Model-driven Toolset for Embedded Reconfigurable Cores: Flexible Prototyping and Software-like Debugging

Loïc Lagadec\textsuperscript{a,b}, Ciprian Teodorov\textsuperscript{a,b}, Jean-Christophe Le Lann\textsuperscript{a,b}, Damien Picard, Erwan Fabiani\textsuperscript{a,c}

\textsuperscript{a}Lab-STICC MOCS, CNRS 6285
\textsuperscript{b}ENSTA-Bretagne, France.
\textsuperscript{c}Université de Bretagne Occidentale, France.

Abstract

Improvements in system cost, size, performance, power dissipation, and design turnaround time are the key benefits offered by System-on-Chip designs. However, they come at the cost of an increased complexity and long development cycles. Integrating reconfigurable cores offers a way to increase their flexibility and lifespan. However, the integration of embedded reconfigurable units poses a number of unique challenges in terms of design-space exploration and system exploitation. Over the last few years, model-driven engineering has become one of the most promising methodologies for tackling such challenging software problems.

This paper presents Biniou, a model-driven toolset for embedded reconfigurable core modeling. Biniou is a major step ahead of the Madeo framework that was one of the rare non-commercial environments targeting reconfigurable design automation. In Biniou, the design space is broadened with (re-)configuration modeling aspects, and the exploitation tools are enhanced through the use of multi-level simulation and high-level debugging.

These advancements are illustrated through a case-study focused on the design-space exploration of a coarse-grained reconfigurable architecture and through an examination of the integration of the debug-specific features into the framework. The main benefits of the presented toolset are: efficient domain-space exploration (validation), software design-kit generation (usability), software-like debug facilities (verification).

Keywords: Agile programming, Software Engineering, Model-driven, Debugging, Reconfigurable computing, System-on-Chip

1. Introduction

The tremendous evolution pace of the semiconductor industry, has enabled unprecedented advances in the ways we see computer system design. Today’s integrated circuits (IC) are more complex and heterogenous than ever. Being composed of several processors (\(\mu P\)), digital signal processors (DSP), communication networks (NoC), and complex memory hierarchies, they have attained full system functionality into a single chip, and are referred to as System-on-Chip (SoC). These SoCs have the potential to offer several benefits, including improvements in system cost, size, performance, power dissipation, and design turnaround time. Unfortunately, the SoC design process, which schematically consists of successive refinements of an abstract specification towards the physical realization, is becoming an increasingly difficult task.

In the early 90’s the term hardware/software co-design appeared to describe a confluence of problems in IC system development. The prefix \(co\) is used to denote a joint or partnered development – as opposed to the coincidental development at the same time – of both hardware and software. The key idea behind hardware/software co-design is to find the right trade-off between speed of hardware execution and generality of software, while considering the costs incurred. The main problem in hardware/software co-design is how to design an embedded system that contains both hardware execution (through application-specific ICs\textsuperscript{1})

\textsuperscript{1}Application-Specific ICs are commonly named ASICs

Preprint submitted to Journal of Computer Programming  
June 27, 2014
and software execution support (through µP). A critical decision that has a wide effect on overall system cost is how to partition the system into its hardware and software components. A mistake made in this decision can add significant delay and cost to the design process, since correction in this context implies reworking the entire design. The longer this irrevocable decision can be delayed, the better the chance to keep overall system costs to a minimum.

Historically, reconfigurable devices, such as Field-Programmable Gate Arrays (FPGA), provided a solution to this conundrum by offering support for late hardware customization, which – much like software late binding – enables early availability, reuse and tailoring. Compared to ASIC, the agility of reconfigurable architectures comes from the ability to re-allocate resources (potentially in the field, partially, and on the fly) to form a new circuit. This favors fast prototyping and early circuit implementation, sometimes even prior to full specification availability.

Today, with the emergence of the fabless business model, new competitors have entered the race of reconfigurable platforms[1, 2, 3]. The field of reconfigurable computing has morphed, and besides mainstream FPGA vendors such as Xilinx and Altera, the fabless solution providers offer more specialized reconfigurable devices, ranging from coarse-grained cores for DSP[4], to embeddable units[5].

Embedding reconfigurable cores into an SoC offers a tradeoff between the area overhead and the flexibility of the system that must be carefully considered. Conceptually, the area overhead can be estimated throughout the design cycle by physical synthesis tools (designing the reconfigurable device) while the flexibility is scored using applicative synthesis tools (performing the resource allocation in order to map a portion of the application to the reconfigurable unit).

Unfortunately, traditional solutions require a large amount of manual tuning during the physical synthesis and remain bounded to specific tools during applicative synthesis, reducing the ability to explore new architectural options. The loss of effectiveness of methods and tools to address real hardware over the increasing hardware complexity is referred to as the productivity gap. This grows with every new technological evolution and brings new challenges to the designers [6]. In the scope of SoC design, this trend concerns both architecture and software design. This prohibits short development cycles, and renders the design space exploration (DSE) unaffordable.

To address these issues, in this paper, we introduce an open model-driven toolset for the design of embedded reconfigurable units including generation of both hardware prototypes and their specific SDKs (exploitation tools). We rely on the Madeo framework mixed ADL which provides architectural DSE and retargetable place&route tools [7]. In addition to architectural DSE features (such as sizing LUTs, arithmetic operators, etc.) the Biniou toolset also enables DSE for partial and multi-context dynamic reconfiguration modes, thus widening the design space. Moreover, to facilitate agile development, and to ease exploitation our solution tightly integrates with an innovative software-like debugging infrastructure, named RedPill, which offers high-level signal traceability features and control over the hardware execution throughout all circuit synthesis stages – from the application to the bitstream, and more.

The creation of an integrated toolset targeting design-space exploration of embedded reconfigurable units, such as Biniou, is a complex undertaking requiring flexible and extensible solutions for complex problem-spaces. The solution presented in this paper relies at its core on many years of proactive research and development in the FPGA design-automation field, notably around the Madeo infrastructure[7]. However, addressing the highly-dynamic and challenging field of embedded reconfigurable units, in the context of SoC design, opens the door to a whole new set of problems, notably in terms of tool adaptability, integration, and synergy. Besides providing an answer to these technical and methodological issues, the main contributions of this work are:

- **Configuration plane modeling.** In the context of design-space exploration of embedded reconfigurable architectures, besides computational architecture modeling, the design of the configuration infrastructure plays a very important role in the overall performance of the architecture. The configuration

---

2. Fabless manufacturing is the design and sale of hardware devices and semiconductor chips while outsourcing the fabrication to a specialized manufacturer, called a foundry.

3. In this study we will use SDK, or exploitation flow (tools) to refer to the set of tools aimed at mapping a given application on a target reconfigurable architecture.
infrastructure, composed of memory elements, is in charge of allocating computational resources according to the circuit being created. Biniou adds the possibility to specify the configuration infrastructure through a domain-specific language (DSL) that is orthogonally composed with the computation plane specification (in Madeo ADL). This approach enlarges the design-space completing the view over the reconfigurable core. Moreover, it is the core enabler for the generation of a fully functional prototype.

- **Fully-featured prototype generation.** Reducing time-to-market is one main lever for survival in the highly-dynamic SoC context. Agile-development strategies propose a development process relying on the iterative refinement of a functional system until it fully realizes the specifications. In Biniou, we enable this approach by generating a fully-functional prototype of the reconfigurable fabric. The generation relies on the joint Madeo-ADL/Configuration-DSL specifications to produce a synthetizable VHDL realization. Besides the reconfigurable core, the prototype embeds a specialized micro-coded configuration controller. This approach opens the way for architectural virtualization over commercial-off-the-shelf FPGAs, as is shown in Section 5.

- **Object-oriented view of the prototype.** Biniou relies on the Madeo infrastructure[7] for the specialization of the exploitation tools for the prototype, and it integrates with a multi-level simulation framework[8] as well as with external tools, such as Modelsim[9]. To offer a seamless experience the application, as well as the reconfigurable prototype, are reified in the environment and provide an object-oriented API. This feature paves the way for high-level circuit debugging, and offers the possibility to mix hardware and software objects thus creating an effective SoC prototyping environment, where the OS assures the migration of applicative processes over the software-hardware sub-systems.

- **High-level debug: observability, traceability, controllability.** Biniou offers a parameterized observability by pulling the internal signals out to the hardware interface. During execution, the traceability support enables to connect the captured signals back to the original high-level code variables. To prevent performance degradation the traceability is only activated on-demand. To control the execution, assertions are used to isolate significant time-windows that require deeper analysis, moreover specialized debug controllers are inserted to offer start, stop, step-by-step, resume operations. Moreover, the integration of these features with the architecture at the architecture model-level and the prototype-level offers a unique hardware debugging experience.

This paper is aimed at a compound audience of reconfigurable platform designers and software experts. While this work pushes the state-of-the-art of software tools in the embedded reconfigurable field, and presents the reconfigurable platform designers with practical tools, throughout the presentation the focus is steered towards the similarities and differences between the software and hardware worlds. For the software experts the interest is primary methodological – centered on model-driven and agile development processes – but with a pedagogical note introducing the central software aspects of reconfigurable SoC platforms, and the associated challenges.

**Paper organization.** Section 2 presents the state of the art of modeling tools for reconfigurable architectures, and reviews the main hardware debugging techniques. The overall architecture of the Binion toolset is described in Section 3 with a special focus on the architectural prototyping aspects. Section 4 details the RedPill hardware debugging infrastructure. To better illustrate the concepts presented, Section 5 briefly explores the design of a coarse-grain reconfigurable core, and overviews some practical debug situations. Finally, Section 6 concludes this paper discussing future research directions.

### 2. Related Works

Traditionally, an HW/SW design flow for SoC starts with an application to be further profiled and divided into hardware units in charge of compute intensive tasks and programmable units. Fig. 1 illustrates this point, with a specific focus set on the HW generation. This offers a cost killing opportunity through reusing
components, however it does not support refactoring the SoC’s structure nor the programming tools which remain library-based [10]. Moreover, direct reuse of components typically does not meet the performance requirements for SoC, and the fine tuning of units remains mandatory.

On the contrary model-driven environments offer an automatic way to generate hardware prototypes and their SDKs. For SoC, this mainly addresses the generation of an Application-Specific Instruction Set Processor (ASIP)[11, 12, 13]. Moreover this approach offers validation opportunity through domain-space exploration (DSE). Automating the generation of the full package favors short iterative cycles, while benefitting from simulation feedback to measure the impact of changes over high-level specifications.

2.1. Architecture Description Languages

Modeling hardware relies on specialized DSLs, that are instantiated using textual representations referred to as Architectural Design Language (ADL). For ASIP, three types of ADL are taken into consideration depending on the designer’s intent: structural, behavioral or mixed [14]. Some examples include: a) The MIMOLA environment [15] is oriented towards the micro-coded programmable architectures design, at an RTL level. A synthesizer, a micro-code generator, a simulator, etc. manipulate a common DSL. b) The nML language [16] describes the Instruction Set Architecture (ISA) of processors by capturing their instruction semantics. Compared to MIMOLA, few structural details are considered, and the description is very close to an ISA reference manual. c) LISA[17] focuses on generation of both cycle-accurate simulators and retargetable C-compilers. LISA appears as a tradeoff between a structural approach (MIMOLA) and a behavioral approach (nML), as it supports both pipelining and behavior factorization. LISA is part of the commercial Synopsys toolset[18]. Both the interest from industrial partners and the fact that commercial products are now available show the maturity of this approach [18, 19].

This modeling approach can be transposed to reconfigurable unit design, while considering that the main difference simply lies in the nature of architectural descriptions and generated tools. One of the most significant experiment is centered on the VPR DSL. This DSL enables FPGA modeling, with the GILES tool [20], and it is able to generate an FPGA layout [21]. In addition, the prototype can be programmed through the VPR place&route flow. A more specific approach is ADRES which is based on a parametric architectural template structured around a VLIW processor coupled to a coarse-grained reconfigurable matrix [22]. It takes into account the generation of the architecture to be synthesized and of its specific SDK according to DSE results. A similar approach based on the LISA ADL is proposed in [23].

Despite their completeness these methods make strong assumptions concerning the architectural characteristics of the reconfigurable units. Furthermore, validation forgets the dynamic-reconfiguration feature. These approaches do not address DSE for dynamic reconfiguration modes which offer interesting resource/performance trade-offs. Consequently, the design space is restricted to a pre-defined set of architectures hampering a fine-tuning to meet application needs.

Other approaches enlarge the modeling possibilities with no restriction on the architecture. For example, Mozaic platform enables dynamic reconfigurable architectures prototyping with no restriction on the resource characteristics [24]. It takes into account hardware generation of the architecture and partial/multi-context dynamic reconfiguration modes.

![Figure 1: Overview of a co-designed HW-SW application.](image-url)
2.2. Hardware Debugging Techniques: Comparative Review

Nowadays, most common methods for hardware design verification are based on software or hardware simulation with RTL (Register Transfer Level) as the highest abstraction level, however RTL is quite close to assembly code.

Software simulation is a widely used debugging method since it is cost-affordable and provides complete controllability and observability. However, it suffers from performance drawbacks when simulating large and complex designs.

To overcome the speed problem, verification can be carried out directly within the hardware (built-in self test). However, despite being efficient for error-detection, embedding tests does not fully enable debugging activity. One challenge is, having detected a deviation, to restore controllability on demand, to let the designer control the execution flow (step-by-step, continue, etc.), to allow running multiple scenarios, and ultimately to support design evolution.

Abstraction. When addressing software debugging, four cases must be considered. The first one (ordered by increasing complexity) is debugging sequential code. This draws on tools and techniques developed over many years, appears as mature and can support advanced features such as on-the-fly code replacement, depending on the debugging environment (e.g. Smalltalk debugger). The second case is parallel software debugging. This is more complex as several threads run in parallel, but the scheduling directly relies on mechanisms to prevent mutual accesses to shared resources (e.g. memory). Complexity comes from high amount of parallel threads as well as from potential race conditions. Some tools, however, exist that support parallel debugging [25] but state-of-the-art research proposals [26] remain far ahead of industrial practices. The third case is concurrent processes debugging, which can be seen as a virtualization of the previous case. In that case, several processes run concurrently on a single processor raising the need for a real scheduling. Scheduling means electing a process, out of “ready” processes, to be moved into the “running” state. Once done, only this particular process moves on, resulting in asynchronism. The scheduler - provided as an OS service - is totally opaque and isolated from the programming environment, hence debugging has no visibility of it and reproducibility is hard to ensure. Then, debugging distributed applications faces a parallel/concurrent composite situation with several interacting schedulers. Gaining a consistent view of the execution is challenging, and a global virtual time must be computed in order to serialize events accordingly.

Hardware execution relates to cases 2 and 4, and 3 in case of dynamic partial reconfiguration. However, while software variables are clearly identified, their hardware counterparts are 1-bit signals, spread all over the design. Collecting information seems like an advanced surgery. Also the amount of data to be considered is much higher compared to software execution. As an illustration, while time-sharing executions relies on saving/restoring process control blocks, DPR cannot resume an interrupted task because its context would be far too wide to restore. Instead, DPR swaps tasks, after their completion occurred. As a consequence, debugging FPGA applications is very challenging.

Nevertheless, the FPGA debugging exhibits a real potential, because hardware executions are often deterministic. Besides, not only the scheduler is relatively simple, but is part of the design. Then, while introspection is bounded to the application in a software scope, hardware execution offers additional information, valuable for debug. Last, for circuits designed using a high-level-synthesis flow, the scheduler appears as an application dependent finite state machine, with a known behavior. This automata can be extended on demand, in order to support extra controllability (e.g. step-by-step execution).

As a summary, while software debuggers are the cornerstone of any agile software development, hardware debugging remains far behind due to the tools low maturity. Even though, these three points (simplicity, observability, semantics) offer a real opportunity for advanced debugging of FPGA application.

Probes. Be it software or hardware oriented, efficient debugging relies on probes to provide a way to check the state of (observe) the system at a specific point.

When validating software, the probes do not change the source code design, but will affect the timing of the program execution. Similarly, using an electronic probe does not change the design of an electronic circuit but, when used, it may slightly change the circuit’s characteristics. Probes can be basic (and log a variable) or computed (and log an assertion).
Supporting assertions is the key to the scenario-based specification that itself drives the design and verification process [27]. Using a natural language description of a set of design requirements may lead to problematic situations in which ensuring that all functional aspects of the specification have been adequately verified is almost impossible. The property specification language (PSL) [28] was developed to address this shortcoming. It gives the design architect a standard means of specifying design properties using a concise syntax with clearly defined formal semantics (including temporal behavior). Many works have been reported on automating the verification of PSL-based specifications [29, 30, 31, 32, 33, 34], though only [35] provided a temporal verification scheme of a hardware accelerator.

Depending on the action that may be triggered by the probe’s activation, either breakpoints or watchpoints are considered: a watchpoint logs status information without disturbing the execution, while a breakpoint interrupts processing.

During software execution, a breakpoint immediately opens the system debugger when triggered. It shows the last few functions executed with the top-function in the stack being the function containing the breakpoint. The debugging tool allows extensive exploration of the history of execution-flow, code or variable’s value changes on the fly, and program execution control. After a breakpoint has been triggered the function can be continued, executed step-by-step with or without diving into function calls.

To preserve its speed-advantage, hardware debugging cannot offer full observability, nor execution-stack-like traveling. Observability means more than simply obtaining access to the current state of internal signals. This would require logging mechanisms that are not scalable and would slow-down the execution. As a consequence, the hardware designer has almost no information regarding the past states of the circuit. This drastically hinders the ease of analysis, since to understand the circuit’s current state the designer cannot rely on any of these tools (no observability due to lack of scalable logging, no stack-traveling because of the spatial execution).

**Hardware observability: to hell and back.** When debugging circuits, designers can use embedded logic analyzers [36] or connect some IOs to a mixed-signal oscilloscope (MSO). The logic analyzers offer an insight into the behavior of the reconfigurable circuit (eg. FPGA standing for Field Programmable Logic Arrays), in the context of the surrounding system. This goes through the connection of some internal signals to physical pins, only a small number of which are commonly available.

Agilent Technologies however provides a software solution that overcomes some of these limitations by offering dynamic probes [37].

Another solution lies in bitstream instrumentation [38][39]. Xilinx ATC2 cores [40] can be added either during the design stage or within a post-synthesized netlist (similar to byte-code transformation [41]), to offer access to any internal signal and communication with external MSO. From a Smalltalk point of view, this is similar to requesting the designer to load a parcel in order to support Transcript show: operations.

ChipScope [42] is another solution to reflect activity after signal capture. In addition, some FPGAs offer read-back capability [43, 44, 45], and internal signals can be retraced. In a sense, Chipscope is more or less a Smalltalk inspector, but that would be available, for example, only on a particular virtual machine implementation (and would be virtually impossible to port it to a new one). It brings nice features, at the expense of linking the designs to one platform, with all the risks that ensue.

These few solutions brought pieces of the observability designers missed for years by reflecting the FPGA internal state. Although, these functionalities remain available at a very low-level, compared to functional specification, and debugging requires more than just observability. In RedPill we address this issue by using on-demand traceability which links the two views enabling the recomposition of, for example, an 32 bit integer from its 32 logic signals that might be routed all over. Besides, some timing windows are critical since this is where the critical operations happen (inter-process synchronization, looping, conditions, etc.) while digging into some others should be avoided to prevent over consumption of logging resources. Controllability enables the discretization of time and isolation of hot-time windows, hence all the designer needs to do is to focus on these windows. The observability policy can then be tuned up depending on the window tag so that only hot-spots are considered with care.
2.3. Summary of contributions

This article presents a holistic approach ranging from the fabrics design, simulation and emulation as a virtual FPGA, to providing an object encapsulation for circuits under use. By design, we refer to the ability to model the fabric through an ADL. This is comparable to the LISA approach [46] but applies to a different architectural domain, namely FPGA. From this model, we produce both a software programming environment (similar to [47]) and an implementable description of the fabric (as [20] does). The main difference lies in the wider range of fabrics we can model, as illustrated by Figure 4b. Also we incorporate a "multi context" feature following the DPGA scheme introduced in [48]. Besides, our fabrics support dynamic partial reconfiguration (as described in [49]) but exhibit much more flexibility in term of reconfigurable zones (Figure 3).

By encapsulation, we refer to a language construct that facilitates the bundling of data (high-level specification, netlist, circuit’s current state) with the methods operating on that data. These operations are: High level synthesis (similar to [50]), probe insertion (similar to simplified PSL [28]), circuit swapping, access to variables (but in a portable way compared to [42]), GUI update (Figure 7), etc. This mechanism offers a basic support for debugging the application, but also favors a system usage of the fabric.

3. Biniou: MDE-based Toolset for Embedded Reconfigurable Units

In this section we present a new toolset supporting the hardware generation of embedded reconfigurable unit prototypes along with the associated exploitation tools. This toolset, named Biniou, provides a complete prototyping environment starting from high-level DSE down to a hardware implementation for deeper analysis. It relies on high-level specifications of the targeted architecture to derive synthesizable prototypes and to specialize the compilation/synthesis flow for them. This approach enables the designer to perform early exploration of architectural concepts guided by a set of metrics. Moreover, the automatic specialization of the exploitation tools provides the means for experimentation with real-world applications.

![Figure 2: Global flow supporting generation of a complete environment for reconfigurable unit prototyping.](image-url)
Fig. 2 illustrates the global flow of our toolset decomposed around two main axes: architecture prototyping and exploitation tools. These two axes are complementary and interdependent, and ideally should be present in any design-space exploration tool-flow. The architecture prototyping enables the specification and the refinement of the reconfigurable unit according to architectural specific evaluations (such as area, power consumption, etc). The exploitation flow complements the designer toolbox with the possibility to use the designed prototype and eventually tune it towards the specific needs of the targeted application domain. Besides these two axes, in Fig. 2, the Biniou toolset offers support for two important transversal concerns, namely simulation and debug. On the exploitation tools axis these correspond to the possibility to simulate and debug the user application at all the synthesis stages (from C code to synthesized netlist). On the prototype side, simulation corresponds either to cycle-accurate architectural simulation or to specialized simulation/debug facilities added to the prototype to facilitate the debugging of the mapped applications.

The following subsections will present these core components of the Biniou toolset. The user should note that the debugging facilities of Biniou will be thoroughly discussed in Section 4.

3.1. Prototyping Embedded Reconfigurable Units

In Biniou, the architecture prototyping is decomposed into the computational-resources specification and the configuration-plane specification to perform orthogonal DSE for each concern. The resource model corresponds to the routing and computing resources which are sized in term of granularity according to exploration results. The configuration model gives the structure of the configuration plane as a set of regions. Every region is independently reconfigurable and supports multiple contexts. From a practical point of view, both these models contribute to deriving the back-end applicative tools including a synthesizer that produces netlists, a placer-router in charge of resource allocation and a bitstream model with a generator that outputs the configuration file.

Prototype implementation is based on the generation of a VHDL design taken as input of cycle-accurate simulators, such as Modelsim, or synthesized on state-of-art FPGA. The configuration-plane of the hardware prototype is connected to a specialized configuration controller that supports both partial and multi-context reconfiguration modes.

3.1.1. Reconfigurable Architecture Modeling

In Biniou, the reconfigurable architectures are specified using the legacy Madeo[51] ADL, a functional-style language specifically designed for the hierarchical structural specification of reconfigurable architectures with either homogeneous or heterogeneous tile structure. The primitives of this language can be classified into the following categories: logic resources, memory resources, I/O and routing resources, composites, and domains.

Logic resources model computational units such as look-up tables (LUT), operators, or specialized functional units. Memory resources model registers, latches, memory-banks, etc. The I/O resources represent either the pins of the composed elements or the IO pads on the periphery of the chip. The routing resources correspond to the connection blocks, the switch blocks, or other interconnect resources such as buffers and programmable interconnection points (PiP). All these primitive architectural components are assembled into composite elements, that are replicated in homogeneous domains.

When addressing reconfigurable architectures, a common baseline is provided by VPR generic model and is referred to as FPGA[52]. Using our ADL two types of tiles are considered: configurable logic blocks (CLBs) and input-output blocks (IOBs). A CLB will be represented as a composite element with: two routing channels, a switch and four logical elements, each of which embeds a 4-LUT, multiplexers, a register and a tri-state buffer output control. The granularity of routing channels is one-bit width.

Listing 1 presents a snippet of the description of a composite element showing a coarse-grain configurable operator and one register. The logic resource introduced by the keyword FUNCTION (line 2) represents a coarse-grain operator with two 8-bit inputs (fin0, fin1) one 8-bit output configurable to implement one of the operators specified on line 7. The register (line 18) is specified with the register keyword and represents an 8-bit register that is functionally represented in the model.
The Madeo ADL serves a simplified concrete syntax for the instantiation of the architectural model. It should be noted that the framework is completely open and other architecture description languages can be supported. Moreover, for specific cases, where these functional descriptions cannot easily represent the required structure, Smalltalk code can be evaluated in the context of the ADL description.

The last aspect worth mentioning is the possibility to associate a representation to each model element (see Listing 1 lines 8 and 20). This representation is used to create an architecture-specific circuit editor that is tightly integrated with the prototype exploitation tools.

3.1.2. (Re-)Configuration Infrastructure Modeling

In the context of design-space exploration of embedded reconfigurable architectures, besides the computational resource design (described in Sec. 3.1.1), the design of the configuration infrastructure plays a very important role in the overall performance of the architecture. The configuration architecture can be seen as a layer of memory elements tightly connected with the computational plane. This configuration layer stores an application specific pattern of binary words (bitstream) that sets each element (logic functions, registers, configurable connection points, etc) in the state required for implementing the given application.

The Biniou toolset integrates the Drage extensions of the Madeo architectural model which couples the computational-resource specification to the specification of the configuration infrastructure, thus enabling orthogonal design-space exploration of these two concerns and fully functional prototype generation.

In Drage, the configuration infrastructure (configuration plane) is specified as a disjoint set of regions, each region being associated with a number of tiles of the computational plane. The role of these regions is to group together the configuration memories of the associated tiles, and to provide the support for their independent configuration with respect to the other tiles. Each configuration region can hold a number of configuration contexts, thus enabling multi-context reconfiguration. The number of regions and the number of contexts per region shape the design space trading off the partial and multi-context reconfiguration features of the designed reconfigurable unit.

The configuration memory architecture can implement a double-buffering policy to enable dynamic reconfiguration without disrupting the execution. Each region stores a current execution context which is loaded in parallel from its context set. The computational layer elements are connected to their corresponding layer through a configuration bus.

Another important parameter of the configuration plane is the configuration throughput (in bits/cycle), which directly influences the time (in number of cycles) to load a configuration. For each configuration region
(a) The architecture is exposed to the designer (tabular view on the right, list of elements on the left) who specifies the regions and their contexts. The list in the middle shows the ids of the declared regions. The selected region is highlighted in the tabular view.

(b) The configuration manager executes a micro-program specifying a configuration schedule (right branch in Fig. 2).

Figure 3: The Front-End GUI allows defining configuration domains and generates the architecture along with a configuration manager.

the latency is the ratio between the size of its configuration bus and the size of the region’s input port. This parameter trades-off the configuration speed (in terms of cycles) with the number of direct connections (and ultimately the area) between each region and the configuration controller.

Fig. 3a shows the interface through which the designer describes the configuration plane in the Biniou toolset. It should be noted that the user can specify the regions either using Smalltalk code (suitable for handling complex specifications) or using a simplified XML-based DSL (see Listing 2).

Listing 2: Example of configuration plane specification using an XML-based DSL, specifying the two regions presented in Fig. 3a

To alleviate the complexity of configuration management in the context of partial and/or multi-context configuration Biniou introduces a programmable configuration controller (see Fig. 3b) which is automatically synthesized according to the configuration plane specifications. This controller executes a micro-program compiled from the scheduling results of the user application. Every micro-instruction encodes a region id, a context id, as well as information concerning the bitstream to configure. The controller, presented in Fig. 3b handles 3 types of instructions: a) activate that configures the current context of a region by activating the configEn port; b) load that sends a configuration in specific context of a given region; c) delay that enables the synchronization of the embedded reconfiguration architecture with the host system. Thus, the host processor can control the data exchanges between the accelerator and the rest of the SoC.

3.1.3. Virtualization

Besides extending the Madeo ADL with the configuration plane specification and the controller generation, Drage enables the generation of a VHDL-based fully functional architecture prototype. This prototype is directly usable in cycle-accurate simulators to emulate the designed architecture. Moreover, it enables architecture virtualization on top of commercial-off-the-shelf FPGA architectures[53]. This feature is particularly interesting in association with the architecture exploitation flow since, beside obvious gains in the architecture simulation context, it opens the way to: a) open bitstream model, and synthesis toolchain; b) application portability across multiple vendor FPGAs; c) dynamic, partial, multi-context reconfiguration
even when the target COTS FPGA does not enable it; d) orders of magnitude faster reconfiguration – in
the context of coarse-grain architectures. However, reconfigurable unit virtualization is out of scope of this
study. The interested reader is thus encouraged to look at [53, 54] for more detailed presentation of this
particular scenario.

3.2. Re-targetable Compilation-Flow for Embedded Reconfigurable Units

While the previous section was focused on the design of embedded reconfigurable units, this section
presents the exploitation flow introduced in Fig. 2 and its articulation points with the configuration and ar-
chitecture models. The exploitation-flow starts with application partitioning. Applications are decomposed
into partitions based on the sizing of the existing configuration regions. Since the circuit in a region can
be swapped in/out, a smart sizing of the regions prevents internal fragmentation. Every partition is then
implemented (resource allocation) using place&route algorithms. From the results, a collection of partial
bitstreams (binary representation) is extracted. It should be noted that the flow presented here is somewhat
specific to fine-grain and medium-grain reconfigurable architectures. In the case of coarse-grain architectures
the architecture abstraction is closer to the application level thus the synthesis flow is simplified and more
closely resembles a classical software compilation flow. Nevertheless, our approach fosters the benefits of a
model-driven compositional software architecture thus specific exploitation flows can be created according
to the specific requirements. The following subsections will focus on three important aspects of application
synthesis on reconfigurable units: a) high-level synthesis; b) physical-synthesis; c) bitstream extraction.

3.2.1. High-level Synthesis

High-level synthesis is responsible for converting the user application, represented internally as a high-
level CDFG, to the technology-specific low level representation (low-level CDFG). It takes into account the
local implementation on the fabric as well as optional synchronizations with a DMA controller (in SoC
scope) performing data transfers over a unit’s local memories.

The first CDFG transformation phase adds hierarchical inter-connected controllers [55]. The controllers
synchronize with each other through hand-shaking. The circuit specification must be deterministic and
inter-process communications conform to a blocking scheme. Our control model is based on Petri nets; each
controller is responsible for handling its own token, according to the control structure it implements. For
example, if a parallel constructs broadcasts multiple copies of the control token a synchronization barrier
takes place when joining-back the concurrent branches. Send/Receive operations conform to a process-to-
process pairing per channel, and the intentions to read or send signals are used for handshaking.

The second phase maps the netlist onto an implementation technology. A direct mapping is supported
depending on the availability of operator libraries. Moreover, primitive generators are available to implement
tailored operators depending on their IOs; a structural description is built up before on-demand specialization
(e.g. using a constant instead of a variable input), flatten operation and optimization (logic minimization).

The third stage does the retiming of operators to speed up the circuit. During this stage flip-flops are
inserted to split up the combinational path hence minimizing the cycle time.

3.2.2. Physical Synthesis

The physical synthesis step is responsible for computing the actual mapping between the target specific
low-level CDFG (obtained during the previous steps) and the reconfigurable fabric. It consists mainly of
placement and routing as progressive refinement phases. The placement step maps the application operators
 spatially on the computational resources of the reconfigurable unit, while the subsequent routing step maps
the logical connections between the operators to the physical routing resources present. The Biniou toolset
relies on the MoNaDe physical synthesis framework[56, 57]. MoNaDe implements the physical synthesis
process using a model-driven methodology, which decouples the algorithmic aspects of the physical synthesis
from the architectural and the application model. This approach enables the orthogonal composition of the
architecture / algorithm / application axes of the design space, thus enabling an incremental exploration
based on quantitative evaluations.
3.2.3. Bitstream Extraction

Once the application is allocated on the reconfigurable unit, the configuration bitstream is extracted. The bitstream is a binary representation of the resources needed to implement the application—similar to executable binaries (i.e., elf files). However, as opposed to the executable binaries, the bitstream represents the spatial resource utilization and not the temporal sequencing of instruction. The bitstream model is automatically constructed from the architecture specification by inferring the number of configuration bits needed for each tile and their association with the tile’s reconfigurable elements (LUT, multiplexer, etc.). Once the user application is mapped onto the fabric, the bitstream is extracted by iterating over the configuration regions and collecting the configuration bits for all the tiles. Different bitstream extraction strategies can be designed by altering the iteration order over the architectural tiles and/or their elements.

The toolset can export the bitstream hierarchical structure, either for documentation purposes (via a GraphViz backend), or as an intermediate format, which can serve as basis for third party tool integration (i.e., external bitstream generators).

3.3. Multi-level Simulation

The following paragraphs briefly introduce the Biniou simulation infrastructure. The interested reader is encouraged to refer to [8] for more details. In the Biniou framework, the application simulation is seen as a transversal concern that ideally should be enabled at every step of the exploitation flow. Thus allowing the user to incrementally validate the application focusing on the abstraction-level dependent properties. The Biniou framework traces the application refinements through the synthesis process enabling high-level signal reconstruction from the low-level (bit accurate) representations. This traceability feature directly enables the high-level debugging presented in Sec. 4. However, Biniou does not currently support the direct simulation of the place-and-routed circuit, nor the bitstream interpretation. This is mainly due to the high technical complexity of creating realistic simulation infrastructures at these abstraction levels. Thus it relies on external tools (such as ModelSim[9]), to offer simulation capabilities at the architecture level. Therefore, once the application bitstream has been generated, it can be used to configure the architectural prototype and perform external simulation on the virtual prototype.

```verilog
module test_bench;
  reg CLK = 0;
  reg RESETn = 0;
  reg armlock_0 = 0;
  reg armlock_1 = 0;
  reg resume_test = 0;

  // Clock generation
  always begin
    #(500/2) CLK = ~CLK;
  end

  initial begin
    fork
      initial begin
        #(1 * 500) RESETn = 1; // Protocol
        #(3 * 500) armlock_0 = 1; // Protocol
        #(5 * 500) armlock_0 = 0;
        #(7 * 500) armlock_1 = 1;
        #(91 * 500) resume_test = 1; // Continue token
        #(92 * 500) resume_test = 0;
      end
    join
  end

  initial begin
    @(posedge t_1_1);
    $display("SIMULATION FINISHED");
    $stop
  end
endmodule
```

Listing 3: The generated code embeds system events to drive the external simulator, it conforms to a forced model of register transfer logic.

12
It should be noted that besides simulation and debug support, the Biniou simulation framework enables the generation of low-level test-bench templates composed of synthesized high-level execution traces implementing the applications communication protocol which can be specialized to drive the low-level cycle-accurate simulations. Listing 3 shows such an example which is derived from a high-level simulation session.

At the CDFG level, the first simulation phase performs a system simulation of the HW/SW components, without considering the communication architecture. Here, the system (and the inter-component communication that drives all synchronization mechanisms) is modeled at an abstract-level by the exchange of events or tokens. The time taken to generate and consume a communication event only depends on the size of the data being transferred. The output of this step is a transaction level modeling (TLM) trace, the events of which fire the execution of mapped logic.

The second phase performs cycle-accurate bit-accurate simulation of mapped logic, as shown in Fig. 8. Listing 3 illustrates how our environment translates a TLM trace into a forced-model to an external simulator.

3.4. Going Further: Entering the Object World

The object paradigm is very convenient to handle structures owning states and behavior. Compared to typical objects, a circuit keeps its state in registers while its behavior is hardwired. A minimal message broker (MB) pattern provides an object encapsulation to the circuits: specific classes are instantiated to create the surrogates for hardware circuits under use. The MB guarantees the basic mechanisms for transparent distribution using the MB Hardware driver mechanism. The MB is responsible for marshaling/unmarshaling Smalltalk objects to a byte stream for transmission to the driver (order matters when considering status bits).

```
setUp>
"Platform definition"
  fpgaModel := FPGAModel redPillArray new.
  fpgaNetList := fpgaModel grammarDefinition buildVHDL.
  application := UIBiniou new doIt.
  application := applicationVariable first name: 'test' arg: '5'.
  applicationNetList := application generateSupport; instantiateSupport.
  "Place and route"
  module := fpgaModel implement: applicationNetList at: 4@4.
  partialBitStream := module produceBitstream.
  "Platform configuration"
  fpgaNetList loadRegion: 1 context: 0 with: bitstream; sync; activate: 1@1; sync.

  testSimulatedApplication>
"ModelSim interface"
  modelsim := Modelsim new loadVerilog: applicationNetList.
  bulletTimeStart := applicationGUIm simulateUntil.
  modelsim run: bulletTimeStart.
  applicationGUIm context keysAndValuesDo: [:name :value |
    self assert: value = (modelsim readBack: name).].

  testHardware>
  fpgaNetList run.
  applicationGUIm context keysAndValuesDo: [:name :value |
    self assert: value = (support perform: name asSymbol).].
```

Listing 4: SUnit code operating on circuits

The simulator and the local substitute of the circuit in use (mock) provide the same API. A direct benefit of our approach is the automation of characterization tests. Listing 4 presents a Biniou script setting up simulation-based SUnit tests: a) setUp method (1–14); b) low-level simulation using a third-party tool (16–22), c) and real hardware execution\(^4\) (24–27). As we will see further on, in Red Pill (Section 4), this mechanism complements the variables/signal traceability (variable reconstruction based on signal values) to offer abstract analysis, and observability. Combining it with the ability to perform fast changes

\(^4\) time is ignored for the sake of simplicity as the hardware runs two orders of magnitude faster than simulation
(through HLS) and controllability (through adding dedicated controllers) opens the way to high-level circuit debugging.

Going beyond debugging, this approach offers the promise of "blind" mixing of software objects and hardware circuits, delegating the hardware-software task-migration support to the operating system (OS). This can be achieved via a simple mechanism for load/unload operations, as OS primitives. Listing 5 illustrates a simple round-robin mechanism for task (circuit) replacement over the platform. Support objects are used to capture/restore flip-flop values (process state). Circuit objects handle their region-context information as well as their support object (contexts are supposed pre-loaded in this case).

```plaintext
roundRobin
| platform isRunning | while True : |
| circuit := activeCircuits removeFirst |
| circuit support snapshot |
| waitingCircuits addLast : circuit |
| entering := waitingCircuits removeFirst |
| fpgaNetList sync ; activate : entering region@ entering context ; sync |
| entering support update |
```

Listing 5: Naive Round-Robin based OS scheduler

Research efforts such as SqueakNOS [58] alleviate the need for a "typical" OS when managing an object-oriented platform. By corroborating the work presented in this study with such complementary approaches we could ease the integration of reconfigurable accelerators within COTS computer systems (i.e. PCs, tablets, smartphones, etc.).

4. RedPill: Hardware introspection and controllability

Cost-reduction policies and time-to-market pressure, lead the hardware designer to operate following a component-based approach (bottom-up integration). This solution is at variance with a top-down (system view) approach that promotes a derivation of the platform from a list of requirements. The component-based approach relies on a set of components that traditionally come with their own verification environments (debuggers, status registers, etc.). As a consequence, a key concern is to grab many partial views into a whole picture so that designers can take sound decisions. This requires the mixing of several levels of abstraction and formalisms, as the SoCs are intrinsically heterogeneous. This also requires control mechanisms to support distributed operations such as time-cut. Since in general no process in the system has an immediate and complete view of all process states, only a distributed control can help in approximating the global view of an idealized observer having immediate access to all processes. This issue is well known to distributed software engineers[59]. The complexity in a SoC scope comes from offering a unified control mechanism as well as snapshot facilities.

Software agile programming promotes a just-fit approach. Testing has come to be a key piece of the designer toolbox. Besides, because any evolution may cause a regression, and because only the debugger can provide a significant insight, even component-based development makes massive use of the debugging environment.

On the other hand, when designing a hardware product, the main design scheme conforms to the waterfall model, with the early decisions being revised as little as possible. Assertions are not seen as executable specifications but as safeguards to be checked during the simulation phase. Hardware design exhibits extremely long development cycles compared to software, with specific issues in testing, as testing at-speed often requires in-situ testing.

4.1. Reconfigurable hardware

Despite being accepted that reconfigurable architectures increase the designer productivity by providing flexible hardware support, productivity remains strongly dependent on the development environment and ease of verification. In the hardware design world, execution is fast, but debugging lacks some of the facilities that exist in software design (context-stack analysis, replay, continuation, hot-code replacement). Therefore, the key is finding the right tradeoff between the software-debugging flexibility and the hardware
execution performance. Apart from the performance issues, the main enablers are: a) the observability; b) the controllability, the ability to steer the execution flow; c) and traceability (abstract analysis), offering a programmer-oriented view of the execution by preserving the programming model (e.g. source code vs byte-code, variables vs registers, etc.). Abstract analysis, also referred to as traceability, translates the observed values back to high-level values, thus speeding up the understanding by focusing on key aspects such as algorithmic issues. This is very important since most of the changes required to fix the deviations happen at high-level.

4.2. RedPill Overview and Integration in the Biniou Toolset

RedPill is an environment that aims at offering introspection and controllability of hardware elements generated using Biniou. Keeping the bi-directional link (high-level to hardware and back) alive supports abstract analysis. Moreover, changing the value of variables is possible to stress some hypotheses, however this is not sufficient to offer 100% software-like debugging facilities. However, RedPill introduces the basic functionalities that pave the way to debugging.

RedPill brings ideas from typical Smalltalk-like IDEs to the world of reconfigurable hardware, especially assertion-based debugging. It also takes advantage of Smalltalk polymorphism to support domain variability (reconfigurable target), and benefits from a wide legacy work [60]. RedPill can be seen as having a dual purpose: on one hand to open the hardware world to the dynamic language developers, and on the other hand to bring a real debug IDE for reconfigurable SoC designers and end-users.

Providing full visibility over the system state evolution is realized by logging and tracing values/signals. This is not applicable though, if the traces exceed the storage capacity or high-speed execution is mandatory. However, offering a way back to previous states without paying the cost of full state storage remains possible by combining controllability and multiple runs. As long as the circuit execution remains deterministic the environment can capture and re-issue proper external stimuli on-time to drive the circuit into the desired configuration.

Our approach combines the best of both simulation-based and in-situ solutions, by focusing on controllability and observability, while preserving the speed of hardware execution. It offers in-time traveling capabilities, while keeping the link active to/from behavioral specification of the circuit. Our solution is integrated with the Biniou high-level synthesis framework (from high-level code to circuit translator), and verification/debug of applications happens at different abstraction levels depending on the designer’s needs [61]. The high-level specification acts as characterization for the circuits at selected milestones of the execution flow.

Software execution serves as a characterization test during verification. A gcc-compatible C program is generated out of the source specification. This program is based on POSIX threads and emulates memories as files in which to read/write. This feature saves the designer’s time by favoring simple verification schemes. Early verification cut-off most of the functional mis-coding. Besides, despite the increase of execution time along with complexity, it remains an affordable non recurring cost. An example of such an external tool interfacing is further provided by Figure 8.

The application appears as a set of concurrent processes, some of which are granted for storage access. For application specification the C language is used, since it offers the promise of a wide-spread and well understood language, although being enriched with specific CSP-like constructs[62] for inter-process communications and looping structures for spatial unrolling. Based on a set of working files (C + XML address generators + processes topology), flattened C code is generated.

All the variables are looked-up and logged, during either assignment, or channel-based inter-process communication. Breakpoints over variables are conditional, while breakpoints over operators are simple breakpoints. We consider three kinds of probes: watchpoints provide full visibility over a variable, breakpoints freeze the execution and conditional breakpoints are breakpoints that react to assumptions over the value of a given variable.

4.3. The Present: Debug Capabilities for Embedded Reconfigurable Units

As reconfigurable architectures allow design modification, they are good candidates to embed a support for software-like debug capabilities.
Our high-level synthesis framework – that can be seen as a retargetable compiler – addresses fine-grained reconfigurable devices (IPs) connected to local memories. This conforms to recent works such as [63] that focus on SoC integration of reconfigurable IPs.

In RedPill, observability is enabled through adding watchpoints. Hardware counterparts to software variables are signals. Watchpoints are automatically inserted in the design on demand, by wiring the probed signals to the top interface. Then traces can be analyzed, making visible any internal signals by extending the interface of the modules recursively, and performing signal binding. Preserving the generality of the approach prohibits the picking of internal signals at run-time using dedicated tools such as Chipscope.

The gain is portability as observability is restricted to IOs at the expense of over sizing the interface. This would be equivalent to adding some new accessors to a class over private states compared to using an inspector that can access any instance variable, private variables included.

As understanding the insight of an execution often remains a matter of "what if..." and "it’s when...", software debugging tools enable to stop the execution at a given point, to process step-by-step or to restart the execution of a function. This relies on conditional watchpoints, for which an operator is used to perform a boolean operation on the probed signal. In hardware, controllability comes from using embedded controllers that support start/stop/restart actions. The circuit is based on the coupling of hierarchical controllers and datapath portions [55]. Hence, freezing a controller propagates a lock progressively throughout the full circuit – be this portion connected and synchronized with other sub-circuits or not.

When the circuit is on hold, extracting the global state can be achieved with no timing pressure as the circuit remains frozen until a step or a continue command is issued to the controller. At this point, watchpoints are read back. The variable’s current values are then updated in the front-end as this is much easier to understand than bit-level signals.

Conditional breakpoints take as inputs the probed signal, the value to compare and control bits for selecting the comparison operator, and depending on the comparison, halt the circuit through its controller. This allows the dynamic change of the argument’s value, the activation/de-activation of probes and the change of the probe’s condition. The only part to remain static is the pool of probed variables/signals.

The condition result is wired to the top hierarchy giving its status. Hence inserting a conditional watchpoint, consists in adding an operator and two simple watchpoints: one on the value to probe, and the other on the boolean output of the operator. Conditional breakpoints have to be very flexible, being either valid or not, supporting hot replacement of conditions. The breakpoints to be validated must be selectable on demand so that only a subset of them is active at a time. This offers the promise of multiple runs, with different active breakpoints so that the designer can speculate on the cause of the error and roll-back to the original deviation.

As a consequence, this enables stopping under a given condition, then tracing internal signals on which to tune the probes’ conditions. Not only can the execution be resumed but it is also cost free to start again a new run again with the newly tuned conditions in order to reach an earlier stop point. This delivers the standard software debug facilities, such as digging in the functions call stack, that hardware designers traditionally lack.

More complex schemes must be implemented to address temporal assertions. However the underlying mechanism is still the same: Once the states are captured, the assertion verification becomes a combinatorial logic computation.

4.4. The Future: Debug-Aware Reconfigurable Platforms

As previously mentioned, the reconfigurable platform architect’s client is the hardware designer. He is the one who cares about having a verification scheme. The RedPill solution offers some gains, and can likely be ported over existing FPGAs. A meet-in-the-middle approach, combining both the validation and the verification schemes can contribute to pushing the boundaries of these innovative improvements.

The idea is to inject these facilities into the platform rather than to post-process the applicable netlist. This work is currently at the heart of the on-going ARDyT project, funded by the French National Research Agency. The purpose of ARDyT is to provide a complete environment for the design of a fault-tolerant and self-adaptable reconfigurable platform. It means that the reconfigurable architecture, the associated
programming environment and the management methodologies for diagnosis, testability and reliability are all within the scope of the project. The project aims to provide a reliable component for terrestrial and aeronautic applications, at a lower cost than commercial competitors and to provide an European alternative to avoid importation restrictions. Aging support, single event upsets, security issues all together fall within the scope of ARDyT.

The platform model has to be extended to incorporate new elements. These elements can be hardware primitives that exhibit higher robustness, better observability or simplify advanced introspection.

For example, some operators may carry meta-data with different semantics such as: resilience to faults, aptitude to detect impaired modules, etc. Depending on the external environment, the platform designer tunes his requirements, and the framework reacts accordingly. The physical synthesizers must support some yield improvement policies such as at least triple module redundancy (TMR).

Another aspect that is currently under investigation is to consider platforms that could include verification-oriented timers to stress PSL assertions.

5. Practical Design-Space Exploration and Debugging Infrastructure

The first part of this section presents the prototyping capabilities of the Biniou toolset through a case-study based on a generic coarse-grained reconfigurable core, named CGRA. After a brief presentation of the architecture, the prototyping of the computing and configuration plane is discussed, along with the Drage-enabled architecture virtualization on commercial FPGA. The second part of this section gives a practical view of the RedPill debugging infrastructure through a series of commented snapshots and simulation results.

(a) Overview of the CGRA architecture
(b) Application placed and routed on CGRA array (exploitation flow in Fig. 2).
(c) Layout of the prototype composed of a 4 × 5 CGRA array connected to a Microblaze (virtualization in Fig. 2).

Figure 4: CGRA architecture and Biniou synthesis results

5.1. Exploring a Coarse-grain Reconfigurable DSP Core

CGRA pipelined core is structurally similar to the PiCoGA [4] and PipeRench [64] architectures. Fig. 4a shows such a 3 × 4 CGRA array. Each pipeline state (row) in a CGRA array is composed of ALU tiles and IOB\(^5\). Each ALU tile is modeled as a composite element, embedding a function that supports three arithmetic operations (+, <<, −), IO multiplexers, registers, and a switch for intra-stages connections.

\(^5\)IOB – Input Output Block
The granularity of the CGRA arithmetic operations addresses DSP application and is larger than in the fine-grained FPGAs.

If the array’s size is too small to spatially implement the full application, then resources are virtualized through temporal multiplexing. These capabilities rely on the support for dynamic partial and multi-context reconfiguration resulting in an execution flow similar to that of the PipeRench architecture[64]. The interested reader is directed to [65] for a detailed account of this technique, and the Biniou evaluation results.

An indicator of reuse rate of tile descriptions is given by the heterogeneity factor $H$ computed as:

$$H = \frac{D}{T}$$

Where $D$ is the number of different tiles and $T$ the global number of tiles. The lower $H$ is, the less the number of descriptions issued. Heterogeneousness factor $H = 0.76$ for a minimal size of the FPGA array, with $4 \times 2$ IOBs and $3 \times 3$ CLBs, while $H = 0.03$ for a scaling to $4 \times 16$ IOBs and $17 \times 17$ CLBs. A noticeable point is that as the compute density is lower than for CGRA, small arrays are not usable. As a consequence, $H$ hits high values. For example in Fig. 5a(left), $H = 0.83$ while in Fig. 5a(right) $H$ decreases as the array’s size increases with $H = 0.03$.

**ADL Expressivity.** $H$ alone does not denote the internal complexity of tiles hence does not measure the effort required to reach a full hardware model. Other metrics can be introduced according to the designers needs. For example, the ADL expressivity can be introduced to emphasize the effort needed to describe the architecture. This metric is the ratio between the number of lines of code in Madeo ADL and the number of lines of code of the generated VHDL code.

ADL-based prototyping aims at increasing the designer’s productivity. Fig. 5b compares the ADL expressivity metric between 4 array configurations, containing 6, 20, 72, 272 tiles. Two parameters drive the number of generated VHDL lines: the array size and the number of contexts implemented by the configuration plane regions. These factors impact the number of instantiated cells (tiles and configuration registers) and the wires for interconnections. The bigger the architecture and the more contexts it supports, the larger the gain. For the CGRA cases in Fig. 5b, the optimal gain is reached for 4 contexts with 272 tiles. For FPGA, even small arrays bring gains thanks to the routing complexity and the induced number of wires to be declared.

**Exploring the Configuration Planes.** The decision criteria for configuration plane structure comes from defining the reconfiguration regions under several criteria. The desired throughput (bits/cycle) drives the number of registers per reconfigurable region.
(a) Impact of throughput specification (8, 16 or 32 bits/cycle) over configuration latency and register consumption with respect to array size

(b) Synthesis results for different CGRA array sizes

Figure 6: Impact of the configuration plane throughput and prototype synthesis results

Fig. 6a provides the amount of registers depending on throughput. Despite using classical throughput values, no restriction over the values exists. For example, the first array (6 tiles) requires a rounding of 4 bits, but using a non-conventional 26 bits value would avoid the over cost with a 12 cycles configuration latency.

Bitstream sizing. Bitstream size is automatically derived from the bitstream model. The DFF line in Fig. 6a provides an insight into the evaluation of four CGRA arrays. This leads the designer to determine best fit storing strategies for bitstream (memory sizing, compression, etc.). Heterogeneity of tiles brings a non linear increase factor for CGRA bitstream depending on the array’s size.

Virtualization on Commercial FPGA. The virtualization experiments were carried out using the Xilinx XUPV5-LX110T[66], which contains a Virtex-5 FPGA with a capacity of 17290 slices. Each slice is composed of 4 6-LUTs and 4 multiplexers. Even though this architecture contains 64 DSP slices, currently they are not used by our virtualization backend. The system prototype consists of the CGRA core and its associated configuration controller connected to a Microblaze[67] soft-core processor. Fig. 4c shows the layout of the system emphasizing the CGRA core and the Microblaze. The connection between the processor and the CGRA prototype relies on an adapter with two Fast Simplex Link[68] buses (input/output). Fig. 6b overviews the synthesis results in terms of resource utilization – slices, registers, LUTs – and occupation ratio (Occ.). The ALU tiles of the CGRA arrays are implemented using FPGAs LUTs, thus increasing the array size directly increases the LUT usage. The global occupation factor (Occ in Fig. 6b) represents the percentage of slices allocated on the Virtex-5 FPGA, and we can see that it increases with the size of the array. These results emphasize the important trade-off between the architecture size and the application temporal multiplexing. The higher the array size, the smaller the number of temporal partitions, thus the higher the performance of the application at the expense of higher resource usage.

5.2. Practical Debugging Infrastructure

Biniou front-end and RedPill debugger. Our front end (Fig. 7a) invokes the Biniou synthesizer [60] and either simulates or implements the resulting RTL netlist. The netlist simulation allows the computation of the variables’ current value bottom left, Fig. 7a; the clock cycle and the active breakpoints are provided as an aid to the designer. The interface supports adding/removing (conditional) breakpoints, and injecting control tokens such as step, resume, etc. In Fig. 7a variables are probed (t1.1...t1.6), a breakpoint is set (top left) if t1.5 > 4, and the execution is controlled (bottom right) through run/step/restart commands plus internal view access on demand.
Variable/Signal Traceability. Figs. 7b and 7c illustrate two complementary views of the circuit. Fig. 7b offers a structural view of the application mapped onto the target architecture, while Fig. 7c shows a remote reference of the circuit as an object with instance variables corresponding to the initial application variables and reflecting the circuit’s state. The state of these variables is updated on demand, by "drilling" the hardware. From a practical point of view, the variables are extracted only when the execution is on hold. This time window referred as "bullet time" in [69] corresponds to the introspection time shown in Figure 8.

Breakpoint activation. Fig. 8 illustrates a common view of the internal states and signals of a circuit under test. It demonstrates the controllability over a circuit based on probe activation. The clock signal is highlighted and the circuit activity is reflected by low-to-high and high-to-low transitions of signals. The first circle points out the rising edge of a conditional breakpoint. This leads to an inactive period where no transition happens. As soon as the restart signal is issued, the execution resumes. This inactive period enables the extraction of the internal state of the circuit for in-situ testing. This internal state can be further post-processed to favor abstract analysis, as an example through an object representation.

6. Conclusion

Current SoCs embed several units, some of which may be reconfigurable and provide a hardware support for compute-intensive tasks to be accelerated. The reconfigurability preserves the ability to update the circuit on demand and in the field, that results in a product-life increase. In particular, in a demanding context, that is characterized by fast changing needs and shorter time-to-market, such a feature simply makes sense.

However, hardware performances remains bound to tailoring the architecture to classes of applications. A first requirement lies in the ability to generate dedicated architectures at will to be embedded in a platform. This also requires the generation of well suited SDKs that support efficient use of the architecture when implementing an application. These SDKs implement metrics and drive the validation process. The first half of this paper focuses on these concerns and demonstrates that our framework supports both of these issues. This happens through model-based engineering and dynamic language use.

This modeling approach carries the following benefits: a) The learning curve pays-back, since Biniou relies on two orthogonal DSLs that capture all reconfigurable architectures. The designer gets in return dynamic partial reconfiguration facilities (Fig. 3b), exploitation SDK (Fig. 4b), and real prototype (Fig. 4c). b) Our framework generates a fully-functional prototype using a mainstream synthesizable format.
Figure 8: Multiple level simulation of the design under debug: high level simulator (left) and Modelsim simulator (right) [9]. The probe fires the distributed locking mechanism. The processes freeze waiting for a token that the probed operator is prevented from delivering. The whole execution stops following a temporal wave (appearing as hold latency) as is usual in distributed systems. The probe freezes the circuit until a new signal is issued to resume the execution.

(VHDL files), while maintaining productivity in a sharp-rise. c) Both the application and the prototype are reified and offer object encapsulation, which eases unit testing and alleviates the need for specialized hardware skills when using these platforms. A minimal round-robin time-sharing scheduler for hardware tasks, is provided as demonstrator.

The next enabler to supporting this technology in off-the-shelf PCs is the delivering of debug capacities to the platform’s end-user. When developing complex software, productivity owes a lot to advanced debuggers. Unfortunately, hardware development is far behind in terms of debugger use. The second half of this paper illustrates how our framework generates extra hardware circuitry that supports software-like debugging facilities.

Among other techniques, debugging object-oriented code can be gained at the cost of, either decorating the bytecode, or instrumenting the virtual machine. In a similar fashion, our framework proposes either to decorate applicative netlists with debug-oriented controllers or to instrument the platform. The latter is at the heart of the Ardyt Project. This project aims to offer an all-in-one solution to design a low-cost and self-healing reconfigurable platform, to be used in aggressive environments such as avionics.

Acknowledgements

Part of this research has been supported by the ARDyT project (ANR-11-INSE-15).

References


