A Framework for Predictable Hardware/Software Component Reconfiguration

João Gabriel Reis[1], **Eduardo Augusto Bezerra (Advisor)**[2], **Antônio Augusto Fröhlich (Co-advisor)**[1]

[1] Software/Hardware Integration Laboratory – Federal University of Santa Catarina
Florianópolis – SC – Brazil

[2]Embedded Systems Group – Federal University of Santa Catarina
Florianópolis – SC – Brazil

jgreis@lisha.ufsc.br, eduardo.bezerra@eel.ufsc.br, guto@lisha.ufsc.br

Abstract. *The current pace of innovation in computing makes it difficult to assume a fixed set of requirements for the whole life span of a system. Aggressive technology scaling also imposes additional constraints to modern hardware platforms. Field-Programmable Gate Array (FPGA) reconfiguration can help systems cope with dynamic requirements such as performance and power, hardware defects due to Negative-Bias Temperature Instability (NBTI) and Process, Voltage and Temperature (PVT) variations, or application requirements unforeseen at design time. This work proposes a framework for reconfigurable components whereby the reconfiguration of a component implementation is performed transparently without user intervention. The reconfiguration process is confined in system's idle time without interfering with or being interfered by other activities occurring in the system or even peripherals performing I/O. A telecommunications switch was used as a case study for the deployment of reconfigurable components as well as the impact I/O interference has in the process and to explore non-functional trade-offs between implementations.*

1. Introduction

Many modern computing application scenarios, including smartphones, autonomous vehicles, and IoT devices, evolve so quickly that it is no longer viable to define a fixed set of requirements for the whole design process. Increasingly, systems are having to adapt themselves on-the-fly in order to cope with varying (and sometimes conflicting) requirements. Additionally, these scenarios are being enabled by aggressive technology scaling, which, besides producing low-cost, high-performance systems, also brings about a series of new issues such as process variation, failure of Dennard's law, and emergence of dark silicon [Taylor 2012, Rahimi et al. 2015]. These systems must constantly monitor and adapt themselves in order to be energy-efficient and to wear out evenly, being often called *self-aware* [Sarma and Dutt 2014, Falaki 2012, Pant et al. 2012, Donyanavard et al. 2016, Rahimi et al. 2015, Wanner and Srivastava 2014]. In this context, rigidly partitioning components between software and hardware inevitably leads to sub-optimal results, once the design space for each component becomes constrained by other component's requirements, by system-wide requirements, and by the overall availability of resources, even if those components are not meant to be used simultaneously.

Adapting components at run-time is an alternative to cope with dynamically changing requirements such as performance and power [Li et al. 2013]. Adaptability also improves tolerance to hardware faults, particularly those arising from Negative-Bias Temperature Instability (NBTI) and Process, Voltage and Temperature (PVT) variations [Martins et al. 2015]. An adaptive component architecture enables each individual component to exist in multiple implementations, each encompassing a specific compromise between the quality of the functionality being provided and the resources needed to provide it [Reis et al. 2015]. For instance, a component may coexist as a sequential software to be run in a single CPU, as a parallel software to be run on a multicore CPU or on a GPU, as hardware to be instantiated on an FPGA, or as a remote (web) service. With proper run-time support, different implementations of a component can be activated along the time, enabling the system as a whole to cope with the demands presented at each moment. Multimedia, vision, and physical simulations are good example of subsystems that make use of components that can be adapted in this sense.

2. Contributions

This work proposes a framework whereby reconfiguring the current component implementation is a deterministic process performed without modifying its interface. The reconfiguration is a deterministic process aiming not to disrupt critical system activities. We consider that a reconfigurable component can have multiple implementations with different functional, quality, and business trade-offs. The framework delivers a tailored wrapper for each component according to the number of implementations it has. Each implementation can use different resources in different substrate (e.g. processor core, Field-Programmable Gate Array (FPGA)) and better suit the application at given moment according to the embedded system environmental conditions. Reconfigurable components have a single interface such that from the application point of view, components implementation using software or reconfigurable hardware resources can be invoked seamlessly. The main contributions of this work are:

- The reconfigurable components implementations deployed in the framework are designed following Application-Driven Embedded System Design (ADESD) techniques [Fröhlich 2001], a domain engineering methodology.
- The process of reconfiguring the component implementation is transparent from the application point of view, confined in the system's idle time, and designed not to interfere with critical threads executed by the system.
- Despite being temporally isolated from other threads, the reconfiguration can have its timing determinism compromised when interfered by background I/O operations, specially when a new implementation must be reconfigured in an FPGA.

3. Proposal

In Chapter 3 it is shown how the syntax and semantics of the component interface in our system are preserved across the multiple implementations, such that an application sees no difference (other than changes in quality and cost associated with the component usage) when implementations are changed. The adaptation system is aware of the system behavior by means of models of system and components behavior (e.g. performance, temperature, aging). By feeding those models with data from sensors captured during run-time, it is possible to predict if by reconfiguring a component, the system can

improve a system-level metric (e.g. throughput, energy, temperature) and delegate the action to a manager. Multiple implementations can be available through a pool from which the reconfiguration system can chose based their inherent characteristics. The process is depicted in Figure 1.

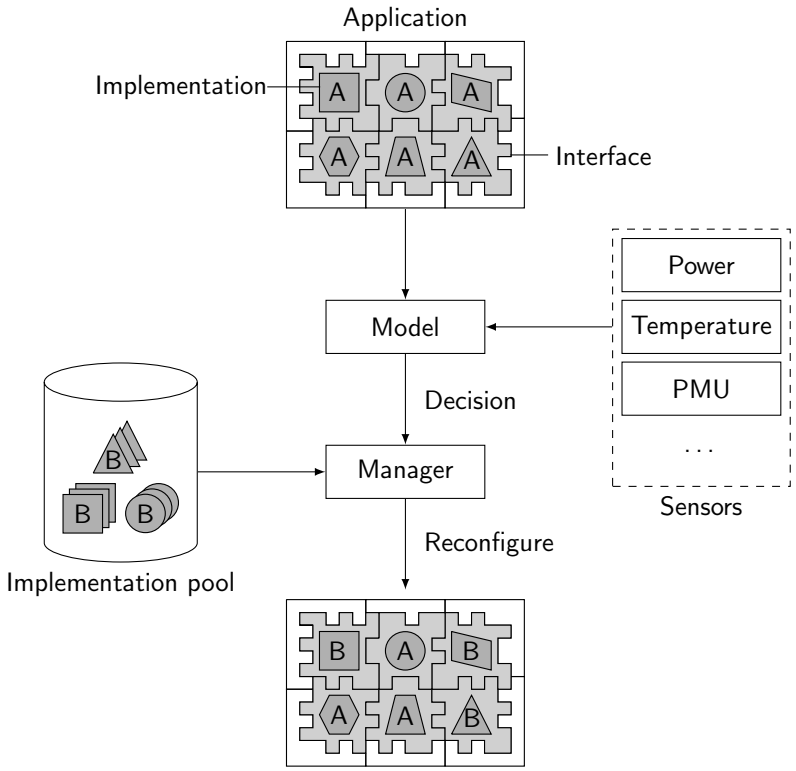

Figure 1. Components reconfiguration. Multiple implementation are available in a pool and can be deployed according decisions based on models inferring system behavior from on sensor data.

The process of reconfiguring the component implementation, presented in Chapter 5, is transparent from the application point of view, confined in the system's idle time, and designed not to interfere with critical threads executed by the system. By delegating reconfiguration to lower-level software layers that can reason on the system's current state, the reconfiguration process becomes transparent from the application point of view as the application programmer does not have to be aware that the reconfiguration is happening. The reconfiguration process of each component is divided into small tasklets such that its largest atomic step can typically be performed within available system slack as long as processor utilization is under $100\,\%$. Performing reconfiguration only when the system is idle allow the systems to temporally isolate other critical operations from it and mitigate the inflicted interference. The reconfiguration process time takes extra care for I/O interference from external system components in a speculative fashion by monitoring system execution.

Despite being temporally isolated from other threads, the reconfiguration can have its timing determinism compromised when interfered by background I/O operations, specially when a new implementation must be reconfigured in an FPGA. A large FPGA bitstream used to store its configuration must be moved from memory to the FPGA recon-

figuration interface while, for example, a video stream is being moved from the memory to a video interface. The sharing of hardware resources (interconnects and memory interfaces) by both streams of data results in an increased execution time for both operations due to interconnect scheduling policies and limited memory bandwidth. Our reconfiguration process speculatively monitors the I/O traffic sources to predict when to deploy FPGA reconfiguration without the hazard of being interfered by other peripherals. It is also capable of powering down devices being used by non-critical threads to reduce I/O interference and prioritize the reconfiguration. In Figure 2 we present how our approach can cope with interference by isolating the reconfiguration process. Peripherals being used by non-critical threads can still interfere with the reconfiguration process are suspended as shown in Figure 2a. Figure 2b shows that the I/O activity performed by the reconfiguration interface is confined in the idle thread with the supporting reconfiguration operations and does not interfere with other I/O controllers.

In Chapter 4, we also propose a static performance-estimation technique for the time delay caused by I/O interference when moving data through shared chip resources such as interconnects and memory controllers. The model uses concepts from queuing theory and estimates the average time each peripheral data transaction has to wait on shared resources. The higher latency and throughput degradation is due to other peripherals' transactions that are passing through the same shared resources that queue the transactions before forwarding them. Such model is specially helpful to illustrate the increase in the FPGA reconfiguration time (specially bitstream loading) when multiple peripherals are contending for system I/O resources. It is also meant to be used as a design tool for application designers to ensure a minimal interference between threads that depend on I/O resources.

4. Concluding remarks and work impact

This work presented a transparent framework for reconfigurable computing geared towards the application programmer. Reconfigurable components interfaces may be realized through many different implementations, ranging from high-quality software versions to software approximations, cloud offloaders, and hardware accelerators. While the syntax and semantics of reconfigurable components interface is preserved across the different implementations, the system may at any time pick any of the implementations that suits better for the current execution context. The framework manages the whole reconfiguration process and ensures that it can be used in critical systems without interfering in its time constraints. With our reconfiguration mechanism, a task set schedulable on a reconfigurable fabric large enough to accommodate at the same time all hardware components used by its tasks, will still be schedulable on a smaller reconfigurable fabric where only some components can be simultaneously instantiated in hardware. The remaining components are momentarily deployed in software without compromising to the tasks requirements since all activities pertaining reconfiguration are performed within the slack time and made aware of I/O interference.

The work resulted in the following publications:

- *OS Support for Adaptive Components in Self-aware Systems*, In: ACM SIGOPS Operating Systems Review. To appear.
- *A Framework for Dynamic Real-Time Reconfiguration*, In: 18th Euromicro Conference on Digital Systems Design, 2015.

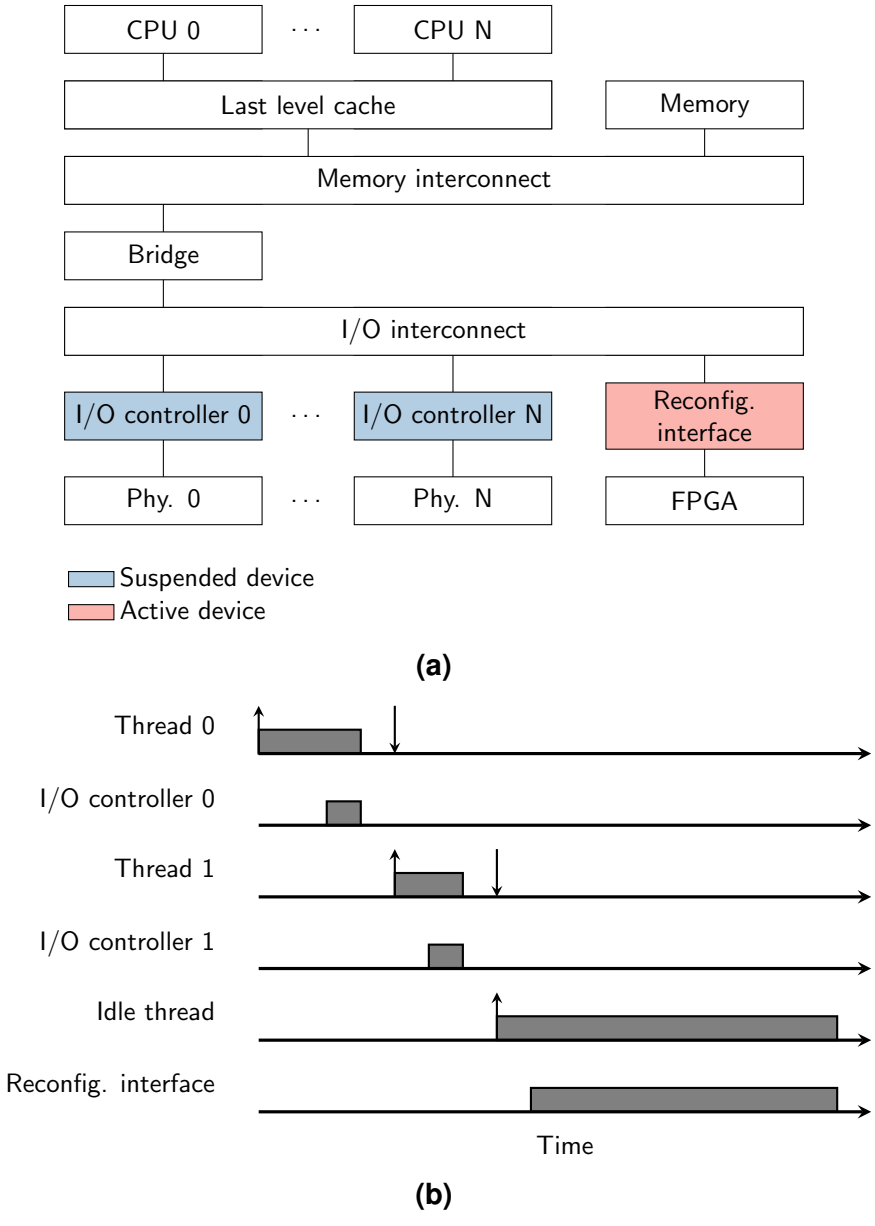

Figure 2. a) I/O controllers are suspended to prioritize the reconfiguration of a component that benefits a more critical thread. b) Reconfiguration is confined in the idle thread and thus does not interfere and is not interfered by other threads.

- *X-Ware: Mutant Computing Substrates*, In: 26th IEEE International Symposium on Rapid System Prototyping, 2015.
- *On the FPGA Dynamic Partial Reconfiguration Interference on Real-Time Systems*, In: 5th Brazilian Symposium on Computing Systems Engineering, 2015. **Best paper award**.
- *Mutant Components: Efficiently Managing Multiple Implementations*, In: 6th Brazilian Symposium on Computing Systems Engineering, 2016. **Best paper award**.

References

Donyanavard, B., Mück, T., Sarma, S., and Dutt, N. (2016). SPARTA: Runtime task allocation for energy efficient heterogeneous many-cores. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, pages 27:1–27:10.

Falaki, H. (2012). *Automating Personalized Battery Management on Smartphones*. PhD thesis, UCLA.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Li, Y., Jia, Z., Xie, S., and Liu, F. (2013). Dynamically reconfigurable hardware with a novel scheduling strategy in energy-harvesting sensor networks. *IEEE Sensors Journal*, 13(5):2032–2038.

Martins, V. M. G., Villa, P. R. C., Neto, H. C. C., and Bezerra, E. A. (2015). A TMR strategy with enhanced dependability features based on a partial reconfiguration flow. In *Proc. IEEE Computer Society Annual Symposium on VLSI*, pages 161–166.

Pant, A., Gupta, P., and van der Schaar, M. (2012). AppAdapt: Opportunistic application adaptation to compensate hardware variation. *IEEE Transactions on Very Large Scale Integration Systems*, 20(11):1986–1996.

Rahimi, A., Cesarini, D., Marongiu, A., Gupta, R. K., and Benini, L. (2015). Task scheduling strategies to mitigate hardware variability in embedded shared memory clusters. In *Proc. Design Automation Conference (DAC)'15*, DAC'15, pages 152:1–152:6, New York, NY, USA. ACM.

Reis, J. G., Wanner, L. F., and Fröhlich, A. A. (2015). X-Ware: Mutant computing substrates. In *Proc. IEEE International Symposium on Rapid System Prototyping (RSP'15)*, Amsterdam.

Sarma, S. and Dutt, N. (2014). FPGA emulation and prototyping of a cyberphysical-system-on-chip (cpsoc). In *Proc. IEEE International Symposium on Rapid System Prototyping (RSP'14)*, pages 121–127.

Taylor, M. B. (2012). Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. Design Automation Conference*, pages 1131–1136.

Wanner, L. and Srivastava, M. (2014). ViRUS: Virtual function replacement under stress. In *Proc. USENIX Conference on Power-Aware Computing and Systems (HotPower'14)*, HotPower'14. USENIX.