×512 doubles	1 496	4	11 651 079	1 419.33	2 580.41	1 451.74	6.86
Heat diffusion	16×16 blocks of 512×512 doubles	5 124	3	93 198	145.17	748.84	170.00	3.68
Int. Histogram	8×8 blocks of 512×512 floats	2 048	2	514 096	217.45	62.07	263.62	2.23
Bodytrack	native input (851MB)	408 525	6	41 869	93.90	120.93	120.93	4.14



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

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

Fig. 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.

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 [18].

Table 2 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 2.

The average per task overhead for each scheduler is negligible compared to the average task execution time shown in Table 2. 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 2 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 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.

4.3 Real Environment Evaluation

Figure 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 ap-

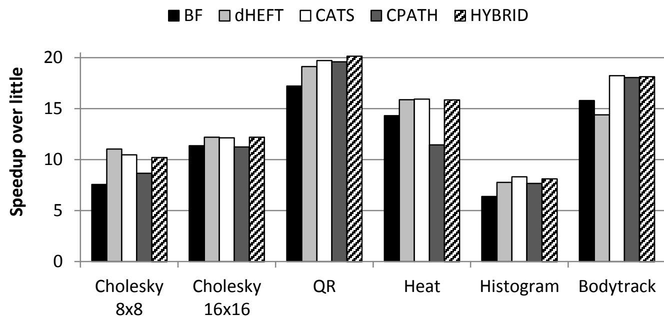


Fig. 6: Speedup of CATS, CPATH, HYBRID, dHEFT and BF on 8 cores compared to the ideal

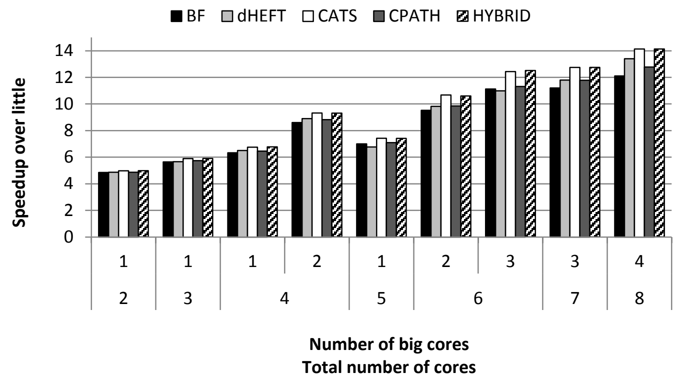


Fig. 7: Average speedups obtained for each scheduler

plications 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 7 shows the average speedup obtained for each scheduler and machine set-up. 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 [13], it shows better performance, although it still does not reach the efficiency of CATS and HYBRID because of its task criticality agnosticism.

Moving in more detail, Figure 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 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 2 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 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 5c. This is the reason why HYBRID gradually outperforms CATS as we increase the number of cores. With a small additional overhead, as Table 2 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,

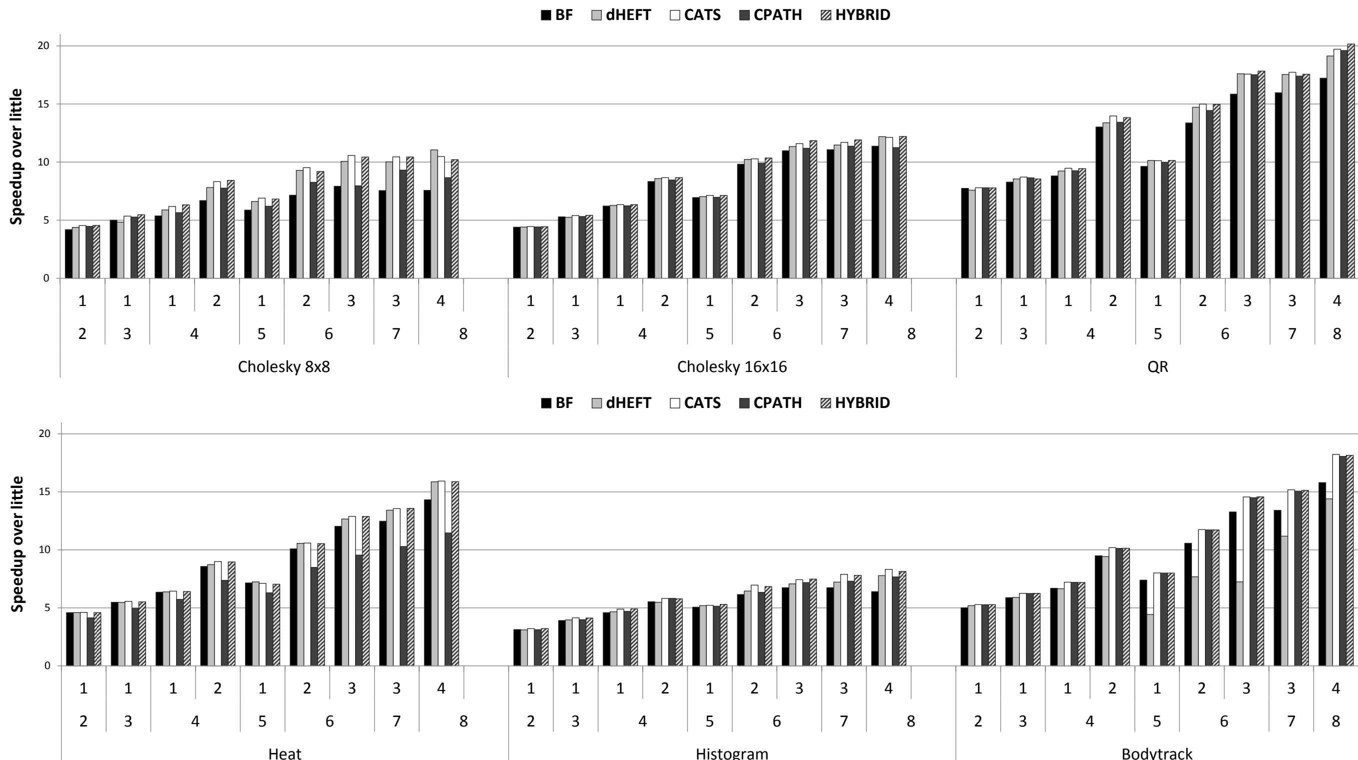


Fig. 8: Speedups obtained for each scheduler and each application

CPATH also reaches a 13% improvement over BF since task cost matters in this case. However, CPATH speedup is still limited compared to HYBRID because of the higher scheduling overheads which in this case is 1.8× higher than CATS overheads. dHEFT also improves BF by finding the earliest executor of each task, but the improvement is limited to 11% which is lower than the other approaches.

This section showed a straight comparison between different heterogeneous schedulers. It is important to note that schedulers like CPATH and HYBRID, that detect the time-based critical path, are the best choices when the application has a large amount of tasks. This is because the additional overheads of these schedulers for critical path computation take place only when there are new tasks on the TDG or when there is a task exit of an untracked 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.

4.4 Simulations

To estimate the impact of the heterogeneity-aware schedulers on larger systems, we run three benchmarks using the TaskSim simulator [16]. 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 9a, 9b and 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 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 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 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.

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 [9], [10], [11], [12], [20], [21], [22] 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 5. The scheduler proposed in this paper is also a list scheduler.

Clustering schedulers [9], [23], [24], [25] 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 [25] 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 [26] group tasks into generations and schedule them according to a randomized genetic technique. Chemical reaction algorithms [27], [28] mimic molecular interactions to map tasks to processors. Some of these guided-random approaches are designed for heterogeneous systems [26], [27]. The scheduler by Page et al. [29] enables dynamic scheduling of multiple-sized tasks for heterogeneous systems, but it lacks support of inter-task dependencies.

Duplication-based schedulers [30], [31], [32] 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 [32] 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 [9], [10], [11], [12], [33]. 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.

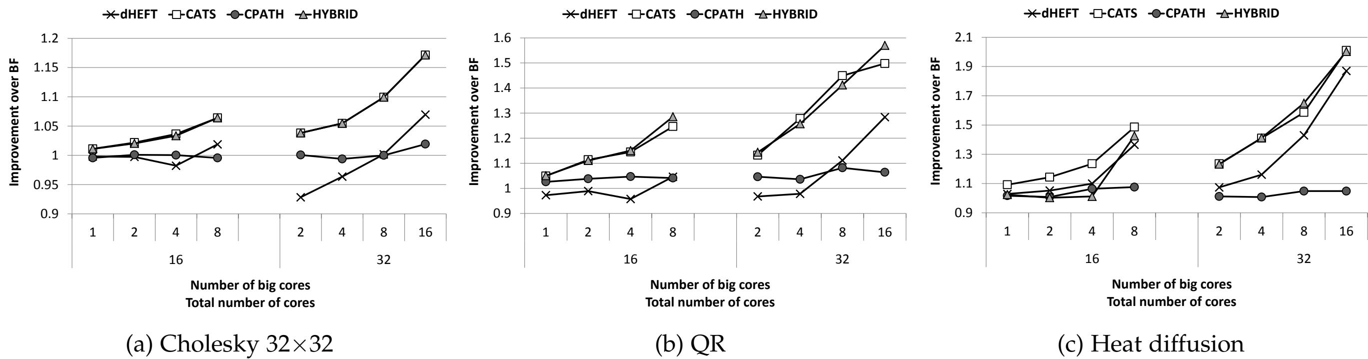
The Heterogeneous Earliest Finish Time (HEFT) algorithm [11] maintains a list of tasks sorted in decreasing order of their upward rank. At each schedule step, HEFT assigns the task with the highest upward rank to the processor that finishes the execution of the task at the earliest possible time. Another work is the Longest Dynamic Critical Path (LDCP) algorithm [10]. 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 [11] 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 necessarely representative of the dependencies in real workloads.

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



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

Fig. 9: Improvement of heterogeneous schedulers over BF for simulated 16 and 32 core heterogeneous systems

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.

Acknowledgments

This work has been supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 610402 and from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement n° 671697. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243).

References

[1] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto, "Maximizing Power Efficiency with Asymmetric Multicore Systems," *Communications of the ACM*, vol. 52, no. 12, p. 48, 2009.

[2] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, pp. 1–8, 2011.

[3] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, E. Ayguade, J. Labarta, and M. Valero, *Runtime-Aware Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 16–27. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48096-0_2

[4] K. Li, X. Tang, and K. Li, "Energy-efficient stochastic task scheduling on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 2867–2876, 2014.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures." *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[6] O. A. R. Board, "OpenMP Application Program Interface ," vol. 4.0, 2013.

[7] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the OpenMP Tasking Model to Allow Dependent Tasks," pp. 111–122, 2008.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011.

[9] M. Hakem and F. Butelle, "Dynamic Critical Path Scheduling Parallel Programs onto Multiprocessors," pp. 203b–203b, 2005.

[10] M. Daoud and N. Kharma, "Efficient Compile-Time Task Scheduling for Heterogeneous Distributed Computing Systems," vol. 1, p. 9, 2006.

[11] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[12] C.-H. Liu, C.-F. Li, K.-C. Lai, and C.-C. Wu, "A dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems," vol. 1, p. 8, 2006.

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

[14] E. Ayguadé, R. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Pérez, J. Planas, and E. Quintana-Ortí, "Extending OpenMP to Survive the Heterogeneous Multi-Core Era," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440–459, 2010.

[15] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP Task Scheduling Strategies," pp. 100–110, 2008.

[16] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero, "On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 36:1–36:20, 2012.

[17] Barcelona Supercomputing Center, "BSC Application Repository," available online on April 18th, 2014. [Online]. Available: {https://pm.bsc.es/projects/bar}

[18] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, and M. Valero, "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite," *TACO*, 2015.

[19] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," Tech. Rep., 2007.

[20] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[21] K. Li, X. Tang, B. Veeravalli, and K. Li, "Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 191–204, 2015.

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

[23] M.-Y. Wu and D. Gajski, "Hypertool: a Programming Aid for Message-Passing Systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 3, pp. 330–343, 1990.

[24] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 9, pp. 951–967, 1994.

[25] M. A. Iverson, F. Özgüner, and G. J. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environment," pp. 93–100, 1995.

[26] H. Yu, "A Hybrid GA-based Scheduling Algorithm for Heterogeneous Computing Environments," pp. 87–92, 2007.

[27] K. Li, Z. Zhang, Y. Xu, B. Gao, and L. He, "Chemical Reaction Optimization for Heterogeneous Computing Environments," pp. 17–23, 2012.

[28] Y. Xu, K. Li, L. He, L. Zhang, and K. Li, "A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3208–3222, 2015.

[29] A. Page and T. Naughton, "Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing," pp. 189a–189a, 2005.

[30] S. Bansal, P. Kumar, and K. Singh, "An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 6, pp. 533–544, 2003.

[31] Z. Zong, A. Manzanares, X. Ruan, and X. Qin, "EAD and PEBD: Two Energy-Aware Duplication Scheduling Algorithms for Parallel Tasks on Homogeneous Clusters," *Computers, IEEE Transactions on*, vol. 60, no. 3, pp. 360–374, 2011.

[32] A. Agarwal and P. Kumar, "Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems," pp. 87–93, 2009.

[33] I. A. Moschakis and H. D. Karatza, "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*, vol. 57, pp. 1–25, 2015.

Kallia Chronaki received the B.Sc in computer science and M.Sc. in computer architecture from the Computer Science Department (CSD) of the University of Crete, Greece. She is currently a Ph.D. candidate at the department of Computer Architecture of the Technical University of Catalonia (UPC), Spain and a research engineer at Barcelona Supercomputing Center (BSC). Her research interests include high performance computing architectures, runtime systems and heterogeneous computing.

Alejandro Rico is a Senior Research Engineer at ARM Research (Austin, TX, USA). Previously, he was a post-doctoral researcher at BSC. He received a Ph.D. from UPC in 2013 and a M.Sc. and B.Sc. from Universitat Pompeu Fabra in 2005. During his studies he worked as an intern at IBM (NY, USA) and ARM (Cambridge, UK). His research interests are high performance computing, multi-core scalability and heterogeneous architectures.

Marc Casas is a senior researcher at the Barcelona Supercomputing Center. Prior to this, he spent 3 years as a post-doctoral fellow at the Lawrence Livermore National Laboratory (LLNL). He received his B.Sc. and M.Sc. degrees in mathematics in 2004 from the UPC and the PhD in Computer Science in 2010 from the Computer Architecture Department of UPC. His research interests are high performance computing, runtime systems and parallel algorithms.

Miquel Moretó is a senior researcher at the Barcelona Supercomputing Center (BSC). Prior to joining BSC, he spent 15 months as a post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from UPC. His research interests include studying shared resources in multithreaded architectures and hardware-software co-design for future massively parallel systems.

Rosa M. Badia holds a PhD on Computer Science (1994) from UPC. She is a Scientific Researcher from the Consejo Superior de Investigaciones Científicas (CSIC) and team leader of the Workflows and Distributed Computing research group at BSC. Her research interests are programming models for complex platforms (from multicore, GPUs to Grid/Cloud). Dr Badia has published more than 150 papers in international conferences and journals in these topics. She has participated in a significant number of European funded projects and contracts with industry.

Eduard Ayguadé is full professor of the Computer Architecture Department at UPC. He is currently associate director of research in Computer Sciences at BSC. His research interests include multicore architectures, programming models and compilers for high-performance architectures. He published around 250 publications in these topics and participated in several research projects with other universities and industries, in framework of the European Union programmes or in direct collaboration with technology leading companies.

Jesus Labarta is full professor on Computer Architecture at UPC since 1990. Since 2005 he is responsible of the Computer Science Research Department within BSC. His major directions of current work relate to performance analysis tools, programming models and resource management. His team distributes the Open Source BSC tools (Paraver and Dimemas) and performs research on increasing the intelligence embedded in the performance analysis tools. He is involved in the development of the OmpSs programming model and its different implementations for SMP, GPUs and cluster platforms.

Mateo Valero is full professor at Computer Architecture Department, UPC and director at BSC. He has published 700 papers and served in organization of 300 international conferences. His main awards are: Seymour Cray, Eckert-Mauchly, Harry Goode, ACM Distinguished Service, "Hall of Fame" member IST European Program, King Jaime I in research, two Spanish National Awards on Informatics and Engineering. Honorary Doctorate: Universities of Chalmers, Belgrade, Las Palmas, Zaragoza, Complutense of Madrid, Granada and University of Veracruz. Professor Valero is a Fellow of IEEE, ACM, and Intel Distinguished Research Fellow. He is a member of Royal Spanish Academy of Engineering, Royal Academy of Science and Arts, correspondent academic of Royal Spanish Academy of Sciences, Academia Europaea and Mexican Academy of Science.