An orchestrated approach to efficiently manage resources in heterogeneous system architectures

C. Bolchini, G. C. Durelli, A. Miele, G. Pallotta, M. Pogliani, M D. Santambrogio

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Milano, Italy
Email: cristiana.bolchini@polimi.it, gianluca.durelli@polimi.it, antonio.miele@polimi.it, gabriele.pallotta@mail.polimi.it, marcello.pogliani@mail.polimi.it, marco.santambrogio@polimi.it

Abstract—Nowadays, we are witnessing trends in technology, fabrication processes and computing architectures that lead to the design and development of processing systems constituted by a relevant number of independent, heterogeneous execution resources. The aim is to achieve high-performance while leveraging on other aspects, such as energy consumption. Indeed, heterogeneity comes at the cost of greater design and management complexity. To reach an optimal solution, system architects need to take into account the efficiency of systems’ units, i.e., general purpose processors eventually with one or more kinds of accelerators (e.g., GPUs or FPGAs), as well as the workload. This often leads to inefficiency in the exploitation of such resources, and therefore in performance/energy. Within this context, we are proposing a runtime resource manager able to observe the system execution and to dynamically optimise its behaviour with respect to one or more identified functional parameters, according to the architectural characteristics, and the users’ and the applications’ needs. Such an adaptation characteristic is intrinsically embedded in the device as a software layer, called Orchestrator, able to adapt the runtime resource management according to the target objectives and to the inputs from the external environment.

I. INTRODUCTION

High performance systems are nowadays designed to serve rather static workloads with high-performance requirements, although we are moving towards a highly flexible, on-demand computing scenario that is characterised by varying workloads, constituted by diverse applications with different performance requirements, variable amount of data to be elaborated and unknown arrival times. A promising approach to address the challenges posed by this scenario is to better exploit specialised computing resources integrated in a heterogeneous system architecture such as asymmetric multicore CPUs, specialised graphic co-processors, GPUs, or reconfigurable HW such as FPGAs. Exploiting these new heterogeneous architectures means taking advantage of their individual characteristics to optimise the performance/energy trade-off for the overall system. However this increase in heterogeneity comes at the cost of greater management complexity at both the application and the system level. System architects need, as an example, to consider the efficiency of the different computational resources as well as the application workload, which often leads to inefficiency in resource exploitation and, therefore, a suboptimal performance/energy trade-off [1]. Furthermore we can expect that, in the forthcoming years, more and more companies will drive their businesses on the basis of complex data analysis, as testified by the Big Data phenomenon. We can then foresee scenarios where companies will rely to utility (i.e., cloud) computing to offload data analysis jobs instead of sustaining the cost of building and maintaining their own private computation cluster. This poses new challenges in platform resource management, further exacerbated by the need for run-time power budgeting and by the increased dynamics in workload behaviour observed in High Performance Computing (HPC) and multicore System-on-Chip (MPSoC).

As a result, the working scenario consists of a highly dynamic behaviour in which workloads wish to freely utilise the multitude of accelerators made available by the underlying heterogeneous computing platform. In fact, complex System-on-Chip solutions combine multiple general-purpose processors, which provide flexibility, with on-chip network and custom accelerator cores (such as GPUs and FPGAs) for performance and efficiency. However these devices must be manually managed by the application programmer, not only for directly executing the code, but also for other system related tasks such as their allocation and de-allocation. For instance deploying jobs on a HPC cluster requires to specify and allocate beforehand all the needed resources in a static manner, usually leading to resource over-provisioning. Consequently, this increases costs for the user, wastes power and causes resource under-utilisation because the provider has to keep all the reserved resources up and ready to run jobs. Therefore, we believe that the traditional paradigms typically adopted for the design and exploitation of embedded as well as HPC systems are becoming outdated, and need be revisited to allow for an actual exploitation of these technologies. In particular, we argue that this is the time for a fresh approach to the way systems are designed and used, devoting part of the increased computational capability to make the system self-aware, balancing the use of accelerators’ resources to improve performance, utilisation, dependability and programmability. Our proposed approach mostly relies on self-adaptive systems that exploit runtime measurements to foresee how the system will react to situations not predictable at design time.

Within this work we focus on the definition of a runtime mechanism, called Orchestrator, that enables heterogeneous systems to manage easily and efficiently accelerators to meet application and system goals, such as high performance for a wide range of workloads. The Orchestrator leverages the new opportunities presented by the presence of heterogeneous
accelerators in HPC and embedded systems, treating them as first class scheduling entities by supporting self-aware scheduling methods that enable their efficient use. The Orchestrator observes the running applications through a monitoring API that continuously gathers information both on their performance and the system power consumption. This information is forwarded to the Orchestrator and to the Processing Element Managers (PE-Managers) that are responsible for adapting the underlying computational resources to meet applications and system objectives (expressed in terms of goals and constraints). Each PE-Manager controls a set of actuators, specific to that computational resource, to optimise a given goal through a user-defined policy. The output of this decision is an action carried out by an actuator to obtain the desired effect.

The paper will first present the overall architecture of the Orchestrator, and then will propose a possible implementation in a specific scenario considering a system with a multicore processor plus a GPU, and a cluster of hardware accelerators.

II. RELATED WORK

Several on-going projects are working towards the definition and development of self-aware/adaptive systems, able to adapt themselves to accomplish a given goal despite changing environmental conditions and demands. There are several existing solutions in this field, for example for meeting power, performance, and resource-metering challenges in cloud computing (e.g. [2], [3]), large-scale datacenters (e.g. [4], [5]), or for exploiting heterogeneous system through a predictive user-level scheduler based on past performance history [6].

The last recent years have seen a few initial studies in the design of adaptive operating system’s components (e.g., [7]–[9]). SEEC [7] provides an interesting and complete framework embracing machine learning and control theoretical solutions, but is not integrated in the operating system and does not support heterogeneous system architectures. BarbequeRTRM [8] is the key element of a highly modular and extensible run-time resource manager; however it mostly employs static partitioning of workloads and coarse-grain parallelism, assuming a platform where no other task is running. Although it provides a sort of an orchestrator mechanism, it does not self-adapt at run-time, which is one of the fundamental challenges we are tackling. Metronome [9] extends GNU/Linux to manage at runtime the core and the CPU time allocation; so far it is based on heuristics and it uses just one metric to measure performance in terms of applications throughput. Moreover, there are a few approaches in literature considering heterogeneous architectures; PTask [10] is a kernel-level abstraction for managing GPUs. CHIMERA [11] presents an architectural solution, with external accelerators, presently characterised by too high a development time, with the technology mapping between the applications and the heterogeneous resources computed at design time. BORPH [12] is an extension of the Linux kernel for run-time support to FPGAs. These are some of the effective solutions for promoting heterogeneous computing systems and constitute important milestones. However, they still lack a comprehensive, autonomic operating systems and run-times, such as the K42 research operating system [13], which makes monitoring and adaptation first-class citizens. Taking advantage of approaches such as PTask, while leveraging on autonomic computing (K42) BORPH could enable a new class of solutions capable of seamlessly employing processors and specialised islands of computation, to achieve full utilisation and high performance with limited effort for developers and maximal benefits. K42 provides a comprehensive support for both monitoring and adaptation; however, it lacks support and formalism when it comes to decision-making. AQusa [14] is an extension of the Linux kernel comprising a run-time to enable reasoning and formal decision-making in operating systems. AQusa provides a control theoretical framework to allow soft real-time applications meeting user-specified Quality of Service (QoS) requirements through adaptive CPU reservation. AQusa only supports adaptive CPU reservation and can provides adequate support to soft real-time applications, only. Indeed, this covers a limited scenario because specialised islands of computation are not considered and most commodity applications are not real-time, but they would still benefit from AQusa reasoning and decision-making capabilities. Finally a first solution for asymmetric processors have been implemented in few Linux distributions. For instance the Linaro distribution used in ARM big.LITTLE platforms provides an ad-hoc scheduling policies for asymmetric processors, named Global Task Scheduler (GTS). GTS is aware of the availability of asymmetric cores; at runtime it profiles the running tasks and migrates the most computational intensive ones to the big cores, which delivers more performance, and the least intensive one to the LITTLE cores to gain in energy efficiency. However no management of GPUs or other HW devices is available in this solution.

III. A BIRD’S EYE VIEW ON THE PROPOSED ORCHESTRATOR MECHANISM

The Orchestrator is an entity in charge of receiving requests from applications to be executed, and to choose the optimal among the available computing resources, trying to meet the requirements issued from each application with system-level constraints. The Orchestrator relies on the concept of feedback loops to monitor the status of the system and minimize the difference between the desired goals and the current system status. To achieve this control, the Orchestrator consists of different entities that work together to make the system evolve towards the desired state. In particular, starting from the concept of Observe-Decide-Act (ODA) loop, the Orchestrator is designed to be composed by components that collect measurements on the system, components participating in the decide phase, and finally entities in charge of acting on the system status. Figure 1 illustrates the outline of the system under development. We can identify three main components in the system: the hardware resources, the applications, and the Orchestrator.

1) Hardware Resources: The system we are targeting is a massively heterogeneous system composed by different instances (organised in clusters) of CPUs, GPUs and DFEs1.

1Data Flow Engine (DFE) is an architecture based on FPGA devices used as hardware accelerators developed by Maxeler Technologies [15].
Goals and Constraints

The Orchestrator about its goal and constraints. Non-Heartbeat orchestration is in charge of managing the execution of a workload composed of multiple applications, each one with his own goals and requirements. We define *adaptive applications* those applications that expose an enhanced interface to be accessed/monitored by the Orchestrator; therefore they need to be written taking advantage of the APIs exposed by the Orchestrator itself. For example, an application can expose its Heartbeat to communicate its throughput, and can inform the Orchestrator about its goal and constraints. Non-adaptive applications can be run as well on the system, but they cannot be fully managed by the Orchestrator, because they do not provide means to close the feedback control loop on the application side.

2) Applications: The Orchestrator controlling the heterogeneous architecture is in charge of managing the execution of a workload composed of multiple applications, each one with their own goals and requirements. We define *adaptive applications* those applications that expose an enhanced interface to be accessed/monitored by the Orchestrator; therefore they need to be written taking advantage of the APIs exposed by the Orchestrator itself. For example, an application can expose its Heartbeat to communicate its throughput, and can inform the Orchestrator about its goal and constraints. Non-adaptive applications can be run as well on the system, but they cannot be fully managed by the Orchestrator, because they do not provide means to close the feedback control loop on the application side.

3) Orchestrator: The Orchestrator component is responsible for the management of the heterogeneous resource available on the underlying architecture. It has all the information regarding the system status, and in particular we highlighted in Figure 1 the following monitoring capabilities (detailed in Table 1):

- **Application Status:** monitors the performance of the applications collecting and aggregating the information exposed by the adaptive applications.
- **Goals & Constraints:** contains the information on the goals that the system have to fulfil and the constraints that must be met at both application and system level. Constraints are in the form of hard limits that are enforced on some resource; goals identify the objective of the system.
- **Resource Status:** collects the information regarding the available hardware. Different hardware resources provide the information through different interfaces and the information must be presented to the Orchestrator in a common form. In this context the Manager Interface provides a standard way for the Orchestrator to collect this information from the different Managers that have direct access to the hardware (see Section IV for further details).

Starting from this information the *Decision Mechanism* takes care of automatically determining the mapping of the applications to the computing resources and the constraints to be given to the specific managers. The constraints for the managers are expressed in terms of resource budgets (power and temperature) that the manager has to respect while controlling the assigned applications. Before delving into the details of the orchestration mechanism, Table I summarizes the information monitored by the Orchestrator organizing them with respect to the classification represented in Figure 1: Application Status, Goals & Constraints, and Resource Status.

### A. Manager Interface

The connection between the Orchestrator and the Managers is provided by an interface which the Managers implement and that exposes to the Orchestrator the possibility to control and fetch information on the underlying architecture. Through this interface, the Orchestrator can assign each manager the applications to manage: when the Orchestrator decides that an application has to be mapped on a specific HW resource, it passes its handler to the Manager responsible for its execution. Through the handler, the Manager will be able to monitor the application performance and adapt the HW resources under its direct control to achieve application-specific requirements and goals. At the same time, this interface allows the Orchestrator to set constraints on the amount of resources that the Manager can exploit. As an example, the Orchestrator can limit the power that a Manager can consume or the energy that it can...
dissipate over a certain period. The manager will be subject to these constraints in trying to satisfy the applications’ requests. Finally, the Manager will provide the Orchestrator, through the Manager interface, relevant information on the HW resources under its direct control as for instance the power and energy consumption.

B. Orchestrator Mechanism

This section delves into the details the information exchange between the Orchestrator and the other resource managers, later presenting the designed control mechanisms as seen from the application point of view.

1) Orchestrator and Managers: A single manager is available for each cluster of hardware resources, leading to an architecture with an Orchestrator and one or more Managers, that will work together in an ODA-loop to manage the systems.

The Managers will expose the state of the hardware resources and the state of the application assigned to them. With this information the Orchestrator is always aware of the state of the entire system at a given time, and can make educated decisions to modify it to meet violated constraints or adapt to new workloads. Once the Orchestrator takes a decision, the Managers will be notified with the action the Orchestrator decided to take along with the new imposed constraints and, eventually, the new applications to be run.

Locally, the Managers will try to take the best decision to distribute the applications assigned to them by the Orchestrator on their available resources. Two possible scenario may occur: (i) the Manager successfully maps each application, respecting all constraints, or (ii) the Manager is not able to map the application respecting all the requests.

In the first case we should have an optimal use of all the resources, also being able to serve other new applications. In the second one, the Manager sends a negative feedback to the Orchestrator, that has to address the problem. The Orchestrator decisions are taken based on a priority system on the system constraints. For example, if a Manager reports a violation that can be solved by increasing the available power budget, the Orchestrator can evaluate whether or not to enforce the increment. Otherwise, if the power cannot be further increased, as we assume that each Manager takes the best decision with respect to the imposed constraints, the Orchestrator will not modify the system.

Note that the control loop of the Orchestrator is aimed at performing application dispatching and at instructing the managers on their available budgets. In fact, given the nature of our distributed system, it will be infeasible to rely on one component only in order to make decision about application distribution, hardware resource management, and system monitoring. Since we already described the Orchestrator as the decision maker at a system level, it will be inefficient to include information about each hardware component and how to manage it. This task will be left to a set of ad-hoc Managers further described in Section IV.

2) Application and Orchestrator: An application, in order to be controlled by the Orchestrator, needs to implement an API that allows information exchange with the Orchestrator. In particular this API allows the application to express constraints, goals and communicate progresses to monitor its performance (as described in Table I). At the same time the API will allow the application to communicate possible knobs that the Orchestrator can control; a simple example of this knob is the possibility to execute the code on different resources such as CPUs and/or GPUs. An application can communicate to the Orchestrator the possiblity to be run on different resources, and the Orchestrator is in charge of monitoring its progress and deciding how to tune this heterogeneous execution knob (i.e. deciding on which resources the application will execute) to achieve the goals and fulfill the imposed constraints. The overhead of instrumenting an application to be compliant with the proposed system can be estimated to be around ten lines of code. A description of this interaction is reported in Figure 2.

![Fig. 2. Representation of the interaction between the Orchestrator and the applications.](image-url)

IV. Computing Resource Manager

The Manager is the component responsible of the hardware resources. It exposes all the information about its hardware resources, as well as the information about the applications currently running. Managers represent the actuators of the Orchestrator’s ODA-loop. These entities are needed as they encapsulate the specific hardware information, and have the knowledge to make the best run-time decision. Without them, this task would be fulfilled by the Orchestrator, resulting in huge complexity and possibly slowing down the decision time.
The managers are the components that know and have full access to the hardware they control. The mapping between managers and hardware resources is not one to one; each manager is in fact in charge of orchestrating a pool of homogeneous resources. In our vision, it is possible to identify three kinds of manager, depending on the number of clusters of components available on the specific architecture: (i) CPUs Manager, (ii) GPUs Manager, and (iii) DFEs Manager. Due to the different nature of the hardware resources, each manager is implemented differently and will have access to the specific knobs exposed by the computational cluster.

All the managers internally implement an ODA loop, for managing only the applications the Orchestrator assigns to them, monitoring their performance and the status of the hardware they control with the possibility to tune specific hardware knobs. This two-layer ODA loop allows to design simpler policies for the managers and the Orchestrator and will allow an easy extension and introduction of new managers since the control of hardware mechanisms is decoupled from the Orchestrator activity. More details on the mechanism and the specific managers are given in the following sections. In particular, the CPUs and the DFEs managers are here discussed, in line with their prototype implementation, while the GPUs manager is still under investigation.

A. Basic Manager Mechanism

The Managers are the actuators of the Orchestrator. When the Orchestrator chooses to run some applications on certain hardware resources, the applications are sent to the corresponding resource manager. At this point the managers will try to take the best decision to distribute the applications assigned to them on their available resources. Similarly to the Orchestrator, managers leverage an ODA-loop to react to changes in the system. In particular, a Manager Observes the behaviour of the system, composed of the hardware resource utilisation and how well applications are performing. Based on those observations and the directives imposed by the Orchestrator, it can Decide how to Act in order to rearrange its workloads to fit the constraints. In order to be monitored by the Orchestrator, each Manager must expose all the needed information about the hardware status, such as, consumed power, frequency of each node, and voltage. Also, it has to expose all the information about the applications currently running on their resources and how well they are performing.

B. CPUs Manager

The actions a CPU Manager can take are mainly three: (i) Task Mapping, (ii) Dynamic Voltage and Frequency Scaling, (iii) Idle cycle injection. Task Mapping is a technique that assigns an application to a specific CPU core. This capability is very important as it allows to balance the load on multiple CPU systems we are targeting, and it can also be used to modify and control the power drawn by the CPUs. Task mapping functionality is usually provided by the operating system scheduler. For example, the Linux kernel allows to pin a task (i.e., a process or thread) to a particular set of CPUs through the affinity mask, a per-process structure the scheduler uses to determine on which CPUs a task can run, using the sched_setaffinity() system call.

Dynamic voltage and frequency scaling (DVFS) is a commonly-used power management technique, where clock frequency and voltage of a processor are decreased to allow a reduction in the power consumption. As the power consumption in a CMOS circuit is proportional to $fV^2$ (where $f$ is the circuit frequency and $V$ is the voltage), power consumption is greatly reduced; this can lead to significant reduction in the energy required for a computation, particularly for memory-bound workloads. Most operating systems already exploit DVFS as part of their power management architecture. Furthermore, most of them provide interfaces for users or applications to manually control the DVFS operating point of the system processors: for example, under Linux, DVFS-related settings are accessible from the user space through the sysfs file system, and the cpufrequtils package provides higher-level utilities to manage them.

Idle cycle injection is a technique that forces the CPU to idle in order to prevent overheating, or to cap its power to not trespass a certain limit. Modern processors have different low-power states, called C-states in the case of Intel processors, which differentiates for their power consumption levels and resume time. The low power states were originally implemented to reduce the system power consumption during idle periods, and the operating system scheduler triggers the low power states when there is no ready process to run. The idle cycle injection technique consists in triggering processor C-states for a short time, even though the system has ready tasks to run, with the goal of lower the CPU power consumption or controlling the operating temperature. Recently, Idle Cycle Injection technique gained popularity both in the context of temperature management [16], [17] and as a generic power management technique. To this extent, Intel released in 2012 the PowerClamp Linux kernel module [18], which keeps the idle percentage of each processor (percentage of cycles where the processor is in a C-state) as near as possible to a value set by the system administrator or by an external controller.

C. DFEs Manager

The DFE Manager acts as a controller over a cluster of DFEs. Maxeler Technologies [15], which is the producer of this architecture, provides computing machines mounting 8 DFEs, similar to 8 FPGA boards used as hardware accelerators. These DFEs are connected to multiple host devices that can dispatch jobs to them by means of an Infiniband connection. The design flow to target these systems is also proprietary and provides an high-level Java-like description to target hardware design and builds upon the classical Xilinx and Altera hardware design flows. The host can send jobs to the DFEs in form of actions, where an action is defined as an atomic entity characterized by a computation to do (i.e. a configuration bitstream) and an input dataset. The dispatching is not performed directly to the target DFE, but it is rather virtualized through the concept of a Group. The application in fact send actions to a specific group which can be mapped on a variable number of DFEs at runtime.
The DFE Manager in this context acts as an interface between the DFEs and the Orchestrator that dispatches actions and controls how the group maps to physical DFEs. The actions it can take are: (i) vDFE resize; (ii) frequency management. By vDFE resize, we mean the ability of the DFE manager to vary at run-time the number of physical DFEs allocated to each group in order to tune the performance of an application. To vary the resources at run-time, the DFE manager exploits the abstraction of a virtual DFE (vDFE). A vDFE is a group of DFEs that are presented to the application as a single entity. When receiving a new job submission, the DFE manager creates a new vDFE backed by a number of physical DFEs variable at run-time, and communicates the identifier of the vDFE to the application; then, the application submits the computation to this vDFE.

The assumption for an application to be profitably managed by the DFE manager is to be composed by a multitude of independent atomic stateless computation units (actions) that can be transparently dispatched to different DFEs without any guarantee on the actual physical DFE that each action will use. Furthermore, as a physical DFE can be removed from a vDFE only after the currently running action finishes, the manager assumes that the duration of a single action is small with respect to the total duration of a job. In any case, the Orchestrator may decide to dispatch to the DFE manager jobs that do not completely satisfy those assumptions: in this case, it can reserve for the job a fixed number of physical DFEs, that are then considered as a fixed constraint when deciding the size of the vDFEs used by other applications.

A metric of particular interest for the manager is the number of completed actions for each vDFE: this allows the manager to compute the current throughput and, under the assumption that the total number of actions is approximately known, the progress rate of the job towards its completion.

Different policies can be devised, according to the type of job running in the system, for instance we designed: an earliest deadline first policy and a throughput-oriented heuristic. On one hand, an earliest deadline first policy provides good results in case of batch jobs, when all the computation results are needed only after the job deadline and all the job deadlines can be fulfilled. This policy assigns all the available DFEs to the job with the earliest deadline, possibly changing the allocation if a new job with an earlier deadline is subsequently submitted. The advantage of this policy is that the number of reconfigurations is minimised, as they can happen only when a job retires or when a new job is submitted. On the other hand, if the system is mostly running jobs where intermediate results are constantly needed, e.g., a streaming application such as real-time video encoding, it is important for the application to sustain a certain throughput instead of executing all the actions in a short time. In this case, a different heuristic has been developed, called throughput-oriented heuristic. It exploits a feedback loop to try and keep the job throughput (defined as the number actions completed per second) in a certain range.

V. RESULTS
This section presents a working example where the proposed Orchestrator has been used to manage runtime resource allocation while running various kinds of workloads. The results have been collected on two different heterogeneous platforms. The first one is a workstation equipped with an Intel Core i5 750 dual core CPU running at 2.7 GHz (with Hyperthreading enabled), and an Nvidia GeForce GT 240 GPU capable of executing OpenCL and CUDA kernels. The second one is a system targeting High Performance Computing (HPC) workloads featuring a cluster of 8 DFEs attached by means of an Infiniband connection to a host machine used to run the tests. We instrumented a set of 5 applications from different domains and we made them compliant with the Orchestrator. The applications include a pricing option application based on the Black and Scholes formula, a Biomedical application that extracts a brain network analysing the correlation between pixel in a brain image, two image processing applications that perform an image twisting and a motion detection benchmark, and finally a simple benchmark to square the values of an array. In the following we illustrate how the Orchestrator is able to dispatch kernels to the different available resources and how the managers can control the workload on different computational clusters; finally we investigate the overhead of the proposed solution. Note that, even if the GPU manager is still under development, it is possible to use any GPU available in the system by relying on the standard operating system behaviour, although without the opportunity to take any specific actions.

A. CPU and GPU Management
When using the first heterogeneous platform, all selected applications have been instrumented with the capability to register to the Orchestrator and to be able to express a goal in terms of throughput. Furthermore each one of the applications has a computational intensive kernel that has been implemented using both OpenMP and OpenCL to exploit the multithreading and the heterogeneous capabilities of the target machine.

Let us assume that two users are interested in performing a pricing options estimation on a set of stock options, having access to the same computational node to perform such analysis. One of the two users might want a higher priority for his/her analysis, thus setting a higher throughput requirement. In this scenario, the system has to run two instances of the Black and Scholes application with two different throughput requirements. The first application arrives at time 0s, with a throughput requirement of 0.7 Heartbeats/s, the second one arrives at 10s, with a throughput requirement of 2.1 Heartbeats/s. Figure 3 illustrates how the classic Linux scheduler is unable to manage this kind of situation, with applications having different constraints and, by design, tries to allocate the available resources between the two applications. In fact, while running in co-location, the two instances of the benchmark achieve the same performance, which is half of the throughput a single instance of the application can achieve (as we can see at the start of the application).

The Orchestrator runs a policy that decides where to dispatch the kernel on the basis of estimated performance. The estimation is carried out at runtime and continuously updated
thanks to the online monitoring capability of the system by means of an Heartbeat-like API [19]. Once an application arrives, the Orchestrator schedules it on each of the available computational cluster (i.e. CPU and GPU) for a given number of iterations, and uses the performance estimations collected in this profiling period to decide in the future where to dispatch the application. After this profiling phase, the managers that can actuate a finer tuning of their controlled resources directly control the application. At the moment, we implemented a CPU manager that controls the number of cores allocated to the application; such a decision is taken on the basis of the application current throughput and follows the policy described in [20] and uses the taskset utility to partition the cores between running applications. The CPU manager itself adopts an Observe-Decide-Act (ODA) control approach to manage the applications that the Orchestrator dispatched to it. This combined management approach helps in solving resource management problems as the one highlighted before.

We performed a first test where we replicate the above scenario with the possibility for the CPU manager to control the CPUs available in the system. In this case the Orchestrator is not performing any profiling and it dispatches straightforwardly the application to the CPU manager that dynamically controls the number of resources allocated to the applications on the basis of their performance requirement. Figure 4 illustrates the obtained results. As we can see, both applications reach the required throughput during the whole execution. More precisely, when the second application arrives at 10s the manager distributes the cores such that the two requirements are met (3 cores to Application 1 and 1 to Application 0). When there is no contention (i.e. only one application in the system) the CPU manager assigns all the resources to one application.

Finally Figure 5 presents the results of a complex scenario where all the applications we instrumented are executed with the possibility to be run on the CPU or the GPU and with both the Orchestrator and the CPU Manager active. Here, when an application arrives it gets profiled online for a given number of iterations on each of the resources and then the Orchestrator decides which cluster the application has to be dispatched to. In Figure 5 we see that all the applications (except for Square, which can be run on the CPU only) has quite a long period with low performance. In this period the Orchestrator is profiling the application on the CPU, learning that the performance is not sufficient to meet the requirement. The application is then profiled on the GPU cluster, where the Orchestrator learns that the performance is sufficient to meet the requirement. In the meanwhile, the CPU Manager is orchestrating the assignment of the cores to the applications it manages (i.e. the two Square applications), receiving an amount of resources proportional to their throughput requirements. Note that the profiling phase is executed once for each application at the beginning, and then the obtained estimation is adjusted at runtime through online information using an exponential moving average.

B. DFE Management

Although the DFE manager complies with the rest of the system, it is hard to find applications that benefit from having different implementations. The reason behind this is that the development of solutions for these platforms is still a complex task and there is no return on investments except when the application is well suited for such computational paradigm. In this context, then, the applications implemented for DFEs will outperform any other implementation and do not justify the adoption of heterogeneous implementations for the same kernel, as in the previous example. Nonetheless the same computational cluster has to be opportunely managed to guarantee the achievement of the desired performance, because multiple applications can be co-located on the same architecture. Thus, the DFE manager implements a policy that allows to achieve a user-defined requirement in terms of throughput or deadline for a given application. The behaviour of the DFE manager when three instances of the Black and Scholes application (implemented for the DFE) are executed on a cluster of 8 DFEs is represented in Figure 6. All the applications execute with different deadlines and the DFE manager is able to monitor at runtime the performance of the application and determine the correct allocation of the 8 DFEs to the applications to meet their deadlines. We recall that, in this case, meeting a deadline means that the throughput of an application at the end of its execution reaches the desired target throughput.
C. Overhead Analysis

The Orchestrator runs as a standalone process in the system and periodically executes to determine the best allocation and resource partitioning for the applications running in the system. In order to evaluate the introduced overhead we tested our benchmark applications with and without enabling the Orchestrator, and collected their execution time. To obtain a fair comparison between the two situations we enabled both the Orchestrator and the CPU manager, which allows to allocate the right number of cores to the application, although we did not actuate the suggested action. In this way the applications will always execute with the maximum number of threads, while at the same time the Orchestrator will perform the same amount of computation as in the real case. Figure 7 presents the result of the test for each one of the adopted benchmark applications. As the figure shows, the overhead is relatively small and it varies from 0.1% for Motion Detection to 2% for Black and Scholes.

VI. Conclusions

In this paper we presented the key features of the Orchestrator, a runtime resource manager for heterogeneous architectures, and we illustrated the behavior of the system in controlling diversified applications running in a multiprogrammed environment through some examples. Results show the ability of the Orchestrator to control this kind of applications to meet the goals expressed by the final user. A key point of the research is the use of high level metrics in the monitoring infrastructure: this allows the user to easily express goals on the applications, and the Orchestrator to clearly understand whether the goals are met or not. The system is well suited for managing parallel workloads and can be deployed on desktop workstations and nodes of a computer cluster to control the execution of applications consolidated on a single node. Future researches will be focused in using the devised Orchestrator to analyze and implement runtime management policies that optimizes power and energy related metrics which are of utmost importance in heterogeneous architectures.

ACKNOWLEDGMENT

This work has been partially funded by the European Commission in the context of the FP7 SAVE project (#610996-SAVE).

REFERENCES