# Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing

Miljan Vuletić, Laura Pozzi, and Paolo Ienne
Swiss Federal Institute of Technology Lausanne
Processor Architecture Laboratory
IN-F Ecublens, 1015 Lausanne, Switzerland
{Miljan.Vuletic, Laura.Pozzi, Paolo.Ienne}@epfl.ch

**Abstract**

*Despite enabling significant performance improvements, reconfigurable computing systems have not gained widespread acceptance: most reconfigurable computing paradigms lack (1) a unified and transparent programming model, and (2) a standard interface for integration of hardware accelerators. Ideally, programmers should code algorithms and designers should write hardware accelerators independently of any detail of the underlying platform. We argue that achieving portability and uniform programming with only limited loss of performance is one of the main issues that hinders the widespread acceptance of reconfigurable computing. To make reconfigurable computing globally more attractive, we suggest a transparent, portable, and hardware agnostic programming paradigm. For achieving software code and hardware design portability, platform-specific tasks are delegated to a system-level virtualisation layer that supports a chosen programming model—much in the same way platform details are hidden from users in general-purpose computers. Although an additional abstraction inherently brings overheads, we show that the involvement of the virtualisation layer exposes potential optimisations that compensate the overheads and bring additional speedups. As a case-study, we present a real design and implementation of a number of building blocks of such system and discuss the challenges involved in materialising the others.*

## 1 Introduction

Temporal computation (i.e., computation in software running on general processors) prevails in contemporary computing systems thanks to its generality and the advantages of post-fabrication programmability. Spatial computation (i.e., computation in specifically-designed hardware tailored for a particular application) provides greater performance and efficiency, but is usually limited to application-specific domains. Mask costs that grow exponentially for each technology generation [17] demand flexible spatial computing engines, thus lowering risks of committing designs to silicon. Although *Field Programmable Gate Arrays (FPGAs)* provide this flexibility, consumer products with millions of units cannot yet afford FPGA prices. However, the importance of *Reconfigurable Computing (RC)* that combines parallel execution with flexibility of defining spatial computation after fabrication [6] may raise in the future, since the number of production units where the use of FPGAs makes sense is increasing over years.

A way to overcome shortcomings of the underlying technology (e.g., clock frequencies that are often more than 5 times slower than those of ASICs and general-purpose processors) is to blend

Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)
1063-6862/04 $ 20.00 IEEE

temporal and spatial computing paradigms [7, 8] and use both general-purpose and reconfigurable hardware: this approach is backed by recent *Reconfigurable Systems-on-Chip* (RSoC) appearing on the market [1, 22]. Unfortunately, due to the lack of a unified programming methodology and of standardised interfaces for hardware accelerators, reconfigurable systems are more difficult to program; porting solutions across different platforms is usually burdensome [12]. Fostering RC toward general-purpose computing requires achievement of the properties that made microprocessor's way to common computers: *ease of programming and portability*.

### 1.1 The Goal

Microprocessors are easily programmed: applications written in high-level languages are compiled and run on a platform assuming a number of abstraction layers (e.g., *Operating System (OS)* services, standard libraries, communication protocols, and virtual machines) that hide platform-specific details and provide a plethora of services [16]. On the other side, programming reconfigurable systems is much more complex: software and hardware need to be written/designed and then interfaced [5]. Exposing platform-dependent communication details to software programmer and hardware designer burdens the high-level design approach and binds the solution to a particular platform.

In this paper, we suggest a programming paradigm for reconfigurable computing that is decoupled from details of the underlying platform. Application-level programmers should be liberated from reconfigurable hardware accelerator details: moreover, they should code applications without knowing whether application threads or tasks are executed in temporal or spatial manner. Dually, hardware designers should not be exposed to interfacing details: in particular they should not be exposed to where the data to be processed is located. The underlying architecture should support the programming model providing an abstraction layer to both software programmer and hardware designer. The abstraction layer should provide programming transparency and standard interfacing between software and hardware tasks through well-known communication paradigms (e.g., shared memory or message passing). This is the way we believe reconfigurable computing must take in order to become mainstream.

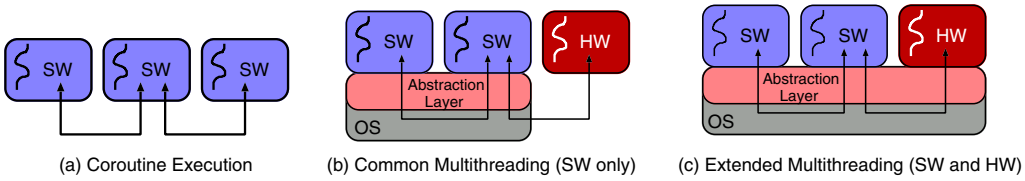
### 1.2 Paper Organisation

The paper is organised as follows. Section 2 discusses possible programming models for reconfigurable computing. In Section 3, we describe a possible system architecture to support the chosen programming model. Suppressing hardware and software boundaries enables a number of advanced techniques that we investigate in Section 4. Finally, we present a case study of a transparently-programmable reconfigurable system in Section 5, and we conclude in Section 6.

## 2 Transparent Programming Model

Existence of a common programming model that abstracts platform-dependent details can bring RC to the same rank with general-purpose computing—assuming that performance is only limitedly affected by the abstraction. Here we present a general-purpose programming model adapted for reconfigurable-computing.

Ideally, application-level programmers code high-level concepts into programming language statements, respecting a chosen programming model (sequential or parallel) and having no knowledge of the underlying computing platform. A good programming style that manages design com-




(a) Coroutine Execution (b) Common Multithreading (SW only) (c) Extended Multithreading (SW and HW)

**Figure 1. Multithreaded programming model. Coroutine execution (a) is the first approach to multithreading without system-level support. Common multithreading (b) supported by an abstraction layer (i.e., standardised libraries and OS services) ignores the existence of hardware threads. Extended multithreading (c) supports seamless integration of hardware threads: programming is transparent and interfacing is portable.**

plexity requires isolating functionally independent entities (e.g., processes, threads, functions, objects) that communicate information using standardised means (e.g., message passing, shared memory, function and method invocation). Without generality loss, we will consider a parallel multithreaded programming model, where a typical application has multiple execution paths, some of them possibly executing at the same time. Furthermore, we assume also that application threads communicate using shared memory paradigm.

### 2.1 Extended Abstraction

Historically, the first multithreaded applications (Figure 1a) are realised through coroutine execution [11]: the burdensome task is on programmers to switch control and arrange communication; there is no system-level support for multithreading. Today, an abstraction layer (consisting of thread libraries, with possible OS support) is used for thread management and communication (Figure 1b). The layer defines a standard programming model for multithreaded applications [13], providing transparent programming and inter-thread communication no matter where threads are executed (e.g., local or remote processors). However, this programming model does not cover the possible existence of hardware threads (i.e., application parts being executed in reconfigurable hardware accelerators). The communication and interfacing of software and hardware threads are exposed to software programmers and hardware designers and, thus, are neither standardised nor portable.

We suggest here (Figure 1c) an extension of the abstraction layer for supporting standard and seamless integration of hardware threads, in order to obtain *transparent programming model and standardised hardware interfacing*. A software thread communicates to a hardware thread *in the same manner as if it were a software one*. A hardware thread communicates with other threads *using a standardised interface*. Both software and hardware rely on the services of the abstraction layer. Thanks to the abstraction layer, time-critical software threads can be migrated to hardware execution *without any notice by the rest of the software threads*.

### 2.2 Software Threads

Figure 2a shows a rough example of a software thread aware of hardware computation. The thread communicates with a hardware accelerator that computes elements of an output vector (C) by adding elements of input vectors (A and B). The hardware accelerator can access only a memory region limited in size. The region is mapped to the thread memory space, as well as the accelerator control and status registers. The thread initialises the input data and the accelerator, and then

```
/* Typical hardware accelerator version */

int *A, *B, *C;

read(A, SIZE); read(B, SIZE);
d_chunk = BUF_SIZE / 3; d_ptr = 0;
write(HWACC_CTRL, INIT);

while (d_ptr < SIZE) {
  copy(A + d_ptr, BUF_BASE, d_chunk);
  copy(B + d_ptr, BUF_BASE + d_chunk, d_chunk);
  write(HWACC_CTRL, ADD_VECTORS);
  while () {
    if (read(HWACC_STATUS) == FINISHED) {
      copy(BUF_BASE + 2*d_chunk,
        C + d_ptr, d_chunk);
      break;
    } else {
      do_some_work();
    }
  }
  d_ptr += d_chunk;
} ...
```

(a)

```
/* Transparent hardware accelerator version */

int *A, *B, *C;

int hwacc_id;
read(A, SIZE);
read(B, SIZE);
hwacc_id = thread_create(hw_add_vectors, A, B, SIZE);
do_some_work();
thread_join(hwacc_id);
...
```

(b)

**Figure 2. Burdensome and portable programming: (a) the software thread is aware of hardware-interfacing details—the code is platform dependent; (b) the software thread is unaware of hardware-interfacing details—the code is portable.**

launches the computation. While waiting for the accelerator to finish, the thread can do some useful work in parallel. Once the status register indicates completion of the computation, the thread synchronises its execution with the accelerator.

One should notice that the software thread is aware of platform-dependent communication and implementation-dependent accelerator control: the programmer takes care of partitioning the data and scheduling transfers to/from the buffer region, and controls the accelerator directly through its status and control registers. Programming is not transparent, since the program needs to be adapted if: (1) the buffer size changes or (2) the format of the registers change. Ideally (as shown in Figure 2b), the software thread should agnostically call the hardware accelerator as if it were a software thread by passing the basic parameters and then synchronise with it at some later point. Programming becomes *transparent* since all communication and implementation details are hidden in the introduced abstraction layer.

### 2.3 Hardware Threads

Hardware threads should not be aware of platform-dependent details. When executing an application containing software and hardware threads (e.g., shown in Figure 1) on a different platform, it should be possible to port hardware threads *without any changes* to their HDL (*Hardware Description Language*) code. As it is the case for software threads, it should suffice to recompile (actually, synthesise) them. In order to achieve this, hardware and software threads should be written respecting the same programming model: they should share the same memory space and comply to standardised parameter passing and synchronisation methods. It is the task of the abstraction layer to hide platform-dependent details.

Figure 3a shows an example of the hardware accelerator adding two vectors by generating platform dependent addresses, in order to access the buffer region. The size of the buffer and its address in the system memory map are hardcoded. Should the target platform change, with its buffer size and memory mapping, the code would need to be modified. Instead of generating physical addresses of the buffer, the accelerator can use virtual ones (as shown in Figure 3b) that belong to the

```
-- Initialisation
ptr_a <= BUF_ADDR;
ptr_b <= BUF_ADDR + BUF_SIZE/3;
ptr_c <= BUF_ADDR + 2*BUF_SIZE/3; ...

-- Computation
cycle 1:
  -- partition of A[]
  BUF_ADDR <= ptr_a;
  BUF_ACCESS <= '1';
  BUF_WR <= '0';

cycle 2:
  reg_a <= BUF_DATAIN;
  -- partition of B[]
  BUF_ADDR <= ptr_b;
  BUF_ACCESS <= '1';
  BUF_WR <= '0';

cycle 3:
  reg_b := BUF_DATAIN;
  reg_c := reg_a + reg_b;
  -- partition of C[]
  BUF_ADDR <= ptr_c;
  BUF_DATAOUT <= reg_c;
  BUF_ACCESS <= '1';
  BUF_WR <= '1';
  ptr_{a,b,c} <= ptr_{a,b,c} + 1;
  if (ptr_c = BUF_SIZE) then
  -- finished for a data chunk
    partial_finish;
  else
    cycle 1;
  end if;
```

(a)

```
-- Initialisation
ptr_a <= A;
ptr_b <= B;
ptr_c <= C; ...

-- Computation
cycle 1:
  -- object A[]
  VIRTMEM_ADDR <= ptr_a;
  VIRTMEM_ACCESS <= '1';
  VIRTMEM_WR <= '0';

cycle 2:
  reg_a <= VIRTMEM_DATAIN;
  -- object B[]
  VIRTMEM_ADDR <= ptr_b;
  VIRTMEM_ACCESS <= '1';
  VIRTMEM_WR <= '0';

cycle 3:
  reg_b := VIRTMEM_DATAIN;
  reg_c := reg_a + reg_b;
  -- object C[]
  VIRTMEM_ADDR <= ptr_c;
  VIRTMEM_DATAOUT <= reg_c;
  VIRTMEM_ACCESS <= '1';
  VIRTMEM_WR <= '1';
  ptr_{a,b,c} <= ptr_{a,b,c} + 1;
  if (ptr_c = SIZE) then
  -- finished for the entire vectors
    finish;
  else
    cycle 1;
  end if;
```

(b)

**Figure 3. Platform-dependent and portable VHDL-like code of hardware accelerator: (a) the hardware accelerator generates physical addresses—the code is platform dependent; (b) the hardware accelerator generates virtual memory addresses, the abstraction layer provides translation and synchronisation—the code is portable.**

application address space. In this way, the accelerator code embodies only pure functionality and communicates through standard shared virtual memory paradigm. The abstraction layer translates virtual to physical addresses and provides standard parameter passing and synchronisation.

### 2.4 Benefits of Encapsulation

The proposed framework not only facilitates software and hardware code portability. It also respects one of the basic principles of a complex-system design: encapsulation of functionally-independent entities.

As an example, consider again the hardware thread invoked in Figure 2a and assume that it now employs a non sequential vector traversal. Although it still performs the same vector-adding function, the new internal implementation of the hardware thread requires changes in the software thread implementation. Therefore, the encapsulation principle is violated. This is is not the case for the software thread in Figure 2b that uses the abstraction layer: the latter can perfectly handle the different type of traversal dynamically.

## 3 Support Architecture

We discuss here a possible reconfigurable architecture that can run applications for the chosen programming model. Given the choice of multithreading, the underlying architecture is naturally similar to a general parallel computer architecture [3]. We show how software and hardware threads

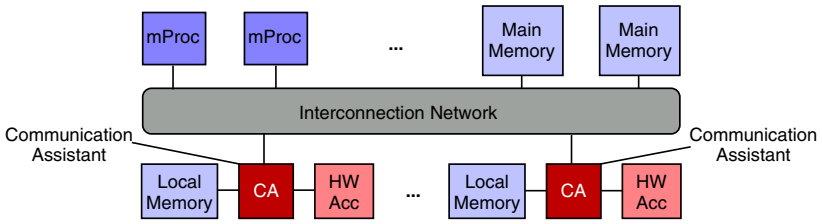



**Figure 4. Reconfigurable parallel architecture. Computing nodes (standard processors and hardware accelerators) communicate through a general interconnection network. Hardware accelerators are interfaced to local memories and to the rest of the system through standardised communication assistants.**

are seamlessly interfaced and executed, through a system-level support—the abstraction layer. It consists of a software part (libraries, OS services) and some hardware support.

Other researchers have addressed issues concerning reconfigurable hardware management [4, 21]. A system-level support hides the problems introduced by the finite amount of available reconfigurable logic. The resources are shared dynamically between applications. Instead of the management issues, the main task of our abstraction layer is to support transparent software and hardware interfacing and execution.

### 3.1 Reconfigurable Parallel Computer

Besides general-purpose processors, our reconfigurable computing system contains application-specific accelerators implemented in—but not necessarily limited to—reconfigurable hardware and running on behalf of a particular application. Figure 4 presents such architecture: general-purpose processors and reconfigurable hardware accelerators communicate through a general interconnection network. A local memory at each hardware-accelerator node is accessible directly by the node and indirectly by others through the network (e.g., a simple instance of such architecture is Altera's Excalibur architecture [1]: a general-purpose processor—ARM, a bus interconnection—AMBA [2], and reconfigurable logic directly connected to a local on-chip memory, a dual-ported RAM; in Section 5, Figure 8b shows the details of the system).

Each reconfigurable node contains a hardware interfacing component that is actually a standardised communication assistant [3]. The communication assistant defines the hardware interface (i.e., signals and protocols) for the hardware accelerators. When an accelerator accesses memory, its communication assistant translates the application address space to the physical one (e.g., from virtual to physical addresses [20]). If the requested address is not in the local memory, the communication assistant guarantees correct execution: either by initiating a copy to the local memory or by accessing the remote data. The copying and remote accesses are performed and supervised by the abstraction layer (introduced in Figure 1). We call it the *virtualisation layer*, since it provides transparent and platform-independent communication of software and hardware threads.

### 3.2 Software and Hardware Interaction

Software initiates the execution of the hardware accelerator and passes parameters through the virtualisation layer. The accelerator accesses memory and synchronises its operation with other system components through the communication assistant. When done, it returns control to software execution through the virtualisation layer.

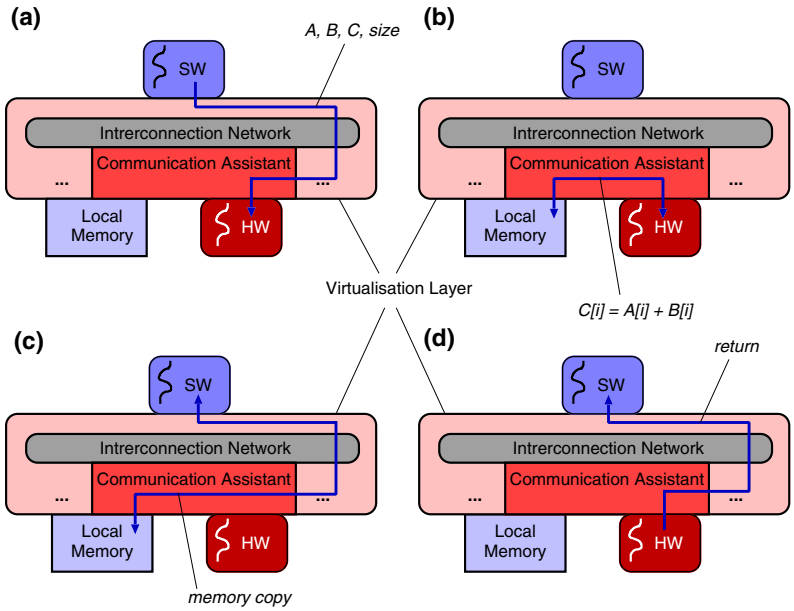



**Figure 5. Execution model. The software thread invokes a hardware thread through the virtualisation layer (a). The hardware thread accesses the local memory through the communication assistant (b). The virtualisation layer provides address translation and transparent memory copies (c). The hardware thread communicates the end of operation through the virtualisation layer (d).**

As an example, in Figure 5 we show execution of the simple application described in Section 2. In order to execute the hardware thread, the software thread invokes a service of the virtualisation layer. The service locates the hardware thread and its communication assistant. It passes parameters to the hardware thread as they are—virtual memory pointers of input and output vectors along with their sizes—through the communication assistant parameter exchange pool (e.g., a local memory designated for parameter exchange). In sequence, the assistant launches the hardware thread, which reads the thread parameters and starts its operation.

The hardware thread accesses memory through the communication assistant. Acting as a typical *Memory Management Unit (MMU)* [9], the assistant translates virtual to physical memory addresses. Since initial contents of the local memory are undefined, the first memory accesses generate misses. The communication assistant demands the virtualisation layer to copy initial pages of the required data to the local memory. When the local memory becomes full, the assistant copies back a produced portion of the output data to its original location, based on standard memory paging principles. In the same fashion, the assistant can provide locally resident data to other threads requesting access to the same addresses. Once the hardware thread finishes the computation, it signalises completion through the assistant and synchronises with the caller thread.

Application threads are unaware of physical location of data. Whatever address is accessed, the virtualisation layer manages transparent data movement and inter-thread communication. Different memory consistency protocols can be applied [3]. As in the case of software-only multithreading, application architects should be aware of potential penalties of coarse-grained data sharing. The virtualisation layer may try to minimise these penalties by both hardware and software means (as discussed in Section 4).

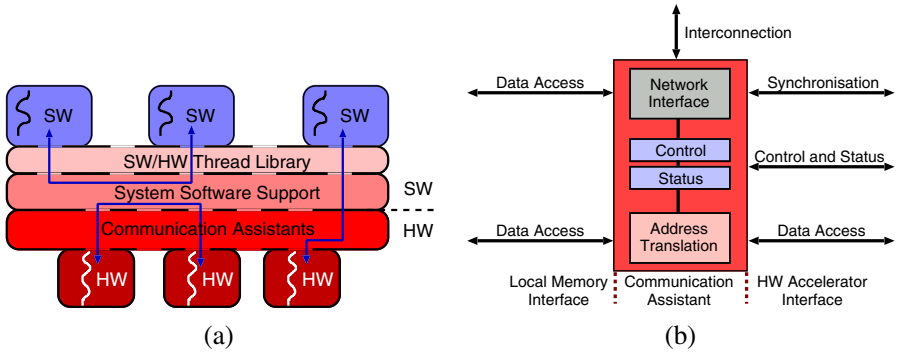



**Figure 6. Virtualisation layer and communication assistant: (a) software and hardware threads are hooked to the virtualisation layer; (b) the communication assistant defines interface signals and translates virtual addresses.**

### 3.3 Structure of Virtualisation Layer

The virtualisation layer consists of software parts (libraries, system software) supported in hardware (communication assistants). Its task is to abstract details of the support architecture. For example, software and hardware threads running on the architecture shown in Figure 4 are screened from the number of general-purpose processors, the type of the used interconnection network, and the local memory sizes. A multithreaded application can be compiled, synthesised, and run on any reconfigurable platform, provided that the virtualisation layer is implemented for the platform.

Components of the virtualisation layer are shown in Figure 6a. At the application software boundary (programmer-written software threads), libraries provide standard thread manipulations (creation, management, synchronisation) to software threads. Library functions are supported by the underneath system software that also controls hardware and provides system-level communication, memory sharing, and consistency management. At the application hardware boundary (designer-written hardware threads), communication assistants provide standard interfacing means (virtual address translation, memory sharing, synchronisation) for hardware threads. Communication assistants are distributed: each hardware thread has its own communication assistant.

Besides virtual address translation, the communication assistant (Figure 6b) defines interface signals that each hardware accelerator needs to implement. For example, hardware threads have to generate virtual memory addresses and use synchronisation primitives supported by the assistant (e.g., start and finish execution, create and join another thread, lock data). Once a hardware thread is called, it acquires the parameters through the assistant. The hardware complexity of the communication assistant depends on the chosen programming model and the hardware/software balance of virtualisation layer implementation. It can provide only basic communication primitives, since the rest can be implemented in the software part of the virtualisation layer. The assistant can be implemented either in custom or reconfigurable technology. Custom communication assistants can be more efficient but, on the other side, reconfigurable ones can be more flexible to protocol changes and extensions.

## 4 Advanced Techniques

The virtualisation described in Section 3 represents an additional layer on top of already existing ones (e.g., memory hierarchies, communication protocol layers); as such, it inevitably introduces

overheads. As in typical layering approaches (e.g., virtual memory abstraction), efficiency is traded for generality and programming comfort. Yet, in this section, we show that the virtualisation exposes further benefits and not always jeopardises efficiency.

### 4.1 Dynamic Optimisations

As it participates in application execution and communication transactions, the virtualisation layer can dynamically decide when and where to improve the execution. For example, with additional hardware support in communication assistants, it can detect memory access patterns generated by hardware accelerators [18]. Further on, it can predict future memory accesses and employ an adequate prefetching technique, trying to hide the memory communication latency.

For instance, in the case of the vector addition, the virtualisation layer can detect sequential data accesses. It can predict future accesses to be also sequential and supply in advance data partitions likely to be accessed in the subsequent phases of execution. With some hardware assistance within the communication assistant, the virtualisation layer can verify its past predictions and decide whether to continue or to bail out. An implementation example is given in Section 5. Prefetching is not limited to sequential accesses and a number of well-established techniques can be used [14, 10].

### 4.2 Unrestricted Automated Synthesis

Hardware synthesis from high-level programming languages has been a challenging topic for long time. Simple control and pure data-flow segments of high-level programs are straightforward to translate into hardware. On the contrary, advanced high-level language concepts and specific features are usually difficult to map onto hardware. For example, accessing memory in the C programming language by using pointers, possibly aliased, complicates hardware synthesis [15]. Also, a programming concept like recursive function call can pose problems for synthesising hardware. The existence of the virtualisation layer significantly simplifies and fosters automated synthesis.

Recalling the vector addition example, one can see that there is no need for special treatment of pointers. The virtualisation layer enables hardware threads to generate memory addresses as they are (i.e., virtual memory addresses within the applications address space). Suppose that a pointer that is synthesised into the accelerator is eventually aliased and that the aliasing pointer is used to access the pointed data. The virtualisation layer can transparently handle the access by copying the data to the access-generating node and, at the same time, take care of memory consistency (e.g., write access by the aliasing pointer may initiate data invalidation at the accelerator local memory).

The virtualisation layer can also appropriately treat recursion, as well as dynamic thread creation, thus significantly simplifying the synthesis. Imagine a hardware thread that, either directly or indirectly, calls itself. Since each call goes through the virtualisation layer, the latter can dynamically decide what to do: if the accelerator supports recursion (e.g., if it has an internal stack to preserve the state), the virtualisation layer will pass parameters and return control to the accelerator. Otherwise, the virtualisation layer can call a software equivalent of the accelerator, thus changing the execution manner from spatial to temporal. Through the shared memory mechanism, changes will be reflected back automatically to the originating hardware thread, once recursion is over.

### 4.3 Flexible Software to Hardware Migration

The existence of a virtualisation layer allowing for unhindered hardware synthesis also facilitates easier software to hardware migration, even dynamically, during runtime. Supposing that the virtualisation layer is part of a runtime support system (e.g., a virtual machine), it can be used for moving

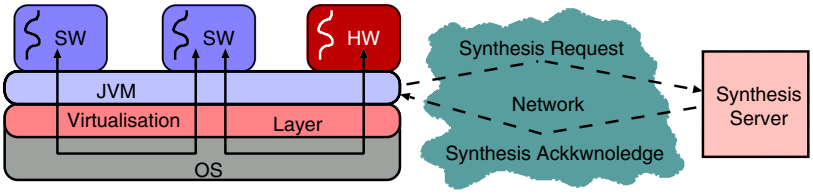



**Figure 7. Dynamic reconfiguration. JVM runs on top of the virtualisation layer. It requests migration of critical methods from software to hardware.**

critical software threads to specialised hardware: the runtime system can observe application execution and perform dynamic profiling to detect critical sections; the code of critical software threads can be synthesised (either locally or on a remote synthesiser) to hardware threads complying to the virtualisation layer; finally, the runtime system can choose which version to run, between software and hardware. Since the thread is hooked through the virtualisation layer, dynamic changes of computation manner (temporal or spatial) are invisible to the rest of the application.

Figure 7 shows a *Java Virtual Machine (JVM)* running on a dynamically reconfigurable system. At the beginning, the JVM executes and profiles *Java Byte Code (JBC)* of Java classes. In the same way as it dynamically decides which methods to compile *Just-in-Time (JiT)* to native binary code [23], it can also decide which methods to synthesise to hardware. The synthesis from JBC to the reconfigurable logic available at the platform can be done locally or externally. The JVM requests the synthesis over the network and continues execution of the classes. Once the synthesised method arrives from the synthesiser, it can be hooked to the application through the virtualisation layer. The JVM will dispatch future invocations of the method to the special hardware, transparently to the rest of the application. One should notice that the compile-once-run-anywhere principle of Java is preserved, since the same application could also run on platforms not supporting the dynamic software-to-hardware migration.

## 5 Real Implementation

In this section, we present our initial efforts towards a transparently-programmable reconfigurable computing system [20]. On a real RSoC system (an Altera Excalibur-based board running the GNU/Linux OS), we show that a simple programming model can be used to hide interfacing details of hardware accelerators. The programming model is supported by the virtualisation layer and by the underlying architecture. We also discuss future challenges that would bring us closer to the more general programming model, explained in Section 2.

### 5.1 Programming

Putting aside synchronisation and memory consistency problems due to parallel execution, we choose a simpler programming model where software and hardware threads are functions executed sequentially. In this way, the design of the virtualisation layer is simplified. Still, it provides hardware-agnostic software functions: functions executed in hardware (i.e., hardware accelerators) are invoked as if they were common software functions.

The particular implementation of the virtualisation layer is called the *Virtual Memory Window (VMW)*, implemented as the Linux OS module (called VMW manager) with some hardware support. It allows a hardware accelerator running on behalf of a user application to access the user-

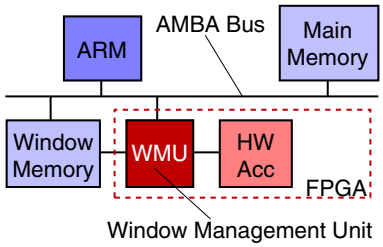
```
int *A; int *B; int *C; ...

read(A, SIZE); read(B, SIZE);

PARAM[0].address = A; PARAM[0].size = SIZE;
PARAM[1].address = B; PARAM[0].size = SIZE;
PARAM[2].address = C; PARAM[0].size = SIZE;

FPGA_EXECUTE(PARAM);
```

(a)




(b)

**Figure 8. Virtual memory window: (a) programming; virtual memory pointers are passed as-they-are to the hardware accelerator; (b) interfacing; the accelerator accesses local memory through the WMU.**

space virtual memory. Figure 8a shows a simplified—since there is no multithreading—and VMW-based version of the previous programming example (in Section 2, Figure 2b): the main function initialises input data and passes virtual memory pointers to the hardware function (i.e., hardware accelerator); it uses a standardised OS service to invoke the execution in hardware. Once the hardware finishes, the VMW manager returns control back to software. All communication details are hidden in the virtualisation layer: *programming is transparent and portable*.

### 5.2 Architecture

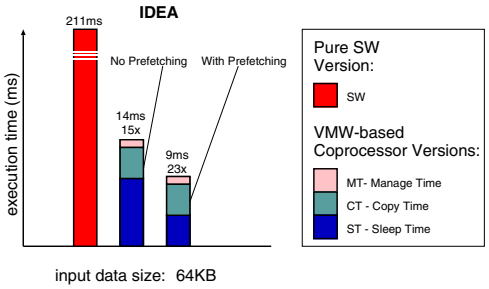
As shown in Figure 8b, we use the Altera Excalibur architecture [1] to support the function-based programming model. It is a simplified version of the general support architecture, presented and envisioned in Figure 4. The software part of the application runs on the ARM processor and accesses the main memory using the AMBA bus [2]. The hardware accelerator mapped onto the FPGA accesses the local on-chip memory through the *Window Management Unit (WMU)*.

The WMU acts as a communication assistant, representing a hardware support that is part of the virtualisation layer. It translates virtual addresses—as a typical MMU does—generated by the accelerator to the physical addresses of the window memory (a dual-ported local memory directly accessible by the main processor and the FPGA). The VMW manager accesses the WMU using an AMBA-FPGA bridge and services window memory misses, by copying data pages to/from the main memory. Page transfers are performed dynamically, *transparently for the application*. Furthermore, each hardware accelerator implements the interface defined by the standardised WMU (i.e., WMU-to-accelerator interfacing signals), thus *hardware interfacing is perfectly portable*.

### 5.3 Results and Future Challenges

We have ported two applications (a cryptography application, IDEA, and a common multimedia benchmark, `adpcmdecode`) to the VMW system and measured the execution times of software only, typical coprocessor (hardware accelerator directly programmed from the user application, similarly to the one shown in Figure 3a), and VMW-based versions of the applications [19, 20].

Significant improvements are achieved (e.g., the VMW-based version of the IDEA coprocessor is 15 times faster than the pure software version). More importantly, we have shown that the performance penalty of the overhead introduced by the virtualisation layer is limited: accelerators based on the standard VMW lag in performance compared to typical coprocessor versions (e.g., in




| Number of pages | Page size | Nonprefetching Misses | Prefetching Misses | Prefetching Late Misses | Prefetching Accesses |
|---|---|---|---|---|---|
| 8 | 2KB | 64 | 2 | 2 | 31 |

**Figure 9. Results for the IDEA coprocessors with and without prefetching. Dynamic prefetching successfully hides memory latency. The table shows different interrupts generated by the WMU and their distribution. In the prefetching case, beside misses, there are accesses (used to trigger prefetching) and late misses (an interrupt appearing while the missing page is already being transferred). The prefetching policy is successful since the number of misses goes down. The window memory is organised in eight 2KB pages.**

the case of the IDEA application, $15\times$ speedup compared to $20\times$) and this is the initial price to pay for programming transparency and portability. However, accelerator versions based on improved VMW (with dynamic prefetching implemented in the virtualisation layer [18]) even *outperform typical coprocessor solutions* ($23\times$ speedup). This was achieved thanks to the implementation of dynamic optimisations within the virtualisation layer, *without any need for intervention by the software programmer and the hardware designer*.

Figure 9 compares execution times of VMW-based coprocessors with software. Execution time of coprocessor versions consists of three components: (1) Manage Time (MT), the time required to manage the WMU, (2) Copy Time (CT), the time required to copy pages to/from the main memory, and (3) Sleep Time (ST), the time while the VMW manager is asleep, waiting to service the WMU requests. The WMU is extended with two 32-bit registers and few tens of logic gates, in order to support prefetching for sequential memory accesses. Based on the information retrieved from the WMU, the VMW manager dynamically predicts future page accesses and schedules their loads in advance. A part of the VMW sleep time is invested for anticipating future page accesses. If the predictions are correct, the coprocessor executes more smoothly, generating less misses (refer to Figure 9).

For the moment, the VMW covers only a simple programming model. Future extensions will include support for simultaneous thread execution. This would require supporting and implementing sophisticated synchronisation and memory-consistency mechanisms [3] in the virtualisation layer (i.e., extending its software part and communication assistants).

## 6 Conclusion

In this paper, we have addressed the problems that we believe are the major obstacles for proliferation of reconfigurable computing systems: (1) the lack of standardised programming paradigm and (2) the lack of standardised interfacing for reconfigurable hardware accelerators.

In order to bring RC closer to general-purpose computing, we show the need for: (1) a programming model that suppresses differences between software and hardware from the viewpoint

of programmers and hardware designers alike, (2) a parallel architecture that supports the chosen programming model, and (3) a virtualisation layer (consisting of system software and hardware support) that abstracts platform details from software and hardware components of applications.

Although the virtualisation layer inherently brings overheads, we have shown that it also fosters further optimisations: (1) dynamic prefetching techniques that hide communication latencies, (2) unrestricted automated synthesis that can facilitate flexible software-to-hardware migration.

As a case study, we have presented an implementation of a real reconfigurable system that is transparently programmable with portable interfacing. Future work should address the implementation of a system with the full-fledged parallel and hardware-agnostic programming paradigm.

## References


[1] Altera Corporation. *Altera Excalibur Devices*, 2003. `http://www.altera.com/literature/`.

[2] ARM. *AMBA Specification*, 1999. `http://www.arm.com/`.

[3] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/software Approach*. Morgan Kaufmann, San Mateo, Calif., 1999.

[4] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2003.

[5] G. De Micheli, R. Ernst, and W. Wolf. *Readings in hardware/software co-design*. Kluwer Academic, Boston, Mass., 2002.

[6] A. DeHon. The density advantage of configurable computing. *Computer*, pages 41–49, Apr. 2000.

[7] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, Napa Valley, Calif., Apr. 1997.

[8] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Napa Valley, Calif., Apr. 1997.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, Calif., third edition, 2002.

[10] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–63, June 1997.

[11] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1968.

[12] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain-specific compilers for programming FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-9(1):205–10, Feb. 2001.

[13] B. Nichols. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly and Associates, Sebastopol, Calif., 1996.

[14] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–26, San Jose, Calif., Oct. 1998.

[15] L. Séméria, K. Sato, and G. De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-9(6):743–56, Dec. 2001.

[16] A. S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Inc., Upper Saddle River, N.J., second edition, 2001.

[17] N. Tredennick and B. Shimamoto. Microprocessor sunset. *Microprocessor Report*, 1 May 2004.

[18] M. Vuletić, L. Pozzi, and P. Ienne. Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors. In *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, Antwerp, Belgium, Aug. 2004. To appear.

[19] M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for a portable reconfigurable cryptography coprocessor. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 2004. To appear.

[20] M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41st Design Automation Conference*, pages 948–53, San Diego, Calif., June 2004.

[21] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nev., June 2003.

[22] Xilinx Inc. *Xilinx Virtex ProII Devices*, 2003. `http://www.xilinx.com/`.

[23] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *Proceedings of the 8th International Conference on Parallel Architecture and Compilation Techniques*, Newport Beach, Calif., Oct. 1999.