Runtime Resource Management in Heterogeneous System Architectures: The SAVE Approach

Gianluca C. Durelli[1], Marcello Pogliani[1], Antonio Miele[1],
Christian Plessl[3], Heinrich Riebler[3], Marco D. Santambrogio[1], Gavin Vaz[3], Cristiana Bolchini[1]

[1]Politecnico di Milano, Italy,
marcello.pogliani@mail.polimi.it, {firstname.lastname}@polimi.it

[3]University of Paderborn, Germany
{firstname.lastname}@uni-paderborn.de

***Abstract*—Modern computing systems featuring different kinds of processing elements have proven to be efficient in terms of performance/energy trade-offs. Furthermore these systems usually have to execute multiple concurrent tasks without any a-priori knowledge on expected arrival times, in an unpredictable and very dynamic environment. This scenario has propelled an interest towards self-adaptive systems that dynamically reorganize the use of system resources to optimize for a given goal. The SAVE project will develop a Heterogeneous System Architecture that will decide at runtime to execute task on the appropriate kind of resources, based on the current requirements. This paper presents a first implementation of a resource allocation policy that dynamically shares heterogeneous resources between multiple running applications. Resource allocation mechanisms are discussed and evaluated in an experimental campaign, showing how the policy helps in attaining users' applications goals.**

***Keywords*—Heterogeneous systems, Virtualization, Resource Allocation**

I. INTRODUCTION

The early computer science literature is filled with a plethora of static optimization approaches, among which the most prominent example of a system applying static optimization is a compiler. A compiler applies a set of common transformations to an application so as to speed up its execution on an entire family of microprocessors (e.g., x86). Whenever a compiler is given additional information, it can harness more aggressive transformations to obtain additional performance improvements on a subset of a family of microprocessors.

The driver of static optimization is the availability of information, which may be scarce depending on the environment. On the one hand there are embedded computing systems that usually perform the same task over and over; they represent the perfect scenario to apply the highest level of static optimization so as to maximize the benefits for users (e.g., maximize performance while minimizing power consumption). On the other hand there are clusters of computing systems that may execute multiple tasks simultaneously without the possibility to anticipate the startup of a task and the finishing of another. Furthermore, the increasing availability of different kinds of processing resources in Heterogeneous System Architecture (HSA) associated with today's fast-changing, unpredictable workloads (e.g., of mobile or cloud-computing contexts), has propelled an interest towards self-adaptive systems that dynamically reorganize the usage of system resources to optimize for a given goal (e.g., performance, energy, reliability, resource utilisation). This scenario calls for dynamic optimization.

The SAVE (Self-Adaptive Virtualisation-Aware High-Performance/Low-Energy Heterogeneous System Architectures) project will develop a stack of hardware, software and OS components that allow for deciding at run-time to execute tasks on the appropriate type of resource, based on the current system status/environment/application requirements. We claim that dynamic (i.e., runtime) optimization is key to our project, which deals with changing requirements and unpredictable environments. For the sake of simplicity let us consider a realistic example where many users run their applications simultaneously in a cloud computing infrastructure. The goal of each user is to get the best out of the infrastructure (finish his/her computation respecting time constraints, if any) while the goal of the administrator may be to minimize the total cost of ownership. If a single user is running his/her application the best strategy can be allocating the highest amount of resources the application can use efficiently so as to idle as soon as possible, thus decreasing power consumption. If multiple users are running their applications simultaneously the best strategy could be completely different depending on how applications interfere with each other, etc. In such a scenario, static optimization cannot deal with changing requirements and unpredictable environments due to the lack of essential information that becomes available only at runtime. Nevertheless, static optimization can be a useful starting point, to be combined with runtime policies.

The project outcome will be an improved HSA with self-adaptation providing not only runtime reaction to changes, but also the means to dynamically achieve optimization goals based on the current context.

II. RELATED WORK

Achieving efficient self-aware system operation while satisfying specific constraints is critical for modern computing systems, ranging from SoCs to complex High Performance Computing (HPC) solutions. This is especially true in the presence of technological process variability and workload

variations caused by the non-deterministic nature of applications. IBM has introduced the Autonomic Computing initiative in 2001, with the aim of developing self-managing systems [1]–[3]. With the growth of the computer industry, notable examples being highly efficient networking hardware and powerful CPUs, autonomic computing constitutes an evolution to cope with the rapidly growing complexity of integrating, managing, and operating computing system. Within this context, classical reconfigurable and multicore systems will shift towards self-aware computing systems; virtualized environments, hardware components, applications and operating systems [4] will holistically adapt their behavior to optimize selected metrics. The availability of operating systems (OSes) and runtime support for coprocessors (e.g., the Intel Xeon Phi Coprocessors and the NVIDIA Tesla GPU) and reconfigurable fabrics (DFEs/FPGAs) is essential for the successful deployment of HSAs. Alongside with the support for multiple execution paradigms, to ease HSA development and exploitation OSes should also provide flexible monitoring, decision-making, and adapting capabilities, so as to use such resources at their best, satisfying users' expectations within reasonable costs. This optimal use and management of HSAs are not yet available, because of various open issues that SAVE will tackle, as the review of the state-of-the-art highlights.

A. Adaptive OS and run-time support for self-adaptability

The last recent years have seen a few initial studies in the design of adaptive OS components, such as SEEC [5] and Metronome [6]. SEEC provides an interesting and complete framework embracing machine learning and control theoretical solutions but is not integrated in the OS and does not support HSAs. BarbequeRTRM [7] is the key element of a highly modular and extensible run-time resource manager; its management abilities are of interest, but it mostly employs static partitioning of workloads and coarse-grain parallelism, assuming a platform where no other task is running. Although it provides an orchestrator, it does not self-adapt at runtime, one of the fundamental challenges SAVE tackles. Metronome extends GNU/Linux to manage at runtime the core and the CPU time allocation; it is based on heuristics and it uses just one metric to measure performance in terms of applications throughput. Metronome is in its infancy but due to its structure it could be an interesting starting point for SAVE, to be extended to the HSA scenario and the HPC domain. When considering HSAs, there are a few approaches in literature; in particular, PTask [8] is a kernel-level abstraction for managing GPUs. CHIMERA [9] presents an architectural solution, with external accelerators, presently characterized by too high a development time, with the *technology mapping* between the applications and the resources computed at design time. BORPH [10] is an extension of the Linux kernel for runtime support to FPGAs and currently runs on the BEE2 architecture [11]. ReconOS [12] is an operating system for heterogeneous multi-cores, that provides a common multi-threaded programming model for tasks on CPUs and FPGAs. This common programming and execution model simplifies the design of adaptive computing systems [13] that migrate tasks between hardware and software.

These are some of the effective solutions for promoting HSAs and constitute important milestones. However, they still lack a comprehensive, autonomic OSes and runtimes, such as the K42 research OS [4], which makes monitoring and adaptation first-class citizens. While leveraging on autonomic computing, BORPH could enable a new class of solutions capable of seamlessly employing processors and specialized islands of computation, to achieve full utilization and high performance with limited effort for developers and maximal benefits.

B. Automatic identification of application hotspots and offloading to heterogeneous computing resources

Numerous research projects have shown that computing with heterogeneous resources, e.g., FPGAs and GPUs, can improve the performance and energy efficiency up to several orders of magnitude. However, programming heterogeneous resources is still difficult, in particular for FPGAs, where the developer has to translate algorithms into customized digital circuits using unfamiliar languages, programming models and tools. Despite these obstacles, heterogeneous computing is increasingly used and numerous projects aim at further simplifying the programming and use of HSAs. These approaches can be classified into three main areas:

1) ***Domain-specific tools****:* As in many areas, domain-specific approaches have also been studied for the specification of heterogeneous accelerators, in particular FPGAs. Research has focused primarily on the areas of *finite state machines* and *digital signal processing* and has also been successfully commercialized in products like *Xilinx System Generator*, *Mathwork HDL Coder* or *Synopsys Synplify DSP*. Current research has successfully applied domain-specific approaches to more domains, such as *loop-pipelining* [14] or *stream processing* [15]. However, the restriction to a particular domain limits the applicability of such approaches.

2) ***High-level synthesis****:* Since the 1990s a general approach (denoted as high-level synthesis) for creating FPGA accelerators from high-level languages has been studied. Despite intensive research and progress, the resulting tools have not been widely adopted for manifold reasons, well discussed in [16]. In summary it could be argued that high-level synthesis is today an interesting technology for translating small, compute-intensive application portions to FPGA implementations, but not a solution for complete, complex applications.

3) ***Just-in-time binary synthesis****:* An innovative and alternative approach is the so-called binary synthesis. It does not rely on source code but uses binary programs as specification for hardware accelerators [17], [18]. An interesting use case is *just-in-time binary synthesis*, where application-specific accelerators are generated at runtime without any interaction with the programmer. Should this approach be efficiently implemented, it would make the benefits of heterogeneous computing available to a broad spectrum of users.

III. The proposed SAVE HSA

In this section we introduce the target architecture we envision in the SAVE project. The project proposes an in-

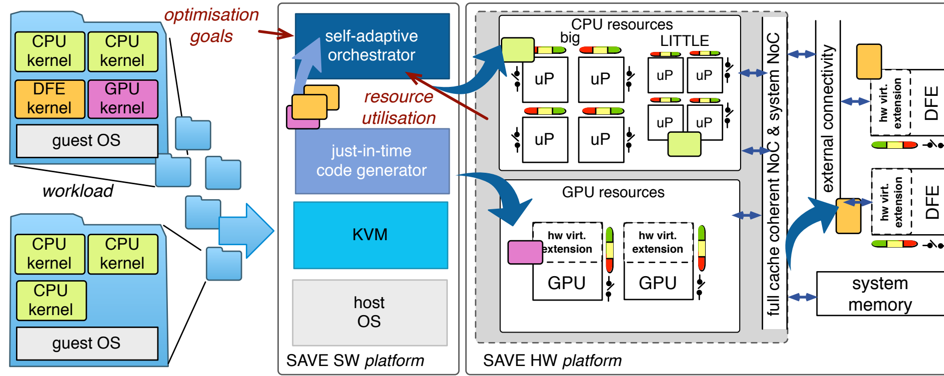

Fig. 1. The SAVEHSA architecture template.

novative architecture, based on a HSA, that combines self-adaptiveness and virtualization, to move one step further with respect to the current state-of-the-art multicore plus accelerators approaches. This self-adaptive, virtualization-aware HSA is called SAVEHSA. It is designed to efficiently support a wide assortment of data- and task-parallel programming models that are enabled by the presence of key hardware and system features suitable for the different system components, consisting of (host) CPUs, GPUs and DFEs, henceforth called SAVEHSA agents. The pool of SAVEHSA agents can be used by different Virtual Machines (VMs) for better security, performance and efficiency. In particular, the SAVEHSA platform aims at exploiting the characteristics of the various computing resources to dynamically schedule workloads on the most appropriate resource, to meet the current user-determined optimization goal, be it the minimization of energy consumption, performance optimization, or a trade-off between them. The actor responsible for performing the resource allocations what we refer to as the *Orchestrator*. This entity is responsible for monitoring the status of the SAVEHSA system and of the running applications and VMs; the Orchestrator will adopt self-adaptive techniques to ensure that users, applications, and the system itself meet the goals expressed by either the users or the system administrators. The Orchestrator can be seen as an advanced runtime Operating System (OS) support layer able to dynamically and seamlessly partition and distribute the various tasks to the available resources, based on changeable workloads and/or optimization goals (e.g., optimize performance for a given energy budget, or minimize energy consumption without decreasing the expected QoS).

We envision this kind of proposed architecture to fit Embedded System (ES) as well as High Performance Computing (HPC) scenarios, possibly with different characteristics and different specific resources.

In this initial part of the project, the main characteristics of these elements have been identified together with their requirements, to define the SAVE architecture template represented in Fig. 1. In a broader perspective, more suitable for the HPC scenario, we also envision an overall system consisting of several SAVEHSA nodes, being part of a much bigger system.

IV. THE ORCHESTRATOR

The Orchestrator is a software component in charge of managing resources at runtime and distributing among them the current workload (composed of several VMs, each one running a set of applications). It relies on the concept of feedback-loops, the basis of a self-aware and adaptive system, generally referred to as ODA (Observe, Decide and Act) loops. The Orchestrator observes the running applications through a monitoring API which continuously gathers information both on their performance and the system power consumption. This information is forwarded by the Orchestrator to *services* that are responsible for the decision phase. Each *service* controls a set of *actuators* to optimize a given goal through a user-defined *policy*. The output of the service is an action carried out by an *actuator* to obtain the desired effect.

The remainder of this section describes the structure of the Orchestrator. At first, a single layer orchestrator is presented as the basic block of the system, and will be used for a preliminary experimental evaluation. Then, we will provide the vision and an overview of the final structure of the Orchestrator, consisting of a dual-layer entity, whose overview is reported in Fig. 2. The Orchestrator, in charge of the decision phase, spans across the host and the virtualized environment. At the host level, it is in charge of partitioning the HW resources among the different virtual machines, while the Orchestrator component that resides in each VM is responsible of partitioning the resources assigned to the VM among the different applications running inside the VM.

A. Single Layer Orchestrator

The simplest implementation of the Orchestrator we have envisioned is composed of a single entity running at the OS level, directly interacting with the application with no layers in the middle (i.e., no virtualization). This solution is very appealing from the programming point of view; however it presents important issues in terms of scalability, isolation and security when applied to a production environment that has to support multiple users' workloads. Nonetheless we can start from the definition of a single layer implementation and make it suitable for our final purpose by defining and

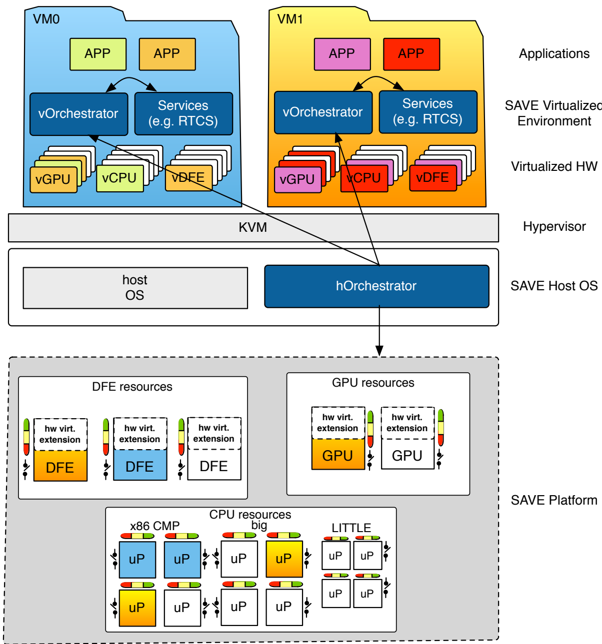

Fig. 2. Structure of the two-layer Orchestrator.

constraining proper communication interfaces to share data across the virtualization layer. For this reason, this work is focused on the definition of the structure of the single layer orchestrator and the result section will report an evaluation of its current implementation.

A visual representation of the current software architecture of the Orchestrator is given in Fig. 3; the most relevant parts are discussed here.

*1) **Orchestrator**:* The central entity of the system is the Orchestrator. We identified the following requirements: a) awareness of the *applications* that have to be controlled; b) knowledge of which *monitoring* information is available; c) knowledge of which *actuators* are available; d) knowledge of which *services* are currently active; e) responsibility to forward the relevant information provided by the *applications* to the *services* and *actuators*.

The role of the Orchestrator is to forward and share the information coming from the different actors cooperating in the system and to decide at runtime which among the available *services* must be enabled to fulfil the goals. This information is shared via shared-memory segments.

*2) **Application**:* The running applications must be instrumented with a simple-to-use API that will allow them to register with the Orchestrator. The applications employ monitoring services and application-level actuators, exposing shared-memory segments to achieve bidirectional communication. An application can send the information on which monitors and actuators it provides to the Orchestrator through standard inter-process communication facilities (in our implementation, a POSIX message queue).

*3) **Service**:* A service is responsible for implementing a given *policy* to fulfill a specific *goal* for which it has been designed. To pursue its goal, the service tells the Orchestrator, through inter-process facilities, the set of monitoring information and actuators it depends on. The Orchestrator is then in charge of informing the services on which applications expose the desired combination of monitors and actuators. For the application matching of the desired set of monitors and actuators of a specific service, the Orchestrator forwards to the service the related memory segment IDs, so they can map the shared memory in their address space.

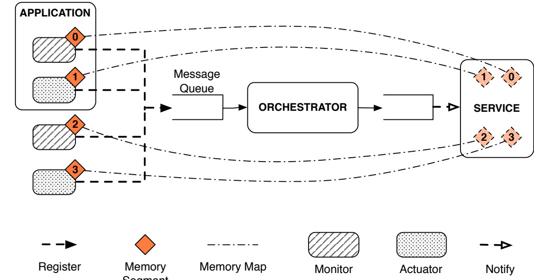

Fig. 3. Structure of the single layer Orchestrator

This three-fold structure decouples the role of the five main actors in the system (i.e., applications, monitors, actuators, services, orchestrator) and allows to dynamically add new kinds of monitors and services. All entities are identified by unique IDs, used to retrieve the proper information upon a request on the message queue. Each monitor, actuator and service is identified by its name, assumed to be unique. Once the Orchestrator forwards the memory segments information to services and after the binding between applications and services is done, the Orchestrator does not further intervene in the interaction between services and applications. A sequence diagram detailing such interactions is reported in Fig. 4. As the diagram shows, a service registers in the system and specifies which kind of monitors/actuators it wants to be notified for; the Orchestrator is responsible for forwarding to the service the requested information when those kinds of monitors or actuators are registered.

B. Two-Layer Orchestrator

The structure of the single layer orchestrator is straightforward, however such simplicity has many drawbacks. First of all a single layer of control does not scale with an increase in the number of applications typical of cloud and large HPC clusters installations, where distributed decisions and resource allocation mechanisms have proven to be more appropriate. Furthermore, since the target of the work is the management of multiprogrammed workloads, there is a strong need of security and isolation between the applications. In this context the state of the art solution is represented by the virtualization technology. The two-layer orchestrator will leverage these state-of-the-art approaches implementing a distributed decision mechanism across the virtualization layer. The decision mechanism will be then split into two layers, the first one acting at the host level (referred to as *Host Orchestrator*) and the second one embodied in the VM (the

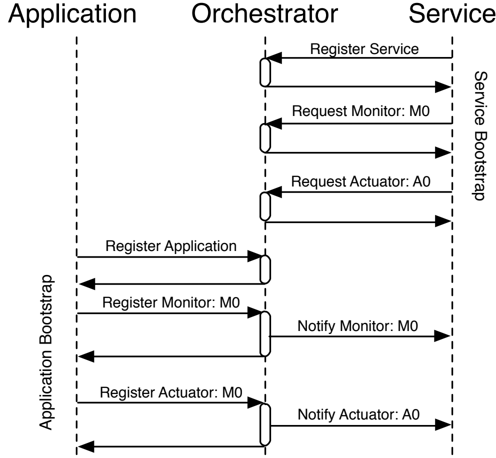

Fig. 4. Sequence Diagram specifying the interaction between actors.

Virtual Orchestrator). Information between these entities will be exchanged through the virtualized environment interfaces. This prototype is currently under development and the next paragraphs will report the requirements and features that the final implementation will provide.

*1) **Virtual Orchestrator**:* The *Virtual Orchestrator* is the component executing inside each of the active VMs running on the SAVEHSA. Its structure is similar to the single layer orchestrator presented above; the user applications directly interact with the orchestrator that assigns them. The difference with the single-layer solution relies in the fact that here the assigned resources are not the physical ones available on the host machine, but rather the virtual ones seen by the VM in any particular moment. The resources will be assigned directly to the applications on the basis of their direct performance measurements. Moreover, the *Virtual Orchestrator* is in charge of collecting information on the running applications, aggregating and sending it to the Host Orchestrator, able to take informed decisions for the whole VM.

*2) **Host Orchestrator**:* The *Host Orchestrator* is the actor running on the SAVEHSA. It will receive aggregated data per VM and will take decisions on the basis of metrics and policies exploiting these observations. Its role is to assign physical computational resources to each of the running VMs. The acquisition of monitoring data as long as the assignment of resources need to cross the virtualization layer and appropriate APIs and interfaces will be defined for this purpose.

V. ADAPTIVE SERVICES

This section presents two of the adaptive services currently supported by the Orchestrator. The first service performs heterogeneous resource allocation in a system equipped with a CPU and a GPU, while the second one performs JIT compilation and code optimization for GPUs and DFEs.

A. Heterogeneous Resource Allocation

This service allows the allocation of heterogeneous resources to applications on the basis of their current performance and the declared goals. It registers to the Orchestrator by sending the information on which monitors (performance) and actuators (heterogeneous allocation) it wants to be subscribed to. This service keeps track of the performance of the monitored application gathering data from the performance monitors and on their basis it controls the actuator that enforces where an application is executed. In this case, the service and the actuator are completely decoupled and asynchronous: the former, implemented at the Orchestrator level, is a policy that assigns resources to applications, the latter is implemented at the application level and allows an application compatible with SAVEHSA to run on multiple computational units.

*1) **Service**:* The decision policy is still under development; at present it is a fairly simple resource allocator. The policy is aware of the amount of available resources that can be assigned to the applications. For each application the policy is aware of a) the available implementations for the different computational resources, b) their profiling on them, c) the resources it needs, thus knowing which implementation is the best for a given application. The decision is periodically made and the frequency affects the accuracy of the resource partitioning and the computational overheads. It is also worth noting that, as some resources may use reconfigurable hardware (e.g., DFEs), the migration of a task due to hardware reconfiguration has a non-negligible impact (hundreds of milliseconds on a typical DFE): a too small decision step could therefore impair performance. The tuning of the frequency is currently being investigated.

Algorithm 1 Heterogeneous resource allocation

```
 1: function HETSCHED(A)
 2:     UL ← list of available resources
 3:     while not UL.empty() do
 4:         for all a ∈ A do
 5:             δ_a ← π_a / ḡ_a
 6:         end for
 7:         A_S ← sort A by δ_a
 8:         for all a ∈ A_S do
 9:             if δ_a < 1 then
10:                 u ← most efficient resource for a
11:                 assign(a, u)
12:                 UL.remove(u)
13:             end if
14:         end for
15:     end while
16: end function
```

The body of the control loop is reported in Algorithm 1. At each iteration, the algorithm gathers from each application $a \in A$ the last measured performance value π_a and the user-defined goal $\bar{g}_a$. Applications are then ranked on the basis of how much they are failing in achieving their goal. Resources are thus assigned as follows:

- the application with the worst behavior is assigned the resources it needs (among those available) to execute the best implementation;
- the assigned resources are marked as used;

- the process is repeated for the remaining applications. Applications not assigned to any computational unit are run on the CPU.

For performance reasons, the decisions are written into a memory segment shared between the scheduler and the actuators present in each of the controlled applications. A library, linked by each application, is responsible for executing the correct implementation and thus to offload the computation to the specific resource.

*2) **Actuation**:* To take advantage of the *Heterogeneous Resource Allocation*, an application must provide an actuator that is aware of the allocation decision provided by the *Service* and selects the correct implementation. The application will register with a proper SAVE API the various implementations of the functions target of the heterogeneous scheduling service; instead of directly calling one of these implementations, the application needs to invoke the heterogeneous allocator function. This function reads the resource assignment performed by the *service* and then invokes the proper kernel implementation. It is important to notice that, in our implementation, the actuation is asynchronous with respect to the resource allocation decided by the *service*; this allows not to introduce any overhead in the execution of the application other than the overhead associated with the function call and the monitoring infrastructure. Also, our implementation is lock-free, and makes use only of the compiler's atomic builtins to achieve consistency.

B. RTCS: Runtime and Just-in-time Compilation System

The precondition for exploiting the ability of the Orchestrator to dispatch workloads to different computing resources in the SAVEHSA is that executable code for these resources is available. There are however important practical use cases where no code for heterogeneous resources is available, for example, when running proprietary, binary applications or when the developer lacks the skills/time to port the application to GPUs or DFEs. To enable application offloading to heterogeneous computing resources, we are developing an adaptive service denoted as *Runtime and Just-in-time Compilation System (RTCS)* that allows for generating implementations for heterogeneous computing resources at runtime and for offloading the application to them if deemed appropriate.

RTCS builds on the LLVM compiler infrastructure and more specifically, on the LLVM execution engine. It works on applications expressed in the LLVM intermediate representation (LLVM IR) format, which can be considered as a binary application representation, comparable to Java bytecode. It is also possible to generated LLVM IR from actual x86 binaries [19]. Once an application is submitted to RTCS, the application is subject to a number of operations that are implemented by three sub-components as shown in Figure 5: *Monitoring Engine*, *JIT Code Generation Engine*, and *Application Transformation Engine*.

The *Monitoring Engine* is capable of analyzing, profiling and generating estimation metrics at runtime. It consists of three components:

1) Analysis: This component analyses the control- and data-flow structure of the application. It specifically searches for loops that contain instruction-level or data-level parallelism to detect computationally intensive parts (*hotspots*). Furthermore the types of instructions (floating point numbers, memory operation, etc.) and the frequency of their occurrence can have an impact on different resources, because they can be infeasible for an accelerator (system calls, library calls, recursive functions calls, etc.) or occupy a lot of area.

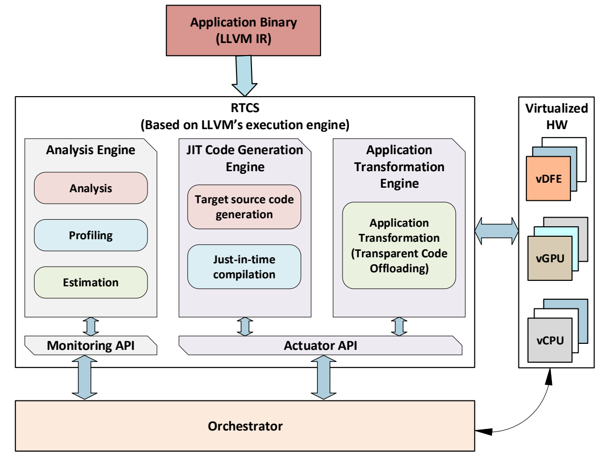

Fig. 5. RTCS: main components and interaction with the Orchestrator and with the virtualized CPU, GPU and DFE resources.

2) Profiling: This component profiles the application traces at runtime. It specifically looks for the execution frequency of loops and functions, different memory access patterns and the data transfer into and out of hotspots.

3) Estimation: It predicts the expected performance, power consumption, and incurred overhead when mapped to different computing resources. It tries to estimate the compilation/synthesis time to generate a new implementation and the time to transfer the required data to/from the resource.

The *JIT Code Generation Engine* is responsible for generating code for a target computing resource as well as linking the newly generated code with runtime libraries. It consists of the following two components:

1) Target source code generation: It is responsible for translating the hotspots from LLVM IR into efficient code for multi-core CPUs (OpenMP, OpenCL), GPUs (OpenCL or OpenACC) or DFEs (MaxJ, MaxGenFD). We don't assume that the results of this JIT compilation compilation process are available instantaneously. In particular for DFEs, this compilation process may take several hours. For long running applications this initial overhead can still be amortized over time. With caching, short running applications that are executed multiple times can also profit from JIT acceleration.

2) JIT compilation: This component wraps the code generated by the previous component into dynamic libraries so that it can be later dynamically linked with the main application. Additionally it is also capable of running architecture-specific compilation tools and links the required communication and runtime libraries.

Finally, the *Application Transformation Engine* is respon-

sible for removing program parts that will be replaced by the accelerated code, adding code for data transfers to/from resources and inserting calls to offloaded functions provided as loadable libraries.

RTCS resides in the user space and interacts with the virtualized HW devices (vCPU, vGPU or vDFE) in the same manner as it would with real hardware. RTCS provides two APIs via which the Orchestrator can communicate with it:

Monitoring API: provides an interface via which the Orchestrator can query the RTCS about the analysis, profiling and estimation data it gathered. Based on this fine-grained information and certain policies, the Orchestrator can then make informed decisions of what/where to migrate it. For each hotspot and for each accelerator, RTCS provides the expected performance/energy improvement, a rough estimation of the compilation/synthesis time to JIT generate code for a particular resource and finally the costs of the migration, including the initialization and the data transfer times to/from the accelerator.

Actuator API: Once the Orchestrator has decided to offload code, this API provides the means of communicating this decision to the RTCS. By using this interface, the Orchestrator can instruct the RTCS to first JIT compile the hotspot for a specific platform, and once the code is available, to offload it.

Initially, the RTCS starts executing the application on the CPU. It collects profiling information at runtime and analyses it to identify hotspots and data access patterns of the application. The Orchestrator queries the RTCS to know whether the application could benefit from offloading such hotspots to another computing resource. Based on the response from the RTCS and the active policies, the Orchestrator can decide to adapt and offloading a hotspot. If, for example, the Orchestrator decides to offload the hotspot to a DFE, it invokes the JIT Code Generation Engine over the Actuator API. The RTCS generates the code for the DFE and signals the Orchestrator when done. The Orchestrator can then instruct the RTCS to offload this computation to the DFE; the RTCS proceeds to initialize the DFE with the configuration, copies the required data and starts the computation of the hotspot on the DFE. Once the computation is complete, the results are copied back and execution continues on the CPU. These resources are not managed by RTCS but by the OS in cooperation with the Orchestrator. Building on this infrastructure allows RTCS to leverage the resource sharing capabilities while respecting the resource sharing assignment decided by the Orchestrator.

VI. Results Evaluation

This section reports the evaluation of the *Heterogeneous Resource Allocation* service (the simplified preliminary version of the Orchestrator) when a single and multiple applications want to take advantage of the SAVEHSA. Furthermore, we evaluate the performance overhead in the monitoring and decision mechanisms. The results have been collected by executing the Orchestrator on a workstation equipped with an Intel Core i7-3720QM processor, 16GB RAM with a NVIDIA GeForce GT 750M GPU.

*1) **Control of a Single Application**:* Figure 6 illustrates how the proposed system is able to control an application execution.

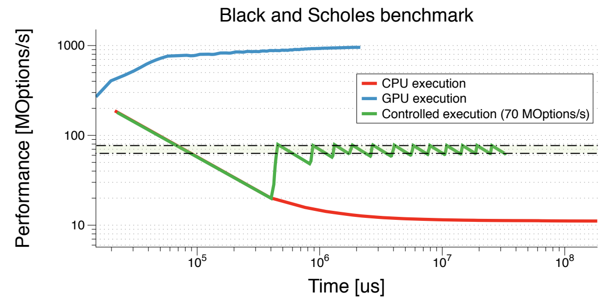

Fig. 6. Execution of a single application using either a single type of resources or exploiting heterogeneous ones managed by the proposed Orchestrator. The controlled execution stays in the performance boundaries decided by the user due to the action of the resource allocation service.

The application considered in the example is an instance of the Black and Scholes demo application shipped with the CUDA framework, showcasing a financial analysis algorithm on an European-like stock option market. This application performs a fixed number of simulations on set of stock options with different parameters and can execute both on CPU and GPU. The blue line shows the performance of the application when running only on the GPU, while the red lines shows the application running on the CPU only. If a user sets a goal of achieving 70 MOptions/s +/- 10% (green area in the figure), the Orchestrator tries to meet it by switching between CPU and GPU (green line). As it can be seen, after an initial set up phase the application lies in the green zone until it ends.

*2) **Control of Concurrent Applications**:* The example of Figure 7 reports the behavior of the system with a mixed workload. Two concurrent instances of the Black and Scholes application are running with different performance goals. In this case the Orchestrator has to allocate the available resources by trying to meet the performance of both applications, taking into account that the architecture has one GPU contended by the two applications. The applications have two different performance goals of 400 and 700 MOptions/s through the monitoring infrastructure (dotted lines in the figure), respectively. The Orchestator takes care of executing each one of the simulations on the proper resource in order to respect the expressed goals. The average performance is represented by the continuous stroke; meeting the goal means that at the end of the execution this line should match the goal set for each of the applications, as it happens in the experimental campaign.

*3) **Service Overheads**:* We measured the overhead of the monitoring infrastructure and the decision policy. The overhead of the Orchestrator component is negligible, being involved only in the initial phase (binding between application and services). Most of the overhead is due to the monitoring infrastructure and the control loop. From our experimental results, the monitoring overhead is mainly due to the call to the system function to read the timestamp. On our test platform, each call to `gettimeofday` accounts for an average of 32ns, while the average overall time to issue a heartbeat is 40ns. The other source of overhead is the function that dynamically selects the implementation to run and performs profiling. Also in this case, the overhead is low: it measures 72ns on average, entirely due to the two calls to `gettimeofday` needed to profile data. The

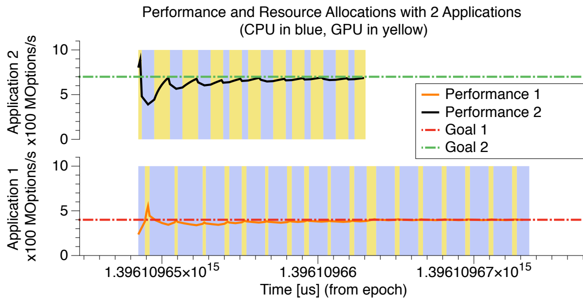

Fig. 7. Heterogeneous allocation policy managing two instances of the Black and Scholes benchmark with different goals. Colored bands reflect the resource allocation: blue bands mean that the application runs on the CPU, while yellow bands mean that the application runs on GPU.

overhead introduced by the control loop is negligible, as it runs at a low frequency; despite this, the algorithm presented here has a time complexity of $\mathcal{O}(n \log n)$ due to the need of sorting the n applications: a careful implementation must be provided for scenarios envisioning many concurrent applications. We believe that the performance overheads are reasonably small (even in this preliminary implementation) with respect to the duration of a computational kernel that is worth to offload to a heterogeneous accelerator.

VII. Conclusions and Future Work

In this work we presented the SAVE approach to resource management in heterogeneous architectures. Although these architectures are increasingly being adopted due to their good performance per energy trade off, we are far from being able to efficiently exploit them and nowadays the burden of resource management is left to the user. This paper describes how the SAVE project will tackle this problem by defining a proper architecture, the saveHSA, along with a set of mechanisms that helps the system administrator and the final user in managing the system and its own application. Two of the solutions developed have been presented: the *Heterogeneous Resources Allocation* and the *RTCS*. The contribution to the self-adaptiveness of these two services has been presented and preliminary results and overheads have been discussed, showing the benefits of dynamic resource management in both a single and a multiple application scenarios. The overheads introduced by this self-adaptiveness mechanism does not influence the correctness and responsiveness of the system.

Future work stemming from this research will focus on the improvement of resource allocation mechanism to perform a better resource allocation and to possibly avoid the need of a-priori profiling information that can be instead estimated online. Another important direction of research is the introduction of power measurements and self-adaptive power management with the realization of a *service* able of guaranteeing applications performance while at the same time lowering the overall power consumption.

Acknowledgments

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

References

[1] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[2] IBM, “White Paper. An architectural blueprint for autonomic computing,” 2005.

[3] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu, “The autonomic computing paradigm,” *Cluster Computing*, vol. 9, no. 1, pp. 5–17, 2006.

[4] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, “K42: building a complete operating system,” in *Proc. Conf. Computer Systems*, 2006, pp. 133–145.

[5] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, “Seec: A framework for self-aware computing,” Tech. Rep. MIT-CSAIL-TR-2011-046, November 2011.

[6] F. Sironi, D. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. Santambrogio, “Metronome: operating system level performance management via self-adaptive computing,” in *Proc. Design Automation Conf.*, 2012, pp. 856–865.

[7] P. Bellasi, G. Massari, and W. Fornaciari, “A RTRM proposal for multi/many-core platforms and reconfigurable applications,” in *Proc. Reconf. Communication-centric Systems-on-Chip*, 2012, pp. 1–8.

[8] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “PTask: operating system abstractions to manage GPUs as compute devices,” in *Proc. Symp. Operating Systems Principles*, 2011, pp. 233–248.

[9] R. Inta, D. J. Bowman, and S. M. Scott, “The ”Chimera”: An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform,” *Int. J. Reconfig. Comp.*, 2012.

[10] H. K.-H. So and R. Brodersen, “A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 14:1–14:28, 2008.

[11] C. Chang, J. Wawrzynek, and R. W. Brodersen, “BEE2: A High-End Reconfigurable Computing System,” *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[12] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, “ReconOS – an operating system approach for reconfigurable computing,” *IEEE Micro*, pp. 60–71, Jan./Feb. 2014.

[13] A. Agne, M. Happe, A. Lösch, C. Plessl, and M. Platzner, “Self-awareness as a model for designing and operating heterogeneous multicores,” *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, Jun. 2014.

[14] R. Menotti, J. M. P. Cardoso, M. M. Fernandes, and E. Marques, “Automatic generation of FPGA hardware accelerators using a domain specific language,” in *Proc. Int. Conf. Field Programmable Logic and Applications.*, 2009, pp. 457–461.

[15] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, “Beyond traditional microprocessors for geoscience high-performance computing applications,” *IEEE Micro*, vol. 31, no. 2, pp. 41–49, 2011.

[16] S. A. Edwards, “The challenges of synthesizing hardware from C-like languages,” *IEEE Design & Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.

[17] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, “An overview of a compiler for mapping software binaries to hardware,” *IEEE Trans. Very Large Scale Integration Systems*, vol. 15, no. 11, pp. 1177–1190, 2007.

[18] G. Stitt and F. Vahid, “Binary synthesis,” *ACM Trans. Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 1–30, 2007.

[19] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, “A compiler-level intermediate representation based binary analysis and rewriting system,” in *Proc. Conf. Computer Systems*, 2013, pp. 295–308.