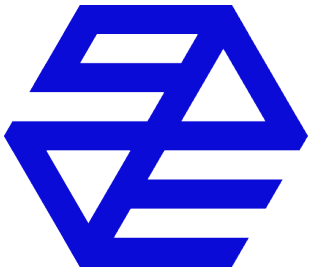
SAVE

Self-Adaptive Virtualisation-Aware High-Performance/Low-Energy Heterogeneous System Architectures

`http://www.fp7-save.eu/`

FP7-610996



Deliverable D6.4
Special session on research topics covered by SAVE

Delivery due date:	M28 (Jan 2016)
Date of delivery to the EC:	Jan 19, 2016
Lead beneficiary:	POLIMI
Dissemination level:	Public
Nature of deliverable:	Other

Editor(s)	Antonio Miele
Responsible Partner	POLIMI
Status-Version:	Final-2
Date:	Jan 19, 2016

WP/Task No.:	WP6/ Task T6.1
Editor(s):	Antonio Miele (POLIMI)
Contributor(s):	Antonio Miele (POLIMI)
Reviewer(s):	Cristiana Bolchini (POLIMI)
Pages:	16

Abstract	This deliverable issued at M29 gathers the information about the Special Session organized during the 10th HiPEAC Workshop on Reconfigurable Computing, held in Prague within the HiPEAC Conference, to disseminate the results and ongoing research activities of the SAVE project.
Keyword List:	Special Session, Dissemination

Approved and issued by the Project Coordinator: 	Jan 19, 2016

Project coordinator: Prof. Cristiana Bolchini - Politecnico di Milano
e-mail: cristiana.bolchini@polimi.it - **Phone:** +39-02-2399-3619 - **Fax:** +39-02-2399-3411

Document Revision History

Version	Date	Description	Author
v1	Jan. 18, 2016	TOC & Format	Antonio Miele
v2	Jan. 19, 2016	Content and Feedback	Antonio Miele

Table of Contents

Index of Figures

1 Executive summary

Research outcome visibility and information sharing are important aspects of a collaborative scientific project, where efforts are devoted to the design and implementation of innovative technical solutions, that can also be exploited by the scientific community as building blocks for future developments.

In this perspective, a Special Session has been organized within the activities of a well-known and attended workshop, the HiPEAC Workshop on Reconfigurable Computing, focused on research aspects very close to those tackled by the SAVE project. Indeed, it has been a significant opportunity to share the outcomes of the research up to that moment, and to interact with the scientific community working on similar topics.

The rest of the document is organised as follows. The next section collects the three papers that have been presented at the Special Session, being the result of joint work by SAVE participants.

2 Special Session program

The following Figure 1 reports the part of the Workshop program devoted to SAVE.

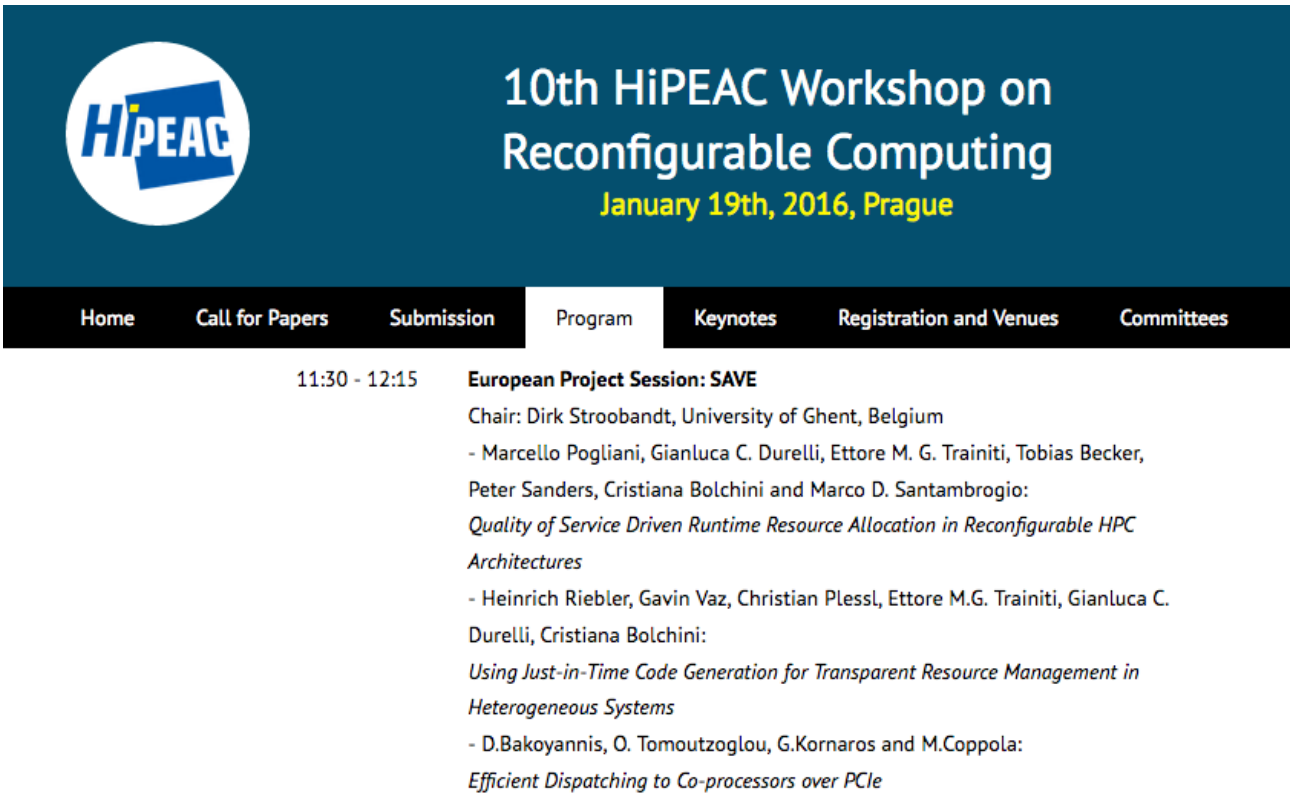


Figure 1: HiPEAC Workshop on Reconfigurable Computing – Special Session on SAVE.

3 Presented papers

Quality of Service Driven Runtime Resource Allocation in Reconfigurable HPC Architectures

Marcello Pogliani*, Gianluca C. Durelli*, Ettore M. G. Trainiti*, Tobias Becker†,
Peter Sanders†, Cristiana Bolchini* and Marco D. Santambrogio*
*Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano, Italy
†Maxeler Technologies Ltd., United Kingdom

***Abstract*—Heterogeneous System Architectures (HSA) are gaining importance in the High Performance Computing (HPC) domain due to increasing computational requirements coupled with energy consumption concerns, which conventional CPU architectures fail to effectively address. Systems based on Field Programmable Gate Array (FPGA) recently emerged as an effective alternative to Graphical Processing Units (GPUs) for demanding HPC applications, although they lack the abstractions available in conventional CPU-based systems. This work tackles the problem of runtime resource management of a system using FPGA-based co-processors to accelerate multi-programmed HPC workloads. We propose a novel resource manager able to dynamically vary the number of FPGAs allocated to each of the jobs running in a multi-accelerator system, with the goal of meeting a given Quality of Service metric for the running jobs measured in terms of deadline or throughput. We implement the proposed resource manager in a commercial HPC system, evaluating its behaviour with representative workloads.**

I. Introduction

Despite the unprecedented increase in complexity and performance requirements of computing systems, improvements in silicon technologies and fabrication processes cannot guarantee the yearly doubling of system performance of the past decades. With power densities of microprocessors approaching those of a nuclear reactor, power consumption emerged as a hard limit to their evolution [1]. As single-threaded performance levelled off, a paradigm shift was needed to continue increasing performance: this led to the rise of multi- and many-core architectures, and to the shift from instruction-level parallelism to thread-level parallelism. However, the "dark silicon" phenomenon [2] limits even multi-core designs: as the power budget limits the number of cores that can be integrated on a chip, transistors are under-utilized, and the architectures cannot scale beyond a few hundreds of cores.

In this context, Heterogeneous System Architectures (HSAs) are emerging as a common paradigm to provide high-performance solutions for consumer and High Performance Computing (HPC) systems. HSAs exploit different architectures for different types of tasks: typically, a standard multi-core CPU, which runs the control-intensive part of the applications, is coupled with highly efficient and specialized devices (accelerators) that run the computational intensive and performance-critical parts. A popular accelerator is the Graphical Processing Unit (GPU), an architecture originally conceived for graphic rendering, which can be used for general purpose computation by means of specialized programming languages such as OpenCL [3] and NVIDIA CUDA [4]. Besides GPUs, reconfigurable hardware – usually implemented with Field Programmable Gate Arrays (FPGA) – is recently emerging as a key player in the HPC context. Although GPUs are good at performing massively parallel floating-point computation, reconfigurable fabric allows to completely change the data-path if the computation requires so. Reconfigurable hardware is able to couple high performance with low power consumption, at the expense of ease of programming. HPC solutions featuring FPGA accelerators are available on the market: customers can build their own computational cluster or buy compute time from cluster owners following the utility computing paradigm. To achieve high efficiency, those specialized systems must have a high utilization. Research is moving in this direction [5], but current solutions do not consider the Quality of Service (QoS) delivered to customers, which is of utmost importance in a utility computing paradigm. This work targets the problem of achieving a given QoS in a HPC system using FPGA-based accelerators by means of self-adaptive techniques and application-specific policies.

This paper is organized as follows: Section II describes the problem; Section III introduces the proposed techniques; Section IV evaluates this work on a Maxeler HPC system using representative workloads. Finally, Section V presents an overview of related work, and Section VI closes the paper.

II. Problem Definition

Aim of this work is to design a runtime resource manager for HPC systems that accelerate computational intensive kernels of performance-critical applications with reconfigurable hardware. We target a system with multiple accelerators shared among multiple applications. To ground the discussion, throughout this paper we refer to a Maxeler Technologies MPC-X[1] system, which allows sharing FPGA-based accelerators – Maxeler's own Data Flow Engines (DFE) – to multiple CPU nodes over an Infiniband network. The resource manager supports highly parallelizable applications structured as a series of mutually independent operations; they have a computational intensive and highly data parallel loop, with as few dependencies as possible, running for many iterations. This structure is extremely simple, but representative of scientific computation kernels that benefit from hardware implementations: it is found in HPC applications such as option pricing, image and video processing, and computation of the correlation matrix in the brain network analysis problem.

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

[1] https://www.maxeler.com/products/mpc-xseries/

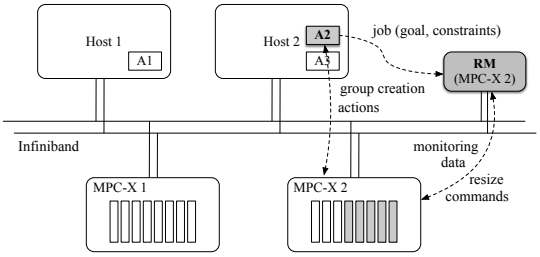


Fig. 1. The proposed resource manager in a typical MPC-X deployment.

We introduce the Maxeler-related concepts of *action* and *DFE group*. An *action* is a single and atomic invocation of DFE functionalities (computation), together with data and parameters sent to the DFE. A *DFE group* is a virtual DFE, backed up by one or more physical device; applications submit actions to a group without knowing what is the physical DFE running the computation.

The problem we tackle is to manage allocation for jobs; a job J is a set of stateless (i.e., no status is kept on the DFE memory) and independent actions $J = \{a_1, \ldots, a_n\}$ that require a DFE configured with the same bitstream. The output of the resource manager is a time-varying well-formed allocation. Given a set of jobs $\mathcal{J}$ and a number N of resources, a well-formed allocation θ is a function $\theta : \mathcal{J} \to \mathbb{N}$ where

$$\sum_{j \in \mathcal{J}} \theta(j) \leq N$$

We allocate resources to meet an explicit goal; as different applications running on the same HPC cluster may have different performance requirements, goals are job-specific and each job has the following metadata:

- A *goal*, which takes the form of a desired deadline or a desired throughput;
- If the goal is a deadline, the job *size*;
- An optional *constraint* on the minimum and maximum number of DFEs assigned to the job while it is running; the resource manager enforces the constraints.

In this framework, the resource manager tackles the following problem: *given a set of jobs $\mathcal{J}$, and N resources (DFEs), find a time-varying well-formed allocation θ that satisfies the constraints, so that the goals are met if possible*. This allows to leverage *DFE groups*: in fact, our resource manager creates a group for each job, and varies the allocation of DFEs to a *job* by modifying at runtime the *group* size.

III. Design and Implementation

The resource manager is a user-space application, deployed in an environment composed of one or more high-performance switched Infiniband interconnects, multiple CPU nodes, and one or more MPC-X devices. As shown in Fig. 1, it runs on a generic CPU node, is tied to a specific MPC-X, and exchanges data with CPU nodes and the MPC-X over a network. A purpose-designed API allows applications to register to the resource manager, specify goals, and submit jobs; the resource manager connects with the Maxeler management software to retrieve monitoring data and issue group resize commands. Having the resource manager mediate every DFE-related operation would result in massive performance overhead; thus, the resource manager is a passive component and the managed applications directly perform operations such as loading bitstreams and submit actions. We developed two DFE resource management policies, each one targeted for a different workload scenario.

A. Earliest Deadline First

The *earliest deadline first* (EDF) policy, applicable to jobs specifying deadlines as a goal, assigns all the allocatable resources to the job with the earliest deadline among the ones active in the system. The allocatable resources are the DFEs in excess to the ones needed to cope with any constraints specified in the metadata. The events that can trigger a change of allocation are the retirement or the submission of a job.

This policy provides good results in case of batch jobs, when all the results of the computation are needed only after the job deadline, and provided that the system is able to meet all the deadlines. The main drawback is a lack of predictability of the response time: as the exact time a job is scheduled depends on the deadlines of other jobs, it is difficult to control the response time when submitting a job. If the duration of a single action is small with respect to the whole job, the resource manager effectively preempts a job when it changes the allocation. In case of a workload where all the jobs are instances of the same application and use the same bitstream, the problem is similar to classical real-time scheduling with preemption: EDF minimizes the maximum lateness of the jobs, and is optimal in the sense that if there exists a schedule for a set of jobs that fulfils all the deadlines, then also the EDF schedule fulfils all the deadlines.

B. Throughput-Based Policy

In streaming applications, such as real-time video encoding, intermediate results are constantly needed: the important metric is not the job response time, but the throughput, i.e., the rate at which new results become available. As this scenario does not cope well with the unpredictability of the EDF policy, we developed a different heuristic that exploits a feedback loop to keep the job *throughput* in a certain range.

Unlike the EDF policy, where the scheduling events are confined to submission and retirement of a job, the resource manager for the throughput-based policy runs periodically, querying the MPC-X device for its status, running the decision algorithm, and issuing group resize commands according to the new allocation. This periodic nature is the main drawback: a short control period, while allowing for a fine grained control, may cause too many reconfiguration events and hurt the system performance; for this reason, in the remainder of this discussion, we consider a workload composed of multiple instances of the *same* application, such as an hardware accelerated video-encoding cluster where different users have different QoS requirements.

This policy uses a target throughput as a goal g; when a job j specifies its goal as a deadline, it is translated into a time-varying throughput $g_i(k)$:

$$g_j(k) = \frac{\text{total number of actions}_j - \text{actions}_j(k)}{\text{deadline}_j - \text{time}(k)}$$

Here, actions(k) is the total number of actions completed at the control step k, and time(k) is the current time at k; unlike EDF, the job metadata must specify the total number of actions.

The high level structure of the throughput-based resource management algorithm follows an approach already profitably applied to other resource management problems [6]: we separate the algorithm in two layers having a clear interface between them, the *Job Controllers* and the *Resource Broker*.

***Job Controllers*:** The job controllers work independently and in the context of a single job j, computing its "ideal" resource request. They take as an input the job metadata, the current allocation and the completed actions, producing two synthetic values. The first value is a performance metric p_j, which indicates how the performance of j are far from its goal: if j is performing better than its goal, $p_j > 1$, otherwise $p_j < 1$; the resource broker uses p_j to decide which jobs to penalize if some of the requests cannot be satisfied. The second output value is the resource request r_j, expressed as the number of desired DFEs. If the size of actions belonging to the same job is approximately constant and the jobs scale linearly with the number of assigned DFEs, we can compute the resources needed to enforce the required throughput as follows. Let $g_j(k)$ be the desired throughput of j at the control step k, and $t_j(k)$ the throughput of j computed as in (1), where actions(k) is the total number of completed actions and time(k) is the current time at the step k. The duration of an action can be estimated, as shown in (2), by the ratio between the allocation and the throughput over the last time window, smoothing the resulting value through an exponential moving average filter with a constant smoothing factor $\alpha \in [0, 1]$; by varying the value of α, the algorithm can react quicker or slower to a change of the action duration at the expense of being able to filter outliers out (e.g., particularly slow actions, measurement errors). We then compute the resource request $r_j(k)$, and $p_j(k)$ value as the ratio between the current throughput and the goal, as reported in (3) and (4) respectively.

$$t_j(k) = \frac{\text{actions}(k) - \text{actions}(k-1)}{\text{time}(k) - \text{time}(k-1)} \tag{1}$$

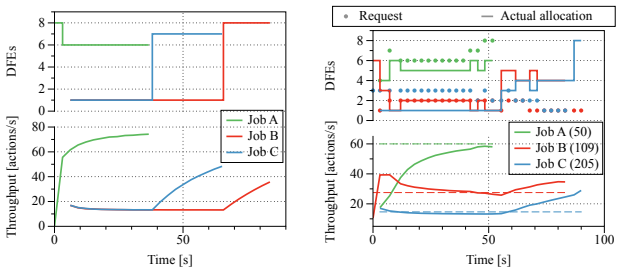
$$\hat{a}_j(k) = \alpha \cdot \frac{\text{allocation}_j(k-1)}{t_j(k)} + (1-\alpha) \cdot \hat{a}_j(k-1) \tag{2}$$

$$r_j(k) = \lceil \hat{a}_j(k) \cdot g_j(k) \rceil \tag{3}$$

$$p_j(k) = \frac{t_j(k)}{g_j(k)} \tag{4}$$

Resource Broker: The resource broker computes the allocation according to the values computed by the job controllers and the system constraints. As input parameters, the resource broker (RB) takes a list of currently running jobs, and the number of available DFEs (*max*). The value of *max* is the number of available physical DFEs minus any DFEs reserved for jobs initialized but not yet started, and any blacklisted DFE[2]. If the resource request fits the MPC-X, the RB divides the excess DFEs among the jobs that declared a deadline as a goal: throughput-bound applications do not usually gain in having *more* resources than what they need, due to other synchronization constraints in other parts of the computation pipeline the DFE application is part of. If the resource request does not fit the available DFEs, the RB penalizes some jobs assigning less resources than required according to a heuristic.

[2]A DFE is blacklisted when it fails repeatedly a reconfiguration process.



(a) (b)

Fig. 2. Execution trace of a workload with three instances of the same applications, managed by the EDF (a) and throughput-based (b) algorithms.

In order to make the job submission process simple for the users, we do not require previous profiling data: when a job enters the system, it is not possible to predict its resource request. However, the throughput-based algorithm targets long-running jobs: the initial allocation has a small impact on the job performance. If a job j enters the system and there are enough unallocated DFEs to cope with its constraints, we assign to j the unallocated resources; otherwise, we deallocate DFEs from active jobs ensuring the resource request of each active job is less or equal than the resources it is allocated. If it is not possible to do so, we reject j.

IV. Results

We implemented our resource manager on a system with: a Maxeler Technologies MPC-X2000 device with six *Maia* DFEs (Altera Stratix V D8 FPGA, 48 GB RAM for each DFE), and a CPU node with two Intel®Xeon®E5-2670 (8 cores, 2.6 GHz, 20 MB LLC, Intel HyperThreading) and 64 GB of DDR3 RAM, running CentOS 6.4 and MaxelerOS 2014.2. The MPC-X is connected to the CPU node via two switched Infiniband networks; the HCAs are Mellanox ConnectX VPI PCI-Express 2.0, capable of Quad Data Rate (QDR) transfers.

As a benchmark application, we chose a simplified financial option pricer based on the Black and Scholes formula, representative of the kinds of processing found in financial risk analytics products. The Black and Scholes formula is used to compute Value at Risk (VaR), a measure of the risk of loss on a portfolio of financial assets. VaR computations need to analyse many scenarios, so they are easy to parallelize; the processing kernels are suitable for implementation on coprocessors, with low memory cost and large computation/data transfer ratio.

A. Deadline Goals

We ran our system with multiple instances of our benchmark application without considering the reconfiguration overhead. Fig. 2a and 2b depict the dynamic behaviour of both the developed policies; in the example we consider, deadlines are set to 50, 109, and 205 seconds respectively with a constraint of at least 1 DFE assigned to each job. The top half of the plots shows the allocation; in case of the throughput-based policy, the dots are the request performed by the job controller before the resource brokerage phase, while the lines are the allocation after the resource broker intervention. Although the

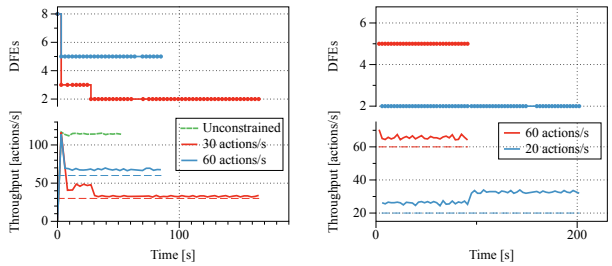


Fig. 3. Execution trace of the benchmark application when controlling the throughput: (a) different executions; (b) two co-located instances.

throughput-based policy internally enforces a throughput over a time window, we chose to represent the global throughput (average throughput from the beginning of the job), as it is representative of goals expressed in the form of a deadline, and allows to easily understand whether the deadline was met. The bottom half of the plot shows the global throughput; the dashed lines in Fig. 2b are the deadline goals converted to a global throughput (if the deadline is 200s and the job size is 3000, we draw a dashed line at $^{3000}/_{200} = 60$ actions/s). It is also possible to notice the difficulty to predict when the job will actually start its execution with the EDF algorithm.

B. Throughput Goals

Fig. 3a shows the throughput-based policy behaviour when the goal is specified as a throughput. The plot shows the throughput over a time window equal to the control period, running a single instance of the benchmark application. The dashed line refers to an unconstrained execution; for the blue and red lines, we set the goals to 30 and 60 actions/s respectively. The MPC-X allows to assign to a job only an integer number of DFEs: for a goal of 60 actions/s, the throughput is noticeably higher than requested, as using less DFEs would make the system to underperform. Fig. 3b shows the behaviour of two co-located applications with different goals, 60 and 20 actions/s. The throughput of the second job increases after the first job finishes even with the same allocation (2 DFEs): when both applications run together, the interconnection bandwidth saturates; as soon as the first job ends, the bandwidth is not saturated. The allocation does not change and the throughput is higher than the goal due to the choice of the α coefficient in the job controller's filter: with a lower value of α, the system adapts to the new condition dynamically changing the allocation of the second job to 1 DFE.

V. Related Works

Commodity operating systems do not cope well with HSAs: they manage devices such as GPUs or FPGA-based systems as I/O devices, without higher-level abstractions. Research has recently moved towards better system-level abstractions for HSA; two fundamental research challenges are resource virtualization and elasticity.

An important topic in *resource virtualization* is to allow virtualized operating systems access to hardware resources without impairing performance, and to solve contention on resources used by multiple operating systems. This topic, originally explored for GPU solutions [7], starts to be explored for FPGA-based systems. For instance, OpenStack was extended [8] to provision FPGAs like standard virtual machines; in this view, cloud platforms can offer to users virtualized FPGA resources loaded with precompiled or custom functionalities.

In the vision of a heterogeneous cloud, it becomes desirable to enable *elasticity*. Hydrogen [9] attempts to provide elasticity in multi-tenants environments for the same class of systems targeted by this work. It associates each job with a Job-Level Objective metric (e.g., execution time); each Hydrogen instance manages a pool of resources, and a resource manager allocates jobs to the resources according to a scheduling policy. A key difference with our work is that we consider long-running jobs and perform online profiling, without needing the offline profiling used in Hydrogen. We also consider user-provided hardware implementations (bitstreams), while Hydrogen assumes a fixed repository (library) of high-performance commonly used kernels. Finally, the Maxeler platform used in this work is already harnessing virtualization and elasticity [5]. However its runtime allocation mechanism, aims to maximize DFE utilization without taking into account any QoS metrics: this results in a poor job for applications with different requirements sharing the same MPC-X cluster.

VI. Conclusion

In this work we proposed runtime resource management techniques for HPC reconfigurable systems, implementing the proposed system on a commercial platform with promising results. Unlike the state of the art, our system aims to control workloads with QoS-bound jobs, using metrics meaningful to the application. We considered two types of job-specific goals – deadline and throughput – showing that it is possible to control applications having a structure that is widespread among HPC ones. Resource management techniques are good enablers for resource sharing, even on devices conceived as single-application and single-user appliances: resource sharing can extend the suitability of HSAs to a wider user base.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[2] N. Hardavellas *et al.*, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, July 2011.

[3] *OpenCL – The open standard for parallel programming of heterogeneous systems*, Khronos OpenCL Working Group and others Std., 2011. [Online]. Available: https://www.khronos.org/opencl/

[4] J. Nickolls *et al.*, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, Mar. 2008.

[5] J. G. F. Coutinho *et al.*, "Harness project: Managing heterogeneous computing resources for a cloud platform," in *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014.

[6] D. B. Bartolini *et al.*, "Automated Fine-Grained CPU Provisioning for Virtual Machines," *ACM Trans. on Architecture and Code Optimization*, vol. 11, no. 3, pp. 27:1–27:25, Jul. 2014.

[7] M. Dowty and J. Sugerman, "Gpu virtualization on vmware's hosted i/o architecture," *SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, Jul. 2009.

[8] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. of the 22nd IEEE Annual Int. Symp. on Field-Programmable Custom Computing Machines(FCCM '14)*, May 2014, pp. 109–116.

[9] P. Grigoras *et al.*, "Elastic management of reconfigurable accelerators," in *Proc. of the 12th IEEE Int. Symp. on Parallel and Distributed Processing with Applications (ISPA '14))*, Aug 2014, pp. 174–181.

Using Just-in-Time Code Generation for Transparent Resource Management in Heterogeneous Systems

Heinrich Riebler[1], Gavin Vaz[1], Christian Plessl[1], Ettore M.G. Trainiti[2], Gianluca C. Durelli[2], Cristiana Bolchini[2]

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

[2]Politecnico di Milano, Italy {*firstname.lastname*}*@polimi.it*

***Abstract*—Hardware accelerators are becoming popular in academia and industry. To move one step further from the state-of-the-art multicore plus accelerator approaches, we present in this paper our innovative SAVEHSA architecture. It comprises of a heterogeneous hardware platform with three different high-end accelerators attached over PCIe (GPGPU, FPGA and Intel MIC). Such systems can process parallel workloads very efficiently whilst being more energy efficient than regular CPU systems. To leverage the heterogeneity, the workload has to be distributed among all the computing units in a way that each unit is well-suited for the assigned task and executable code must be available. To tackle this problem we present two software components; the first can perform resource allocation at runtime while respecting system and application goals (in terms of throughput, energy, latency, etc.) and the second is able to analyze an application and generate executable code for an accelerator at runtime. We demonstrate the first proof-of-concept implementation of our framework on the heterogeneous platform, discuss different runtime policies and measure the introduced overheads.**

***Keywords*—Application Runtime, Code generation, LLVM, JIT**

I. Introduction and Related Work

Nowadays hardware accelerators like General Purpose GPUs (GPGPUs), Field Programmable Gate Arrays (FPGAs) or Many Integrated Cores (MICs) are becoming increasingly popular. Regular computer systems combined with accelerators can process highly-parallel, fast-changing workloads and reach higher levels of performance whilst being more energy efficient. To leverage the opportunities of such Heterogeneous System Architectures (HSAs), the computational intensive portions of an application (the *hotspots*) have to be identified and distributed (or *offloaded*) among all the computing units. At the same time, each portion should be executed on the unit that runs it in the most efficient way. However, adding heterogeneity to regular computer systems increases the need of smart resource management and leads to more complex programming models with new tools and languages.

The need towards new programming models is reflected in several proposals, most prominent being OpenCL and CUDA. Besides this, there exist domain specific languages and solutions such as MaxJ targeting data flow engines (DFE). Regardless of the particular framework one decides to employ, the realization of heterogeneous implementations still have many difficulties that strongly limit the wide spread adoption. First, the designer has to identify which parts of the application might benefit from the execution on an accelerator. This analysis is nowadays done by using offline profiling tools that the designer uses to determine which application portion is a bottleneck (or an hotspot) and then has to perform a manual analysis to identify whether the hotspot benefits from an heterogeneous implementation. After, the hotspot has to be manually adapted for the particular target/programming model. Each different programming model (CUDA, OpenCL, MaxJ) has its own APIs and optimizations, which strongly limit the portability across different target architectures.

Following the flow as outlined above might be optimal if the application is the only one running in the system. However, this is not always the case, thus the research is focusing on runtime resource management techniques able to control and better exploit the usage of underlying resources when multiple applications are colocated on the same machine. Many of these efforts focus on the multi-/many-cores scenario, managing the resources, i.e. cores, assigned to an application, and their frequencies in order to respect a given goal. Metronome [1] for instance targets symmetric multicores processors and proposes to instrument a running application with an library, inspired from application Heartbeats [2], to collect high-level performance information and tune resource management to respect a desired QoS level. BarbequeRTRM [3] targets a manycore scenario and is focused on the optimization of performance and power. However, it requires a deep restructure of the applications in order to be able to optimize their behavior. SEEC [4] aims at optimizing a performance/watt ratio exploring different control techniques ranging from heuristics to control theoretical and online learning approaches. Finally, other solutions target heterogeneous systems and are aimed at achieving a better runtime decision [5] or integration of accelerators at the operating system level; important work of this type are PTask [6] targeting GPUs and ReconOS [7] aimed at an easier integration of FPGA devices.

In this work we make a step forward in the exploitation of heterogeneous systems by proposing a solution to automatically identify possible hotspots in the application code at runtime and generating the corresponding binary code to target the specific accelerator through a just-in-time (JIT) compiler. This solution will then work in collaboration with a runtime resource management mechanism that will control the dispatching of applications onto heterogeneous resources.

II. Heterogeneous Architecture saveHSA

In this section we briefly introduce the target architecture we envision, the SAVEHSA, an innovative architecture, based on a Heterogeneous System Architecture (HSA) consisting of (host) CPUs, GPUs and DFEs, that combines self-adaptiveness and virtualization, to move one step further with respect to the current state-of-the-art multicore plus accelerators approaches.

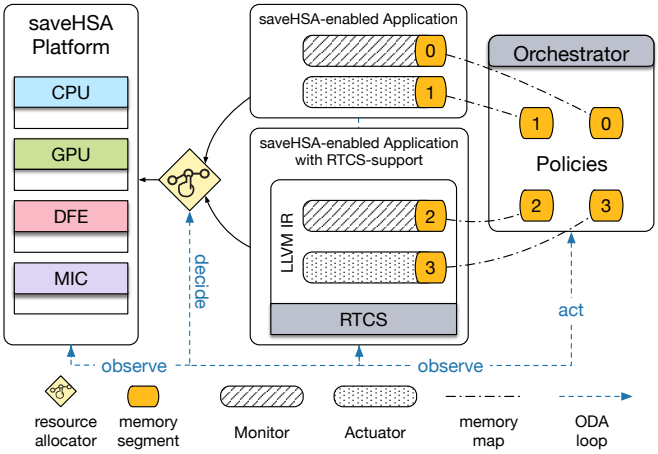


Fig. 1. Interaction of the main components within the SAVEHSA.

An overview of the main SAVEHSA components is depicted in Figure 1. The component responsible for performing resource allocations at runtime and monitoring the system is called `Orchestrator`. It applies self-adaptive techniques and distributes the workload to ensure that users, applications and system goals (e.g., optimize for performance, energy or both) are met. In order to achieve this, the Orchestrator relies on the concept of feedback-loops — generally referred to as ODA (Observe, Decide and Act) loops [1]. It first observes the running applications through a Monitoring API which continuously gathers information on the progress of each iteration. It then uses this information to incrementally tune the application behavior (i.e. utilization, power consumption, throughput, etc.) via a set of actuators. However, in order for an application to take advantage of this, it must be `SAVEHSA-enabled`. Being SAVEHSA-enabled means that the application is able to express goals (in terms of throughput, latency, etc.). It should also be able to and provide information on its execution progress in the form of heartbeats [2]. This enables the Orchestrator to observe the performance of the application and decide (based on internal policies) to improve the execution performance by switching between different resources. The precondition for exploiting the ability of the Orchestrator to dispatch workloads to different computing resources is that executable code for these resources is available. However, this might not always the case, because the development is time consuming and requires different programming models. To overcome this limitation we have developed the Runtime and Just-in-time Compilation System (RTCS) that is capable of performing just-in-time code generation for a given accelerator. The prerequisite of the `RTCS` is that at least the CPU implementation of the code to be accelerated is available. The RTCS performs some analysis and estimation passes to detect the SAVEHSA-enabled components (i.e. the main hotspot loop, available accelerator implementations, etc.). It then exposes its code generation capabilities to the Orchestrator as an actuator. The Orchestrator can query the RTCS for implementations that it can provide, and if required instruct it to generate code for a missing implementation (GPU or DFE, for example). The RTCS generates the code and registers the new implementation with the Orchestrator which can then decide to adapt the application and offload computation to it.

III. SELF-ADAPTIVE RUNTIME SYSTEM

This section presents more details on the Runtime and Just-in-time Compilation System (RTCS) and the Orchestrator. The RTCS is responsible for compilation, code optimization, execution of the application and generation of accelerator specific code when the Orchestrator instructs it to (governed by its policies). The RTCS builds on top of the LLVM compiler infrastructure [8] and works on SAVEHSA-enabled applications expressed in its intermediate representation format (LLVM IR), which can be considered as a binary representation (like Java bytecode). However, before the LLVM IR can be executed, the application must undergo some preparations.

Analysis and Optimization: There are many ways to generate LLVM IR from high level languages or even x86 binaries [9]. Hence, before analyzing the application, we need to canonicalize the IR. We use built-in LLVM passes to perform lightweight but aggressive, high-quality code optimizations. These include promoting memory to registers, loop simplification, normalization of induction variables, loop invariant code motion, constant propagation, combining redundant instructions and dead code elimination. After canonicalization we use custom analysis and estimation techniques to detect the structure of a SAVEHSA-enabled application, that is: the Orchestrator handle (Monitor and Actuator, see Figure 1), the hotspot (Heartbeat Loop), the available implementations, etc.

The hotspot of a SAVEHSA-enabled application is the main loop where the Heartbeat framework is used to measure performance and where for each iteration the Orchestrator takes a decision where the hotspot is executed (see Figure 2). This loop is detected in the analysis phase and used later on to register new accelerator code. This is achieved by modifying the application control flow with a callback function to intercept and change the application behavior at runtime.

Application Preparation: Initially the callback function is empty. However, when a new accelerator implementation needs to be registered (at runtime), it is updated with code that registers the new implementation with the Orchestrator. The callback function is then recompiled and relinked and in turn registers the new implementation in the next iteration of the hotspot. Additionally, the accelerated code might use libraries and API calls (e.g. OpenCL) that might not have been defined in the original application. For the runtime integration of new accelerator implementations to work seamlessly, we prepare the application by including and linking the appropriate libraries. After analyzing and preparing the application, the RTCS starts executing the application. At some point during the execution the Orchestrator might instruct the RTCS to generate missing accelerator code to enhance its decision possibilities by offloading the hotspot to another resource.

Just-in-Time Code Generation: The structure of a hotspot can be broken down into the following stages: data allocation and initialization (done once); data migration to and from the accelerator and the computation kernel (for each iteration of the hot loop). The registered CPU implementation of the hotspot is our baseline and is used for generating code for other accelerators. Once the accelerated code (function) has been generated, we need to call it with the appropriate parameters and data handles, and so we analyze the CPU function and store the arguments used in the CPU call.

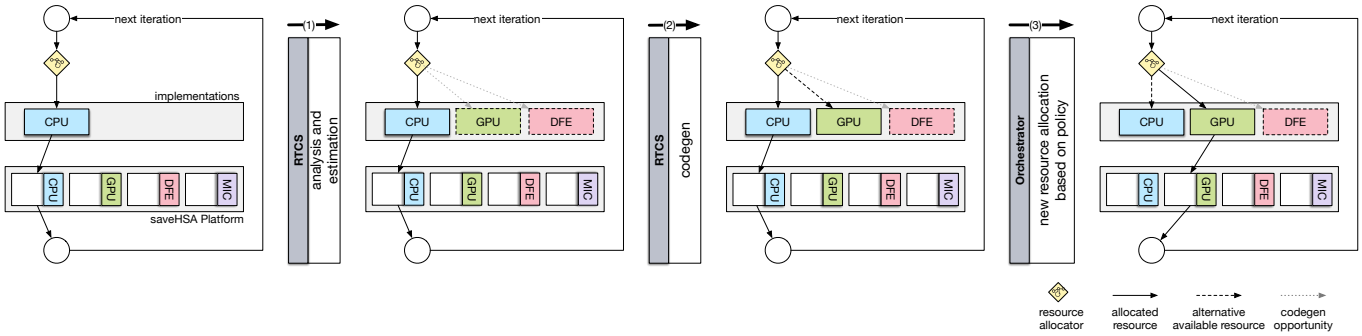


Fig. 2. Main loop of an SAVEHSA-enabled application with RTCS.

Since the Orchestrator can switch between different implementations for each iteration of the hotspot and because of the hotspot structure of a SAVEHSA-enabled application, we always need to transfer data to and then back from the accelerator in order to maintain memory consistency. We achieve this by creating a wrapper function that first transfers data to the accelerator, executes the kernel(s) on the accelerator and then transfers the data back. This wrapper function is registered with the Orchestrator which can then adapt its policies and instruct the application to execute the code on the new resource. Currently, the code generation is in a preliminary state, i.e. use pre-compiled GPU kernels and fetch the right ones at runtime and dynamically link them into the application. By doing this, we have a working just-in-time demonstrator to proof the runtime system, actuators and adapting policies, while having the flexibility of plugging in the missing components as and when they become available.

Actuators and Policies*:* The RTCS functionality constitutes a new possible knob that the Orchestrator might exploit at runtime to further improve runtime resource allocation. Two policies have been realized to exploit the RTCS capabilities. A first policy implements a simple solution to demonstrate the integration. In fact, if we consider an application without the RTCS running on top of the Orchestrator, the application must declare and register upfront all the possible hotspot implementations for the different processing units. We then opted for a straightforward integration of the RTCS in such a flow, simply by invoking the code generation capabilities of the RTCS when an application registers itself with the Orchestrator. In this situation the Orchestrator is willing to pay all the overhead for the generation of all the possible implementations before starting to dispatch the actual execution of the application. The second policy delays the code generation with a lazy approach and invokes the generation of the implementation for a specific architecture only when the Orchestrator might need such implementation. In this situation the overhead will be spread across the whole execution of the application.

These two policies target two different use cases. The first one targets a static system with a well-defined resources pool to use; in this case the Orchestrator might want to know all the possible implementations before the actual execution. While the second solution focuses on a dynamic scenario where resources pool might change at runtime . This second policy will become fundamental when porting the Orchestrator mechanism inside a virtual machine (VM) with the possibility to change the resource attached to a VM at runtime.

IV. EVALUATION

In this section we first evaluate the overheads introduced by our runtime system and after we present proof-of-concept results of the overall SAVEHSA approach. The evaluation is performed on our heterogeneous server platform with a real-world application from the financial analysis domain(BSOP). The Black-Scholes Option Pricing (BSOP) [10] is a widely utilized measure of the risk of loss on a portfolio of financial assets. We put together the heterogeneous platform with up to date off-the-shelf components to implement the proposed concepts, i.e. CPU, GPU, DFE and MIC. The system comprises two Intel Xeon CPUs with 32GB main memory. Additionally it features three accelerators, a GPGPU (NVidia K20 Tesla), a DFE (Xilinx Virtex-6 SX475 FPGA) and a MIC (Intel Xeon Phi 31S1P), connected via PCI Express.

Overheads*:* Our runtime system needs to analyze and modify the application before starting the execution. These steps, compared to a native execution without the runtime features, introduce overheads to the overall execution. We added a timer into the system and sampled it at the five main phases. Figure 3 shows the consumed time of each phase. Most of the time is spent in the initialization of the execution environment (mainly setting up the JIT compilation of LLVM, 55%) and in the analysis & optimization of the application (44%). The other steps together add up to less than 1%. Overall, the introduced overhead is very low (about 182ms) and negligible when our approach is able to match the hotspot to an accelerated function and offload it.

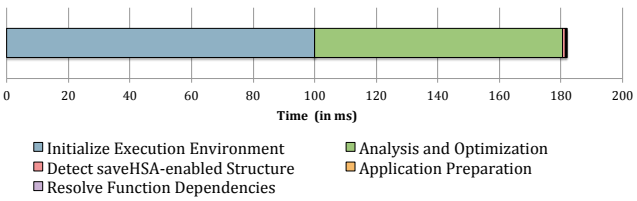


Fig. 3. Housekeeping overheads incurred before application start-up.

Runtime System*:* For our prototypical implementation and its evaluation, we target the GPU (in addition to the CPU). The focus is on the execution of the computationally intensive (hot) loop as other factors like data initialization and storing of results are completely equal. The overheads of our approach are already presented in the previous subsection and the overheads introduced by the Heartbeat framework are discussed in [11].

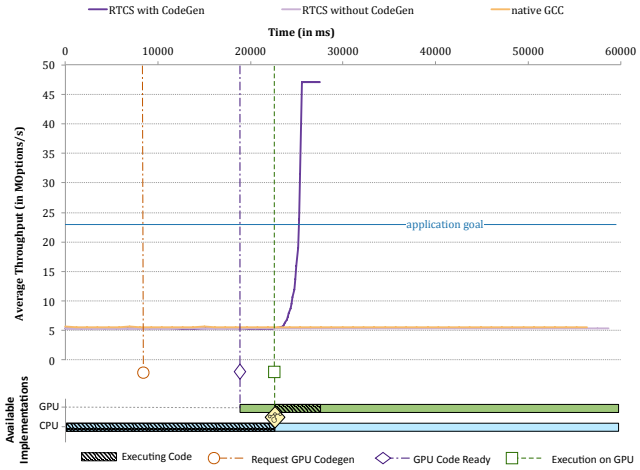


Fig. 4. Hotspot execution of a SAVE-enabled application.

We consider three scenarios of the SAVEHSA-enabled BSOP application, summarized in Figure 4. The top half represents the average throughput vs the loop execution time. The bottom section represents the execution time-line when the BSOP application is run using our proposed approach. This time-line tells us when the GPU implementation was made available to the application (at runtime) and when the Orchestrator decides to switch to it. The vertical dotted lines represent the key events during the life time of the application.

The first scenario is the execution of `native GCC`. Here, the application is run natively with CPU only implementation i.e. it is compiled with `gcc` v4.8.2 using optimization level `-O2`. This provides us with a baseline against which we compare the RTCS-supported versions. In the second scenario, we consider `RTCS without CodeGen`. Here, the application is executed using the RTCS infrastructure, however the RTCS does not expose its code generation capabilities. In this case, the application has no alternate implementation available and is forced to run only on the CPU. It is interesting to note that if we compare the execution times of the first and second scenario we see that the execution time is slightly larger for the RTCS approach (less than 4%). We attribute this increase in execution time to the compilers (gcc vs llvm) . Finally, in the third scenario, we exploit the full potential of the RTCS by executing the `RTCS with CodeGen`. If we look at Figure 4, we can see that initially the application is executing on the CPU, however it is not satisfying the application goal (24 MOptions/s). The Orchestrator observes this and requests the RTCS to generate code for the GPU. Once the code is generated and integrated into the application at runtime, the Orchestrator then switches execution (based on its policies) to the GPU. Once the application starts executing on the GPU, we can observe an increase in the average throughput of the application. This in turn improves the application performance and reduces the execution time.

V. Conclusion and Future Work

In this paper we presented the SAVEHSA architecture, a runtime resource management approach for heterogeneous systems. We described what a SAVEHSA-enabled application is and what goals can be specified. We also looked at the two main components of our approach, the RTCS and the Orchestrator. We saw that the RTCS is able to analyze the application and determine what type of implementations are already present and in turn propose just-in-time (JIT) code generation possibilities for the missing ones. We also looked at the Orchestrator and how it is used as a runtime resource manager for heterogeneous systems. And how it interacts with the RTCS by instructing it to generate code for missing implementations at runtime. We showed that once the code for a specific implementation was generated, the Orchestrator switches to this newly available implementation (based on its policies) in order to satisfy the application goals. We looked at the overheads associated with our approach and showed that it was negligible (less than 1%). We finally evaluated our approach with a real-world application from the financial analysis domain. We demonstrated that by JIT generation of code and later executing the application on the GPU, our approach enables the application to satisfy its throughput goals. In this work, we make use of pre-compiled GPU kernels. The next steps would be to generate OpenCL kernels and explore code generation for the FPGA and MIC architectures.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901) and the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE).

References

[1] 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.

[2] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, “Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments,” in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC ’10, 2010, pp. 79–88.

[3] 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.

[4] 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.

[5] G. Vaz, H. Riebler, T. Kenter, and C. Plessl, “Deferring accelerator offloading decisions to application runtime,” in *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2014, pp. 1–8.

[6] 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.

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

[8] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. Int. Symp. Code Generation and Optimization*, 2004, pp. 75–86.

[9] 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.

[10] F. Black and M. Scholes, “The pricing of options and corporate liabilities,” *The journal of political economy*, pp. 637–654, 1973.

[11] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, “Decision making in autonomic computing systems: Comparison of approaches and techniques,” in *Proc. of ACM Int. Conf. on Autonomic Computing (ICAC)*. NY, USA: ACM, 2011, pp. 201–204.

Efficient Dispatching to Co-processors over PCIe

D.Bakoyannis[1], O. Tomoutzoglou[1], G.Kornaros[1] and M.Coppola[2]
[1]Technological Educational Institute of Crete, Informatics Engineering Dept., Heraklion, Crete, Greece
[2]STMicroelectronics, Grenoble, France

***Abstract*—as FPGA technology provides immense computational capacity for hosting specialized custom hardware application co-processors, heterogeneous architectures emerge challenging the design of various organizations and methods to provide efficient and adaptive off-chip acceleration over PCI Express infrastructure. In this work we present the synergy of software and hardware methods to develop systems that include hardware co-processors accessible through PCIe Gen2 bridging, with a multitude of software controlled DMA engines and monitoring probes, which provide both self-adaptive control and host-based tightly-managed control. We present several different methodologies for effective job dispatching to the co-processors.**

I. INTRODUCTION

With the emerging FPGA technology which is known for providing better performance per watt over traditional off-the-shelf processors has made their usage appealing in life sciences, big data, security, and other HPC-related areas. In addition, modern design flows (e.g. High-Level-Synthesis) make application-specific systems to deliver the highest performance and/or lowest resource utilization. However, efficient computation offloading in terms of performance and latency requires carefully designed schemes in order not to become drawback to the efficiency of the hardware accelerators.

In this paper we propose methodologies that exploit various approaches for transferring data through the software and hardware layers in HPC systems that employ hardware accelerators over PCIe. On top we integrate particular hooks in the custom accelerators to enable dynamic monitoring and adaptations of system parameters, in order to give the user/programmer the capability to optimize the utilization, performance and power efficiency for the acceleration processes. Effectiveness is achieved by exposing the capability of the hardware and software components of our system to the user- and kernel-level. Finding the golden edge for boosting performance and throughput is a matter of careful selection of the appropriate features of the initiators for data transfers (i.e. DMA engines, processors, hardware peripherals in the host and in the reconfigurable co-processors) and communication with the userspace memory and the kernel space.

Past works also target to provide communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. RIFFA 2.0[1] uses PCIe to connect FPGAs to a CPU's system bus. RIFFA 2.0 extends the original RIFFA project by supporting more classes of Xilinx FPGAs, multiple FPGAs in a system, more PCIe link configurations, higher bandwidth, and Linux and Windows operating systems. In SAccO[2], the portable and scalable accelerators are implemented on standard FPGA boards connected via PCIe for accelerating parts of multi-process streaming applications [2], where a high-level communication API is provided which uses the same function calls for SW-SW communication via sockets and SW-HW communication via PCIe. Instead, we address the design of high performance off-chip communication bridges for connecting the host processor(s) to co-processors.

II. DISPATCHING TO PCIE-CONNECTED CO-PROCESSORS

In this section we consider dispatching computing kernels to discrete co-processors. Generally, a co-processor includes one or more application engines that can be dynamically configured to implement a desired functionality. For instance, application engine, which comprise one or more FPGAs, may implement vector processor functionality. Vector processor functions or instruction sets provide efficient high level operations on vectors that are very useful in specific application domains such as scientific applications, or image processing where a large amount of data has to be processed in a repetitive manner. These application domains are characterized by the fact that the computation of each result is independent from the computation of previous results. In this case application engines can be dynamically reconfigured to implement a multitude of vector processing instruction sets. Typical operations might add two 64-entry, floating point vectors to obtain a single 64-entry vector. Generally co-processors also include a common management infrastructure that remains common across the different runtime configurations. Co-processors such as the ones provided by Maxeller or Convey[5], are not sharing the same physical memory of the host processor, but they include a physical memory infrastructure implemented by different memory banks controlled by one or more memory controllers. That can be accessed over PCIe interfaces. Hence, we address the design of high performance off-chip communication bridges for connecting the host processor(s) to co-processors. The goal is to include (i) efficient mapping of the host's address space to co-processor' address space and support for virtualized address ranges, (ii) concurrent communication from the host to the application engine and vice-versa addressing PCIe and AXI4 protocol intricacies and (iii) provide support for monitoring and control of the resources of the accelerator subsystem, exposed to the user applications executing on the host.

New “fused” CPU-GPU architectures that have emerged, i.e., accelerated processing units (APUs), such as AMD Fusion, alleviate the PCIe bottleneck by placing the CPU and GPU on the same die, interconnected with high-speed memory controllers. Even though techniques show that when the memory accesses of GPU threads hit in the L2/L3 cache their latency can be drastically reduced, OS overheads are still present and addressed by our work in the previous sections.

Currently, in configurations that use discrete co-processors over PCIe, transmission of data from accelerator memory must be done by explicitly copying data to host memory before performing any communication operations. This process impacts productivity and can lead to a severe loss in performance. Thus, researchers traditionally target large kernels to amortize the overhead of offloading, and exploit throughput-oriented code which can more easily hide the latency and variability of each individual operation.

In this section we present different methods for transferring computation kernel jobs/data through the software and hardware layers in systems that employ co-processors over PCIe. On top, we integrate particular hooks in the custom co-processor to enable dynamic monitoring and adaptations of system parameters, in order to give the user/programmer the capability to optimize the utilization, performance and power efficiency for the acceleration processes. Thus, effectiveness is achieved by exposing the capabilities of the hardware and software components of our system to the user- and kernel-level. Similar to recent works [5], computing island throughput can be optimized by balancing data placement across the system memories based on the aggregate memory bandwidth available in a system.

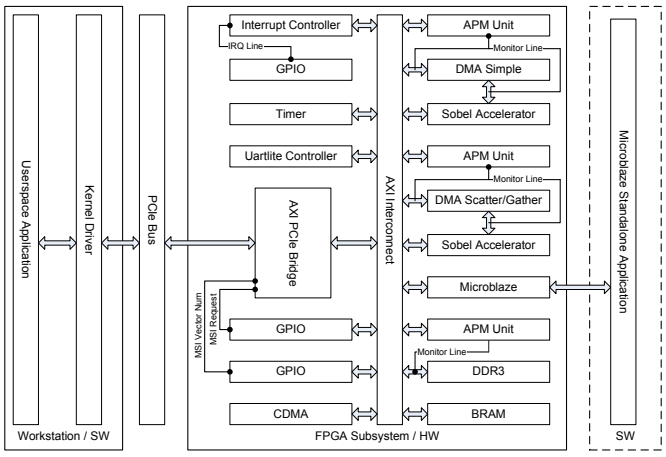


Fig. 1: Layout of system architecture using PCIe-connected FPGA accelerator

We consider several different approaches on how data are transferred through the software and hardware layers in order for the co-processor to process them. The developed platform, also, supports performance measurement units that gather metrics results for each different method of data transfers that is applied for hardware acceleration. In this context a prototype platform is developed using Xilinx’s AXI Memory Mapped to PCI Express core on a Kintex FPGA device that is accessed by a Xeon-based host system through a PCIe Gen2 connection [3]. The subsystem comprises of an application engine that is implemented by a Microblaze-based processor and it is used to execute custom kernels, such as two Sobel Edge Detection functionality, along with different memory-mapped peripherals, DDR3, BRAM, CDMA, GPIOs and timers. AXI-based Performance Monitor (APM) units are also integrated to provide dynamic monitoring services.

The two Sobel Edge Detection cores process image data in streaming mode. The subsystem incorporates two special DMA cores for making large transfers from a memory to another. In this implementation the DMA is dedicated for streaming image data to and from the Sobel Edge Detection accelerator core. It, actually, makes AXI read burst transactions from a given memory address where the image is located, streams the image data over the AXI stream interface of the Sobel Edge Detection core and finally makes AXI write burst transactions to transfer the processed image to another given memory address or to an offset of the previous memory. The second DMA core is used for exercising transfers in Scatter/Gather mode. The CDMA core (central direct memory access) functions similarly to the DMA cores. It can make large transfers but only by making AXI burst transactions between memories or even registers. In our implementation it is, mostly, used to transfer image data from the memory space of the kernel driver to the subsystem’s DDR3 memory and vice-versa. The three GPIO modules are used to trigger interrupts that either target the kernel driver and the user-space application or the Microblaze processor. Two of these are wired with the PCIe Bridge so that to trigger MSI interrupts from the PCIe Bridge to the kernel driver. The third GPIO is used to enable the kernel driver to interrupt the Microblaze processor, since the PCIe Bridge does not support interrupts in this direction. The AXI Performance Monitor Units (APMs) are used for capturing transactions, events and provide metrics such as the number of read/write transactions, the number of transferred bytes, the total time in cycles for a given transaction, latency and idle cycles between transactions. The first APM captures events from transactions over the subsystem’s DDR3 memory while the second APM captures events of the DMA core.

A. Driver and Virtualization of Co-processors

The driver maps the subsystem’s memories and registers to the Linux memory space, thus, it provides direct access to most of the AXI components that are part of the subsystem. The driver, also, provides a more abstract way for the userspace application to communicate with the AXI components. The mapping of the memories and registers of the subsystem is achieved through querying the *Base Address Registers* (BARs) of the PCIe configuration space inside the PCIe Bridge for the available resources of the AXI subsystem.

The current PCIe Bridge supports three BARs. The three BARs of the PCIe bridge configuration space are used to map

the memories and several peripherals of the external subsystem. The first BAR represents the internal co-processor AXI address space where most of the AXI peripherals are mapped internally, thus, it supports access to their registers for direct control operations on the application engines and other subsystems. The first BAR could be considered as partitioned in as many fragments as needed for every component that should be accessed by the outer world. The second BAR refers to the 512K BRAM memory of the co-processor while the third BAR provides access to the 512M DDR3 memory. The address ranges of the co-processor components that should be accessed by the kernel driver are located in contiguous address space totaling 256K of address space that is mapped inside the first BARs 512K memory space. The rest 256K segment is reserved for future components. This way, the BIOS, the operating system and the driver is aware of one piece of memory region for the first BAR. Once the driver needs to access any component of the co-processor it uses offsets of the base virtual address that is assigned for the first BAR.

Each contiguous address space of the co-processor that is exposed to the host system can be partitioned and exposed by a Virtual Machine Manager (VMM) to guests. Processes from different guest systems can concurrently and directly access the assigned partition (marked by VMID or VMID/PID pair) without scheduling by the VMM. The VMM is only responsible to manage the initial setup; on top the VMM must dynamically configure through the second BAR mapping the allocation of a partition to a particular VMID. Building virtualization support is a speculative mechanism enqueueing work requests on the device and an optimized set of queues. The queue set can be separated into exclusive queues in main memory and one shared queue on the device. With main memory as cost-effective resource the exclusive queues scale with the number of processes.

III. REALIZATION TECHNIQUES

The methods and configurations that are described herein, regard the different ways that data are transferred through the software and hardware layers before and after they are processed by the co-processor. The goals are to demonstrate the highest possible achieved throughput capabilities of data transfers and present the pros and cons when implementing various approaches of data transfers while offering insight to user-applications for static and runtime optimizations. The different methodologies are applied by employing Sobel edge detection on images.

Two main sets of methods are employed. The first set/group demands memory allocations both in user-space and kernel space. As a consequence, this group suffers more copies since image data have to be transferred through both memories before they reach the DDR3 memory of the subsystem. The second set of methods only uses memory allocations in kernel space, thus reducing copies between the kernel space and the DDR3. Both groups use three different transfer initiators to move image data from kernel space to DDR3 and vice-versa before the DMA gets access to the latter and supplies the accelerator with those data. The first initiator is the driver which can perform write or read 8 byte transactions. The second is CDMA that is capable of making burst transfers core and the third is write and read 4 byte transfers from the Microblaze processor. We have implemented in both groups one last approach which is even less demanding in memory as it does not utilize the DDR3, thus making the system more efficient by avoiding extra copies. In this approach, the DMA supplies the sobel accelerator with image data directly from the kernel memory space. Table I summarizes the developed methods. The first group includes memory allocation in both the userspace application and the kernel driver.

TABLE I. MEMORIES, TRANSFER INITIATORS AND ABBREVIATIONS FOR THE 8 METHODOLOGIES

Group	Methodology	Abbreviation	Involved Memories			Transfer over PCIe Initiator			
			User Space	Kernel Space	Board DDR3	Driver	Microblaze	CDMA	DMA
1	1	G1M1	✓	✓	✓	✓			
1	2	G1M2	✓	✓	✓			✓	
1	3	G1M3	✓	✓	✓		✓		
1	4	G1M4	✓	✓					✓
2	1	G2M1		✓	✓	✓			
2	2	G2M2		✓	✓			✓	
2	3	G2M3		✓	✓		✓		
2	4	G2M4		✓					✓

In the first method the userspace application invokes the driver which copies the image data from the userspace memory to the driver's kernel memory and then writes data to the DDR3 of the co-processor over the PCIe Bridge. In the second and third methodologies instead, ioctl system calls are used to command the driver to copy the image data from the userspace memory to the kernel memory; then, the application issues another ioctl system call so that the driver interrupts the Microblaze of the co-processor, which, on being interrupted, either triggers the CDMA core to to fetch the image data from the kernel space to the local DDR3 (second method) or performs read transfers itself for the same reason (third method). The Microblaze performs a MSI interrupt over PCIe to the kernel driver that calls its interrupt handling routine which in turn signals the userspace application in order to inform the latter that the image data are processed. In the fourth scenario a DMA core is allocated by the host kernel driver to fetch the image data directly from the kernel space to the hardware accelerator rather than involving any DDR3 memory in the co-processor.

Focusing now on the second group of methods a different approach is employed so that no userspace memory allocation is required. This is achieved by taking advantage of the *mmap* capability of the driver that maps a kernel space memory to the

userspace. Consequently, the application can load the image data directly to the kernel space avoiding additional copies. Initially, the application performs a system call to get the driver assistance to allocate un-cacheable kernel memory. Then, by issuing the mmap call the driver's memory is accessible to userspace. One constrain is that for large workloads the userspace application allocates two memory areas in chunks of 4MB size that are used to store the pre-processed and the post-processed image data respectively[1]. Two memory areas are used to store the pre-processed and the post-processed image data respectively.

TABLE II. COMPARATIVE SUMMARY OF LATENCIES USING DIFFERENT OFFLOADING METHODOLOGIES

Time (μs)	Methodologies							
	G1M1	G1M2	G1M3	G1M4	G2M1	G2M2	G2M3	G2M4
Total	797791.44	194906.332	3231345.29	179886.02	795534.72	193726.31	3229588.99	178641.64
Round Trip	768539.92	165949.36	3201583.77	150554.7	765972.35	164693.78	3200122.5	69177.53
Userspace malloc	3.26	7.3	11.4	3.27	N/A	N/A	N/A	N/A
Kernel kmalloc	4.06	348.08	605.13	361.74	298.17	334.41	488.43	280.18
Mmap	N/A	N/A	N/A	N/A	10.16	11.51	12.09	8.21
Load Image	29228.21	28595.32	29138.28	28963.93	29195.81	28631.68	28905.05	28930.9
Copy from User	322.27	431.94	336.69	328.36	N/A	N/A	N/A	N/A
Kernel to DDR3 Transfer	48721.12	7765.9	1772016.82	N/A	49047.88	7793.3	1771723.28	N/A
DDR3 to Kernel Transfer	649516.93	7558.97	1279041.26	N/A	647750.64	7558.99	1279053.89	N/A
Copy to User	804.88	832.14	827.19	867.91	N/A	N/A	N/A	N/A
DMA Total DDR3 Access	69125.75	69125.75	69125.76	N/A	69125.76	69125.76	69125.75	N/A
DMA total Kernel Access	N/A	N/A	N/A	69161.2	N/A	N/A	N/A	69175.29

In scenarios that the CDMA is utilized the Microblaze requires the physical address of the kernel's memory in order to read the image data. The kernel driver obtains the physical address writes it directly to the AXI address translation register of the PCIe Bridge and then the CDMA and the Microblaze can target the AXI address space of the PCIe Bridge which is mapped to the physical address of the allocated memory of the host system.

The different dispatching methods were tested on bitmap images of 1024x768 resolution, or 3MB size (4 bytes per pixel). On one hand, small size images was proven to give results of large variability, while on the other hand images larger than 4M cannot be supported due to kernel driver's memory allocation size restrictions. The measurements for all methods are averaged for 20 iterations. The measurements on behalf of the userspace application and the kernel driver for the preparation time refer to the necessary actions before initiating the acceleration process. These involve: (i) memory allocation in the userspace application, (ii) memory allocation in the kernel driver (kmalloc() / dma_alloc_coherent()), (iii) memory map of kernel memory to userspace (mmap), (iv) initialization of image data from image to the kernel or the application memory. The time cost for G1M1 is 4.08μs when using kmalloc. Utilizing the dma_alloc_coherent call for the rest methodologies has significant impact in time cost comparing to kmalloc, which ranges from 298.17μs to 605.13μs. The collected measurements are affected by the configuration/ number of PCIe TLPs (Transaction Layer Packets) (we used a single PCIe lane) which is related to the transfer capacity of the driver, by the configuration of the CDMA core, the DMA transaction burst size, the accelerator operation frequency and the AXI interface configuration and finally, by the system calls and interrupts latency. On the basis of captured measurements the optimum throughput is given by the CDMA which is 3175.02 Mbps for memory writes and 3048.98 Mbps for memory reads. With regard to energy and memory utilization though, the G2M4 methodology, which utilizes the lower throughput DMA core, is assessed as the optimum solution since it involves less memory usage and less data transfers for accelerating image edge detection.

1 The size restriction is due to the fact that the driver cannot allocate contiguous memory areas larger than 4MB.

IV. CONCLUSION

In this paper, we discussed synergies of software methods and of reconfigurable hardware that is intended to provide off-chip acceleration over PCIe Gen2 bridging. We expose the capabilities of our system by utilizing the hardware and software components to bring several methodologies to optimize the acceleration and data transfer and evaluate performance by taking into consideration the measurements results that were collected by the utilized monitoring probes. Using a CDMA controller is proved to provide the maximum throughput however the G2M4 methodology, which utilizes the lower throughput DMA core, involves less memory usage and less data transfers for acceleration.

ACKNOWLEDGMENT

This work is partially supported by the EU through the FP7 SAVE project (GA No. 610996)

REFERENCES

[1] Matthew Jacobsen, Ryan Kastner, "*RIFFA 2.0: A reusable integration framework for FPGA accelerators",* In Proc. FPL 2013, pp 1-8

[2] Markus Weinhardt, Bernhard Lang, Frank M. Thiesing, Alexander Krieger, "SAccO: An implementation platform for scalable FPGA accelerators", Microprocessors and Microsystems - Embedded Hardware Design 39(7), 543-552, 2015

[3] Xilinx. AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.6 Logicore IP Product Guide (PG055), June 2015.

[4] Convey Computer Co. "Convey Wolverine® Application Accelerator", www.conveycomputer.com

[5] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S.W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems", In Proc. ASPLOS '15, pp. 607-618

4 Presentation feedback

These papers are related to three of the most important technologies developed for the SAVE project namely, the Orchestrator, the run time compilation system (RTCS) and the general purpose processing unit (GPPU). In addition they demonstrate a strong cooperation between the partners leading to the joint scientific dissemination. The audience expressed high interest on the presented topics, formulating pertinent and interesting questions. Examples of the received comments are the following ones:

- What are the limitations in terms of supported programming languages for run-time allocation of the computing resources?
- Why do we RTCS uses just-in-time compilation? Will it pay off to JIT compile code for a FPGA?
- What is the RTCS input format? C source code or LLVM IR?

Each of these questions have been addressed and taken into account by the partners to reflect on how to improve their technology, and create new research streams for future projects proposals or enhancements for commercial products.