# On Operating Systems for Reconfigurable Computing

Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
PO Box 476 – 88049-900 – Florianópolis, SC, Brazil
guto@lisha.ufsc.br

## Abstract

*Dynamically reconfigurable hardware architectures are the substratum of **reconfigurable computing**. As such architectures allow for the reprogramming of some hardware building blocks while others continue to operate normally, they bring about much of the flexibility usually associated with software also to the hardware realm. Nonetheless, in order to achieve such flexibility: a complete hardware/software infrastructure is necessary.*

*In this essay, we discuss several issues concerning operating systems for reconfigurable computing, aiming at identifying concepts and mechanisms of traditional operating systems that can be reused or adapted to this novel context, and also proposing answers for some questions that cannot be satisfactorily answered based on current technology.*

**Keywords:** *reconfigurable computing, hardware-software co-design, morphware, operating systems.*

## 1 Introduction

The forthcoming of dynamically reconfigurable hardware architectures is motivating several research projects and is coining a new term: **reconfigurable computing**. In currently available Field-Programmable Gate Arrays (FPGAs), elementary functional units physically implemented in hardware can be grouped together, or programmed, to shape more sophisticated components, whose purpose is dictated by particular system needs. Some FPGAs allow for partial reprogramming of specific blocks, while others retain their original program, thus leading to the notion of "reconfigurable computing", in which the computer as a whole, software and hardware, can be dynamically adapted according to varying application requirements during its operation time [5, 9, 7]. More recent architectures, such as those based Magnetoresistive Random Access Memory (MRAM), forecast an even broader scenario for reconfiguration as they allow for dynamic modification in the wiring of elementary functional units, taking reconfiguration to the level of individual gates [11].

The main benefit associated to reconfigurable computing techniques is the possibility of reusing hardware and software components for multiple purposes, eliminating undesirable resource replications and allowing the system to cope with requirements that were not initially taken in consideration. Eliminating replicated components directly improves metrics such as size and power consumption, while increasing reusability and flexibility directly affects non-recurring engineering costs. Nonetheless, in order to achieve these benefits, it is not enough that designers base their projects on FPGAs that support partial reconfiguration: a complete infrastructure, at both sides software and hardware, is necessary.

A dynamic reconfiguration support system able to identify "which", "when" and "how" hardware and software components must be reconfigured in order to adapt a computing system to particular application demands is still far away. Indeed, some researchers predict that the computational cost of such infrastructure is more likely to overwhelm the benefits associated with the technology. This prediction is perhaps motivated by the history of dynamically reconfigurable operating systems of the 80s and 90s, like APERTOS [23] and ETHOS [19], whose reconfiguration infrastructure incurred in very high run-time overhead, preventing them from reaching the market despite all claimed advances. Notwithstanding, the actual scenario for reconfigurable computing is more hardware-bound then that of the 90s and is bringing about new opportunities that must be investigated from a more contemporary perspective.

In this essay, we discuss several issues concerning operating systems for reconfigurable computing, aiming at identifying concepts and mechanisms of traditional operating systems that can be reused or adapted to this novel context, and also proposing answers for some questions that cannot be satisfactorily answered based on current technology. First, the three main scenarios for reconfigurable computing are introduced: application-driven, instruction-level, and OS-supported. Subsequently, two reasonings are presented: how an operating system for reconfigurable computing could look like, and how a com-

ponent architecture for reconfigurable computing could be organized. Finally, the perspectives for the upcoming of a real operating system for reconfigurable computing are discussed.

## 2 Reconfigurable Computing

The inherent flexibility of reconfigurable computing has the potentiality to sustain a new generation of electronic devices that are able to self-modify themselves according to user's needs. A cell phone, for instance, could be reconfigured to perform the functions of a PDA, MP3 player, digital camera, navigation system, game pad, among others. Note that this is a scenario completely different from nowadays' multipurpose gadgets, which require specific circuitry to be integrated for each function they are supposed to perform. A reconfigurable cell phone would instead reprogram its hardware and software building blocks on-the-fly, just as the user selects a different function. If the gadget were to perform as an MP3 player, reprogramming would give rise to a specific audio decoder. While performing a game pad, reprogramming would probably build a kind of graphics processing unit (GPU). In this example, reconfiguration is directly controlled by applications, which explicitly initiate the reprogramming of building blocks as needed [1]. Therefore, we call this kind of reconfiguration *application-driven*.

Another reconfigurable computing scenario that has been consistently explored is the implicitly reconfiguration of hardware components without direct intervention by applications. This perspective of reconfigurable computing is based on the constant monitoring of hardware operational conditions, so as to initiate a pre-programmed reconfiguration whenever the associated conditions are observed. A good example of this kind of reconfiguration is a processor that is able to instantiate additional functional units as it detects overload situations. For instance, an application that runs into a heavy floating-point operations cycle would induce the processor to instantiate an additional floating-point unit (FPU) along with the structures needed to operate both units in parallel [20]. We will refer to this kind of reconfiguration as *instruction-level*.

A third scenario for reconfiguration lays between the previous two, at the *operating system* level. At the one hand, application-driven reconfigurations are only possible for very specific applications, designed and implemented for an specific platform, for it presupposes the application itself must know how to reconfigure the system. Instruction-level, on the other hand, usually limits reconfiguration to small functional units, missing the notion of macro-units such as complete subsystems. A properly designed operating system could bring some of the reconfiguration autonomy of instruction-level to the granularity of application-driven reconfiguration, thus yielding a more effective scenario for reconfigurable computing. The key issues in that design will be discussed next.

## 3 Operating Systems for Reconfigurable Computing

An operating system designed to support reconfigurable computing will have to handle the dynamic reconfiguration of software and hardware components on-demand. Reconfiguration itself can be driven either by functional or non-functional requirements of specific applications. For instance, if an application needs a certain functionality that is not yet available in the system, reconfiguration procedures will be carried out in order to add software and hardware building blocks to the system that together will deliver the required functionality. Independently of how long it takes to reconfigure the system, or how much energy the reconfiguration process consumes, reconfiguration driven by *functional requirements* must be done, otherwise the application will not be able to run.

Besides specific functional requirements, *non-functional requirements* can also drive reconfigurations and are often not restricted to individual applications. For instance, performance, energy consumption and real-time operation are requirements that must be met for the system as a whole and usually drive reconfigurations that affect several applications. In order to improve the performance of a system as a whole, some critical operations, or even complete algorithms, could be migrated to hardware, where parallelism can be more promptly explored. In order to save energy, some hardware components could be reconfigured to operate at lower clock rates, or perhaps could be replaced by more elementary ones. Real-time could be improved, for instance, by the replacement of sequential data structures in software, such as process lists, by associative memory in hardware, thus increasing determinism and reducing operational jitter.

In principle, the scenario for an operating system that aims at supporting reconfigurable computing might seem very different from that of ordinary operating systems, with functional and non-functional requirements changing along with the system execution and igniting complex reconfiguration procedures. Nonetheless, even ordinary operating systems usually implement sophisticated resource management strategies: CPU time, memory, disk, network and a variety of resources are consistently shared among processes and users. If we assume functional units, time, energy, and silicon area to be resources just like memory and disk area, then it should be possible to deploy many of the concepts, mechanisms, and algorithms of traditional operating systems in the realm of reconfigurable computing.

In addition to an interface, which would be accounted for the functional properties, components could also be tagged, during fabrication, with execution overhead, energy consumption and silicon area estimates. Combined, these informations could sustain some old, yet effective, resource management strategies. For instance, if the system becomes overloaded, reconfiguration could be started in order to replace active components by others with com-

patible interfaces, but better performance. If this leads the system to a higher energy consumption configuration, the previous configuration could be restored as soon as the load is set back to normality. Estimates defined at development-time, however, usually fail to remain meaningful as the system undergo severe operational conditions.

Event counters, currently used to estimate energy consumption [3], can be used to complement the static management model. Knowing how intensively a functional unit has been used in a given time interval can provide the resource manager with valuable information about possible reconfiguration targets. For instance, if an active component is being used only sporadically, and there is another functionally equivalent component that consumes less resources (e.g. silicon area), then it might be the case to replace it. The same counters could be used to trigger garbage collection operations, removing components that have not been used for a given time.

Although many technological difficulties are still to be overcame in order to make reconfigurable computing really attractive, from the operating system perspective, there is much knowhow that can be promptly reused. Indeed, we believe that the biggest challenge for an operating system that aims at supporting reconfigurable computing is to provide a dynamically repluggable software-hardware component infrastructure. On top of that infrastructure, a fully fledged operating system could be build mostly on currently available technology.

# 4 A Software-Hardware Architecture for Reconfigurable Computing

As stated in the previous section, an operating system for reconfigurable computing seems to depend mostly on an adequate software-hardware component infrastructure. At first sight, such an infrastructure could actually be built on available mechanisms: from the hardware side, strategies to map components to defined FPGA blocks, thus enabling them to be reconfigured without interfering in other components; from the software side, mechanisms such as dynamically loadable modules and shared libraries, which give similar answers. Nonetheless, reconfigurable computing presupposes not only loading and unloading components—what on its own can be a complex task when both hardware and software elements are involved, but also replacing them during system operation. Issues such as state preservation, compatibility of interfaces across software and hardware domains, limited resources for the infrastructure itself must be addressed while defining a *software-hardware architecture for reconfigurable computing*.

## 4.1 Components

The discussion about what components are—and what they are not—can be tracked to the early days of computing. It is not the goal of this essay to get into that more philosophical discussion, but to identify aspects of components that are relevant to the development of an operating system for reconfigurable computing.

Perhaps the most significant aspect of a reconfigurable computing component is the relationship to its clients; that is, its interface. If the component architecture enables the system to track the use of elementary hardware components up to the application program, then the universe for reconfiguration will be much broader then otherwise. By exporting high-level abstractions, the system can more easily keep track of "what" applications are doing at any time, thus enabling the reconfiguration of larger blocks. Consider a system configured to support a music playing application, for instance. Initially, the system might have been configured to include a custom decoder in hardware, although software and DSP versions of the same component were also available. Now suppose the application must go on-line through a radio network that demands complex signal filtering. It would be impossible for a system to replace the custom decoder by the DSP-based one—thus giving rise to the infrastructure necessary to do signal filtering—had it missed the information that the application was using a hardware component (i.e. the decoder) for an specific purpose (i.e. decode an audio stream). Nevertheless, reconfiguration would be straightforward if the system exported a `decoder` abstraction.

Similarly, replacing a software by hardware is usually far more effective in the context of high-level components with well-defined interfaces. If the application in the example above were executing the decoder software without assistance of the operating system, it would be extremely hard for the reconfiguration infrastructure to detect that the series of repetitive operations overloading the CPU were indeed related to an abstraction that is available in hardware, and thus trigger a reconfiguration. Furthermore, a redirection mechanism would have to be build around the application to intercept function calls and suppress them in detriment of hardware-equivalent operations.

Another important issue regarding software-hardware components concerns implementation languages and tools. In principle, describing software-hardware components in a single language such as SYSTEMC [13] would be ideal, since partitioning of software and hardware could be done, in theory, at any point. In practice, those languages are yet in their early days and manual intervention is often need in order to enable the translation of high-level constructs, thus supporting synthesis [17, 22]. An alternative is to have software components implemented in mature programming languages and hardware components implemented in mature hardware description languages. A meta description of such components can latter reconcile them in the sense of the component architecture: clients do not get to know whether to component is software, hardware, or a combination of both [2, 21].

### 4.2 Component Replugging

The primordial condition for component replugging are well-defined interfaces that are able to sustain some sort of plug/outlet relationship between components. The main difficulty of achieving that for reconfigurable computing is that sometimes the outlet is software and the plug is hardware (or vice-versa). From the software perspective, an interface is defined by a set of method signatures and, sometimes, behavioral constrains. From the hardware perspective, an interface is defined by a set of signals (or wires) and their operational conditions (including timing). Plugging such distinct artifacts together demands some sort of "adapter".

Hardware mediators, defined in the context of Application-Driven System Design [8], solve the problem of hardware-software plugging by deploying static metaprogramming techniques to wrap the hardware with a software compatible interface, thus giving origin to a *virtual software/hardware interface*. This interface has already been consistently discussed by the authors [14].

Another major issue pertaining the replugging of components is state maintenance. When a component is replaced, its successor must have its state initialized accordingly. Saving the state of the old component simply by copying the memory region that contain its attributes (software) or dumping its registers (hardware) will probably not be enough to support the initialization of the new one, since the internal representation of information is particular to each component. State saving and restoring must some how be done through the interface of components, which are the only sustainable compromise among them. An interface method such as `shutdown_and_give_me_your_state` is a straightforward beginning, but the returned state will sometimes have to be reinterpreted before being used to initialize the new component. This reinterpretation can be carried out by components themselves of by some sort of "component broker". Anyway, if components are not fruit of a consistent hardware/software co-design methodology, state incompatibility might render reconfiguration impractical.

Besides proper interfaces, adapters, and mechanisms to save and restore state, component replugging is only feasible if the reconfiguration infrastructure is able to track logical components down to reprogrammable hardware blocks. If the silicon area formerly used by a component that has been shutdown is to be used by a new one, then precise information about that area (i.e. which cells/slices were occupied by the component) must be kept at runtime. Furthermore, the hardware fabric must be so that reprogramming some blocks does not disturb the operation of others. This issue has been extensively addressed by the hardware community [6, 12, 10, 16, 4] and could be regarded by the operating system mostly like a resource allocation problem similar to keeping track of allocated blocks in a disk or frames in a memory, just that some blocks hare particular characteristics (e.g. *block RAM*).

A component architecture build around these premises should be able to support the tree basic system reconfiguration operations: extension, contraction, and replacement.

#### 4.2.1 Extension

System extension happens when a new component must be added to it in order to fulfill the current set of requirements defined by applications, be it functional or not. In essence, extension can be done based on *demand loading* strategies adopted in traditional operating systems to dynamically load large libraries: objects are first loaded when they get used. Typical strategies consist in registering component interfaces within the system, but leaving their implementations unbound. A first attempt to access the component causes a fault that triggers the loading and binding of the missing component []. A similar strategy could be used to dynamically load software and hardware components into the system.

#### 4.2.2 Contraction

System contraction happens when a component that is no longer in use gets removed in order to free resources. The main motivation to system contraction from the perspective of software is to increase the amount of available resources for eventual extensions. From a more generic perspective, however, non-renewable resources must also be taken into consideration. For instance, keeping an unused component operational will very likely consume energy that cannot be recovered. In this sense, tracking information about which components are actually in use becomes crucial for a reconfigurable system.

Contraction can be directly started by applications when they invoke `destructor` or `finalize` methods. Additionally, event counters traditionally used for energy management [3] can be deployed to identify components that have not been used for "long" periods of time. Defining "long" becomes the issue here, for there is no guarantee that a component that has not been used for some time will never be used again. Contracting the system to short latter expand it back to the same configuration is certainly something to be avoided. Fortunately, once more there is extensive research in the context of cache and main memory management to be reused. The concept of *workset* and the related approximation algorithms are good candidates to drive the *garbage collection* of system components [18].

#### 4.2.3 Replacement

Component replacement becomes straightforward if the architecture presents an appropriate hardware/ software interface and the reconfiguration infrastructure supports state saving, area mapping, extension, and contraction mechanisms. Perhaps the most obscure issue regarding component replacement today is the management of the

knowledge needed to decide which components must be replaced, which components must replace them, and when replacing must take place. Similar issues pervade the reflective systems literature and can be regarded as a research field on its own.

### 4.3 Component Repository

Dynamic component replugging presuppose the existence of previously validated components. At load-time, such components usually take the form of binary files that, along with formally described interfaces, build a repository. In non-embedded systems, component repositories are normally stored in a file system or data bank. This, however, is mostly inadequate for embedded systems, since most of them do not feature a mass storage unit. Pre-loading all component into main memory is usually not possible too, for this is often a limited resource.

For stand-alone embedded systems, devices such as flash memory are currently the most viable alternative to store the component repository. During the building of the system, all components that are plausible to be used are loaded into the flash, along with the operating system *nucleus*[1] that implements the basic reconfiguration infrastructure.

For connected systems, downloading components from a remote repository might also be an alternative. Depending on the coupling level, even some of the reconfiguration infrastructure can be moved out of the embedded system into a server. This could, for instance, be the case for a machine or an automobile. Reconfiguration from a remote repository requires additional communication mechanisms and main memory (to store components during download), but brings along a powerful *upgrade* mechanism almost for free: by replacing components in the server's repository and triggering a reconfiguration, the system gets upgraded with having to be stopped.

## 5 Perspectives

The recent achievements of the hardware community with regard to reconfigurable computing are very promising. However, the same is not yet true for the software by-side . At the one hand, it is natural that researchers in the field were first challenged by the hardware aspects of reconfigurable computing, but, at the other hand, the absence of specific operating systems and even of adequate hardware/software component architectures might be a signal that the software aspects of reconfigurable computing are being neglected or that the scientific community intends, one more time, to deploy the "commodity" principle and adopt an all-purpose operating system for the first generations of reconfigurable computers.

[1]The term "kernel" was avoided here not to induce the reader to assume that a reconfigurable computing infrastructure will be similar in size to a traditional operating system kernel. Based on previous experiments [14], we believe that it can be rather small.

We believe that the consolidation of reconfigurable computing depends, among other things, of a proper component architecture to sustain the design of an reconfigurable system as a whole. The Polymorphous Computing Architecture project defines a component architecture [15], but misses a hardware/software co-design strategy. Moreover, many concepts seems to be closer to nowadays web-services than to real embedded systems.

Nonetheless, we believe that a component architecture for reconfigurable computing can be built mostly on existing concepts. The most significant of them have been identified in this essay and pertain to an infrastructure for component replugging, an strategy for resource management, and a component architecture. We believe that the combination of these elements with the existing hardware-based reconfiguration engines could yield a more application-oriented perspective for reconfigurable computing systems.

## References


[1] A. Agarwal. Raw computation. *Scientific American*, 281(2):44–47, Aug. 1999.

[2] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araújo, C. Araújo, and E. Barros. The archc architecture description language. *International Journal of Parallel Programming*, 33(5):453–484, Oct. 2005.

[3] F. Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 37–42, New York, NY, USA, 2000. ACM Press.

[4] O. Colavin and D. Rizzo. A scalable wide-issue clustered vliw with a reconfigurable interconnect. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 148–158, New York, NY, USA, 2003. ACM Press.

[5] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[6] S. Dutt, V. Shanmugavel, and S. Trimberger. Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 173–177, Piscataway, NJ, USA, 1999. IEEE Press.

[7] M. Forum. Introduction to morphware: Software architecture for polymorphous computing architectures. Technical report, Georgia Institute of Technology and Space and Naval Warfare Systems Center San Diego, Feb. 2004.

[8] A. A. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[9] R. Hartenstein. The digital divide of computing. In *Proceedings of the 1st Conference on Computing Frontier*, pages 357–362, Ischia, Italy, Apr. 2004.

[10] M. Huebner, T. Becker, and J. Becker. Real-time lut-based network topologies for dynamic and partial fpga self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, New York, NY, USA, 2004. ACM Press.

[11] R. Kock. Morphware. *Scientific American Magazine*, 293(2):44–51, Aug 2005.

[12] J. Noguera and R. M. Badia. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *Trans. on Embedded Computing Sys.*, 3(2):385–406, 2004.

[13] P. R. Panda. Systemc: a modeling platform supporting multiple design abstractions. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, New York, NY, USA, 2001. ACM Press.

[14] F. V. Polpeta and A. A. Fröhlich. On the Automatic Generation of SoC-based Embedded Systems. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, Sept. 2005.

[15] M. A. Richards, D. P. Campbell, and K. M. Mackenzie. The morphware stable interface: a software framework for polymorphous computing architectures. In *Proceedings of the 28th Annual GOMACTech Conference*, Tampa, U.S.A., Mar. 2003.

[16] I. Robertson and J. Irvine. A design flow for partially reconfigurable hardware. *Trans. on Embedded Computing Sys.*, 3(2):257–283, 2004.

[17] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of systemc specifications. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 238–243, Piscataway, NJ, USA, 2004. IEEE Press.

[18] A. Silberschatz, P. Galvin, and J. Peterson. *Operating Systems Concepts*. John Wiley and Sons, fifth edition, 1998.

[19] C. A. Szyperski. *Insight Ethos: On Object Orientation in Operating Systems*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1992.

[20] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[21] G. F. Tondello and A. A. Fröhlich. On the Automatic Configuration of Application-Oriented Operating Systems. In *3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 2005. to appear.

[22] S. Virtanen, D. Truscan, and J. Lilius. Systemc based object oriented system design. In *Fourth International Forum on Design Languages (FDL'01)*, Lyon, France, Sep 2001. ECSI.

[23] Y. Yokote. The apertos reflective operating system: the concept and its implementation. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 414–434, New York, NY, USA, 1992. ACM Press.