Virtual Memory Window
for a Portable Reconfigurable Cryptography Coprocessor

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

Reconfigurable System-on-Chip (SoC) *platforms that incorporate hard-core processors surrounded by large amounts of FPGA are today commodities: the reconfigurable logic is often used to speed up execution of applications by implementing critical parts of the code as application-specific coprocessors. Cryptography applications are a good example of coprocessor applications: they are known to benefit significantly from spatial execution in hardware and have an increasing importance for mobile and ubiquitous computing. One of the main limits of FPGA-based coprocessors for these systems is the fact that both the coprocessor hardware description and the software program invoking are inevitably ridden with system details of the specific interface FPGA/processor: this limits significantly design reuse, impacts time-to-market, and makes development more complex. In this paper we present a portable reconfigurable cryptography coprocessor designed for a* Virtual Memory Window (VMW) *system. A VMW is a generic virtualisation layer composed of a hardware and an operating system components; it lowers the complexity of interfacing, increases portability, and makes it possible for the coprocessor to access the user-space virtual memory. The approach is illustrated here with the IDEA cryptography application running under Linux on a reconfigurable SoC, having its critical function mapped on the FPGA. A significant fraction of the speed-up inherent to hardware execution in the FPGA is preserved, while the hardware and software designs of the cryptography application become perfectly portable.*

1 Introduction

Due to the increasing mask costs of forthcoming deep sub-micron technologies, use of reconfigurable arrays will become more and more important in future computing platforms. There is a tendency to increase system flexibility and to favour hardware and software component reuse—allowing system architects to cope with complex in-field and even run-time programmability. Still, the speed and area efficiency of the FPGAs are far from those of general processors and application-specific accelerators implemented in ASICs. This suggests that standard high-performance SoCs may offer both computing approaches—standard processors augmented with reconfigurable application-specific parts [10, 11, 23]. Leading producers of reconfigurable devices already offer reconfigurable SoC platforms that consist of processor cores surrounded by peripherals, on-chip memories, and large amounts of reconfigurable logic [1, 28]. In some cases, they include special features such as embedded memories and arithmetic blocks suited for signal processing (e.g., Stratix family [1]).

With their potentials to exploit spatial computing on their hardware resources and their intrinsic flexibility, FPGAs are naturally suitable for cryptography applications. In the era of peer-to-peer networks, wireless communications, and mobile and ubiquitous computing devices, privacy concerns and the avoidance of eavesdropping have raised the importance of cryptography applications for the end-user community. Pure software implementations may be too inefficient for cryptography applications [18]. Furthermore, if real-time responsiveness is required, the natural solution is to move critical parts of the execution into specialised hardware [3]. However, binding rigidly the functionality to hardware decreases the potentials of future reusability, especially in the case of standard and protocol changes. Reconfigurable SoCs appear as an ideal compromise.

In typical cases of coarse-grained hardware/software partitioning for reconfigurable SoCs, critical code sections or functions are mapped to hardware accelerators. Usually, such sections of cryptography applications show a large degree of parallelism and contain a large number of arithmetic and bitwise operations. Although, in general, reconfigurable logic is not as efficient in computation as ASICs, methods and devices exist that can improve this deficiency [20, 1]. An unavoidable issue that hardware designers have to cope with is to interface the application-specific coprocessor with the rest of the reconfigurable SoC. One needs to take care of different peculiarities—bus hierarchies and protocols, shared and/or multi-ported memories, I/O ports, etc. On the other side, software programmers should be aware of the specific communication details between the application software and the coprocessor: for example, the availability and size of shared memory accessible by the processor and FPGA; if such memory is smaller than a dataset to be processed, the dataset needs to be partitioned and a load schedule developed. Furthermore, if the host platform is changed, a large share of the software and hardware parts must be redesigned, with a severe loss in productivity.

In this paper, we present a portable reconfigurable cryptography coprocessor based on *Virtual Memory Window* (VMW). Both the software and the hardware parts of the chosen cryptography application (IDEA) are interfaced to the rest of the system in a transparent and portable way. The VMW consists of a small platform-specific hardware and an Operating System (OS) module which, together, simplify software and hardware interfacing for software programmers and hardware designers. Our example of the IDEA cryptography application shows in detail the reduced complexity of the coprocessor programming and design paradigms. In this way, the portability of the application is significantly improved across different reconfigurable-computing platforms and the design effort is significantly lowered.

This paper is organised as follows: We present the concept of VMW in Section 2. In Section 3 we show design details of the VMW system and the IDEA software and hardware parts. The experimental setup used to demonstrate our system and the corresponding results are presented in Section 4. In Section 5 we discuss the related work. Finally, conclusions are drawn in Section 6.

2 Virtual Memory Window

VMW has been introduced [26] to simplify the design and the interfacing of reconfigurable application-specific coprocessors (for instance implemented in FPGA on a reconfigurable SoC) working under the control of a user-space application. When designed for a VMW-based system, an application (typically written in a high-level language—e.g., C or C++) and the corresponding coprocessor (usually written in a hardware description language—e.g., VHDL or Verilog) are made completely independent of the underlying hardware. It actually extends the well-known concept of the virtual memory [12] and applies it to make the communication between the coprocessor and the user-space software application transparent to the hardware designer and the software programmer.

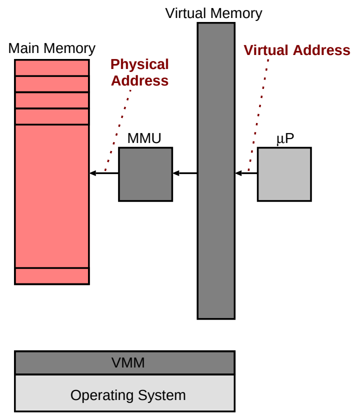

Figure 1. Classic Virtual Memory system.

Figure 1 shows a typical virtual memory system (without the details related to the secondary mass storage). The addresses known to the programmer are *virtual* in that they describe a memory system with no relation to the real one. The *Virtual Memory Manager* (VMM) of the OS supports the programmer's illusion and it is assisted in hardware by the *Memory Management Unit* (MMU). The ability to support this illusion of a large homogeneous memory has two fundamental advantages: (a) the simplicity of the programming paradigm and (b) the portability of the code across systems supporting the same OS but having a different memory hierarchy. The disadvantage is that the automatic allocation of pages by the operating system is, in general, suboptimal.

As virtual memory management, VMW can be used in a reconfigurable computers in order to hide data exchange details between the processor (i.e., the application software) and the coprocessor (i.e., reconfigurable hardware). Figure 2 shows how a VMW complements a virtual memory system. The presented VMW is built without loss of generality using a dual-port memory accessible by both the reconfigurable lattice and the processor. A user application running on the main processor and its corresponding coprocessor have the same virtual address space through the virtual memory mechanism. However, the coprocessor accesses the virtual memory through a different translation path. It uses the standardised translation hardware, called *Window Management Unit (WMU)*, and provides the window on the virtual memory; the window is supported by the OS part (*Virtual Memory Window manager*—VMW manager) that maintains data transfers from/to the user memory transparently for the end user.

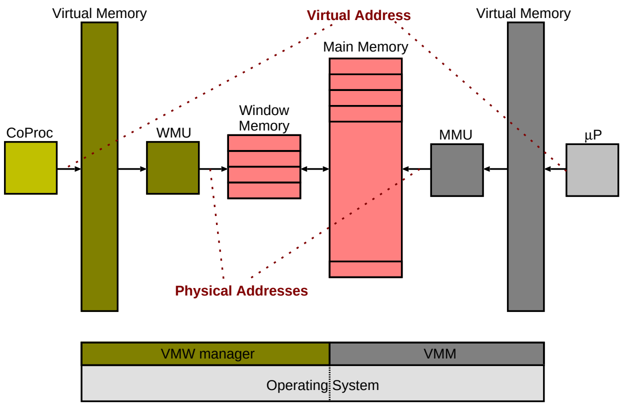

Figure 2. Virtual Memory Window for a coprocessor.

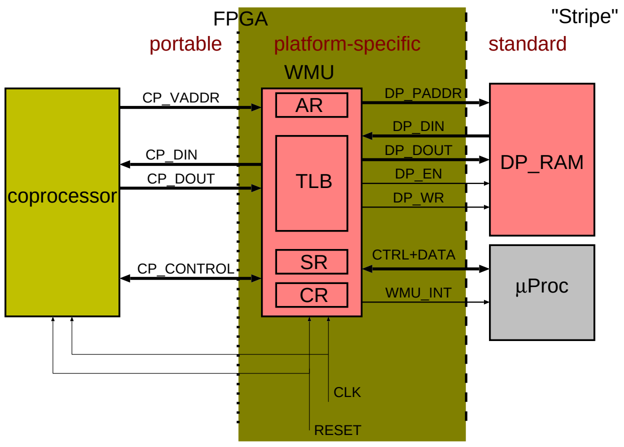

Figure 3. A coprocessor and a processor connected through a WMU.

3 VMW Interface and Coprocessor Design

Design details of the IDEA coprocessor are here used to exemplify the use of a VMW. The coprocessor software and hardware comply to our virtualisation interface. On the software side, the OS services are used to prepare data sharing and launch the coprocessor execution. On the hardware side, the coprocessor is interconnected to the rest of the system through a standard hardware translation engine (WMU), developed once for the platform, independently from the application at hand (IDEA in our case). We describe the platform-specific and application-independent hardware first, then the platform-specific and application-independent software support, to conclude with the application-specific but platform-independent IDEA coprocessor.

3.1 Hardware Interface: The WMU

As mentioned, two components interact to grant the coprocessors the possibility of generating virtual memory accesses: (1) the WMU as a hardware translation accelerator, and (2) the VMW manager as an OS window handler. The WMU is a platform-specific element which is ported once per reconfigurable SoC; different applications on the same platform reuse the same WMU.

WMU. Figure 3 shows how the IDEA coprocessor is interconnected to the WMU. The standard interface consists of virtual address lines (`CP_VADDR`), data lines (`CP_DIN` and `CP_DOUT`), and control lines (`CP_CONTROL`). Control signals between the coprocessor and the WMU are the following: (1) `CP_START`, the coprocessor start signal, issued by the WMU once a user initiates the execution; (2) `CP_ACCESS`, the coprocessor access signal, indicates that there is an access currently performed by the coprocessor; (3) `CP_WR`, the coprocessor write signal, indicates that the access is a write; (4) `CP_TLBHIT`, the translation hit signal, indicates that an address translation is successful—in order to proceed with a memory access, the coprocessor should first wait for this signal to appear; (5) `CP_FIN`, the coprocessor completion signal, indicates to the WMU that the coprocessor has finished its operation. Platform-specific signals (e.g., the window RAM physical address lines `DP_PADDR`, data lines `DP_DIN` and `DP_DOUT`, and control lines `DP_EN`, `DP_WR`) connect the WMU with the rest of the system—and they are specific for different platforms. Inside the WMU, there are the three registers accessible by the main processor (`AR`, `SR`, and `CR`) and the *Translation Lookaside Buffer (TLB)* which emphasises the similarity of the WMU with a conventional MMU [12]. Apart from typical status and control registers (`SR` and `CR`), the address register (`AR`) is examined by the OS, in order to determine which memory access caused an access fault. The main processor accesses the WMU using the separate control and data lines and it is informed about access faults by the `WMU_INT` interruption line.

It is evident that the TLB is the most critical component of the WMU design, whose efficiency is essential for the low-overhead operation of the coprocessor: it should quickly perform the translation from virtual addresses to the physical ones, and at the same time expose management interface to the OS. The design of the TLB is specific to the underlying reconfigurable technology and is made easier by the fact that most modern FPGA families offer embedded *Content Addressable Memory (CAM)* and *Random Access Memory (RAM)* components [1, 28]. More specifically, CAM and RAM memories are used to form the TLB lines [12] that contain virtual page numbers, corresponding physical page numbers, and validity and dirtiness bits. The

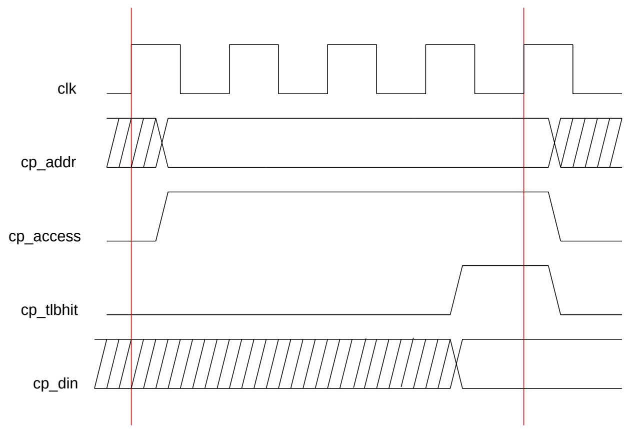

Figure 4. The coprocessor read access. Data is ready on the fourth rising edge of the clock.

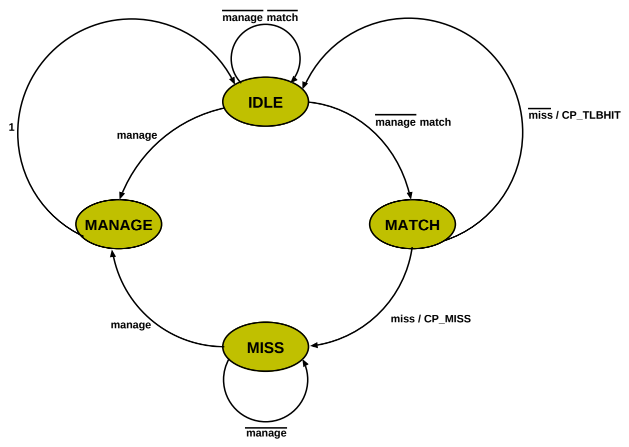
$\overline{manage}$ $\overline{match}$
1
manage
IDLE
$\overline{manage}$ match
$\overline{miss}$ / CP_TLBHIT
MANAGE
MATCH
manage
miss / CP_MISS
MISS
$\overline{manage}$

Figure 5. TLB state machine.

use of the intrinsically slow FPGA technology dictates performing the translation in multiple cycles: for the current TLB design, if we assume no translation faults, four cycles are needed from the moment when the coprocessor generates an access to the moment when the data is read or written. The performance drop caused by multiple translation cycles can be overcome by pipelining. The timing diagram for the current TLB design is shown in Figure 4.

Figure 5 shows the simplified state-transition diagram of the TLB state machine. Its states are: (1) `IDLE`—initial state before any of the actions is undertaken, (2) `MANAGE`—management state that indicates a main processor access in order to examine and change the contents of the TLB, (3) `MATCH`—matching state that indicates a pending coprocessor access and translation request, and (4) `MISS`—the page-fault state indicating the occurrence of a translation miss. The input signals are: (1) `manage`—manage request coming from the main processor, (2) `match`—translation request coming from the coprocessor (by raising the signal `CP_ACCESS`), and (3) `miss`—miss indicator coming from the CAM. The output control signals are: (1) `CP_TLBHIT`—successful translation indicator sent to the coprocessor, and (2) `CP_MISS`—unsuccessful translation indicator used to set the appropriate miss bit in the status register and raise the OS service-request interrupt.

At the beginning, the main processor uses management accesses in order to initialise the TLB. Once the TLB is initialised, the coprocessor enters normal operation (represented by sequences of transitions between `IDLE` and `MATCH` states). If a miss happens, the transition to `MISS` state appears, the TLB waits for the OS management, and the coprocessor is stalled waiting on the `CP_TLBHIT` signal. After a sequence of management accesses, the interrupt cause should be resolved—the TLB lines filled with the correct translation data and the contents of the window memory made to reflect the appropriate parts of the user memory space—and translation process resumed, now with the correct translation being performed. In this way, the coprocessor is designed completely agnostic of specific data addresses. Memory resources between the main processor and the coprocessor are allocated dynamically and the programmer can have a transparent access to the accelerator.

3.2 OS Support: The VMW OS Extension

The VMW manager provides two functionalities: (1) a system call to access and control the coprocessor, and (2) management functions to respond to WMU requests. The system call is called `FPGA_EXECUTE`. It passes data pointers and parameters to the hardware, initialises the WMU, launches the coprocessor, and puts the calling process in sleep mode. The software designer passes to the coprocessor references to objects and their sizes as they are; the coprocessor processes the objects with no concerns about their location in memory—translation of generated addresses and memory allocation is done by WMU and VMW manager.

The window memory is logically organised in pages, as in typical virtual memory systems. Objects accessed by the coprocessor are mapped to these pages. The OS keeps track of the occupied pages and the corresponding objects. The window manager responds to the WMU requests. The OS determines the cause of the interrupt by examining the status register of the WMU. There are two possible requests: (1) *page fault*—the coprocessor attempted an access of an address not currently in the window memory, and (2) *end of operation*—the coprocessor signals the end of operation to the main processor and the manager then ensures that the user memory reflects correctly the state of the window memory.

Besides window management, the OS provides also a parameter passing protocol. Once its operation is started, the

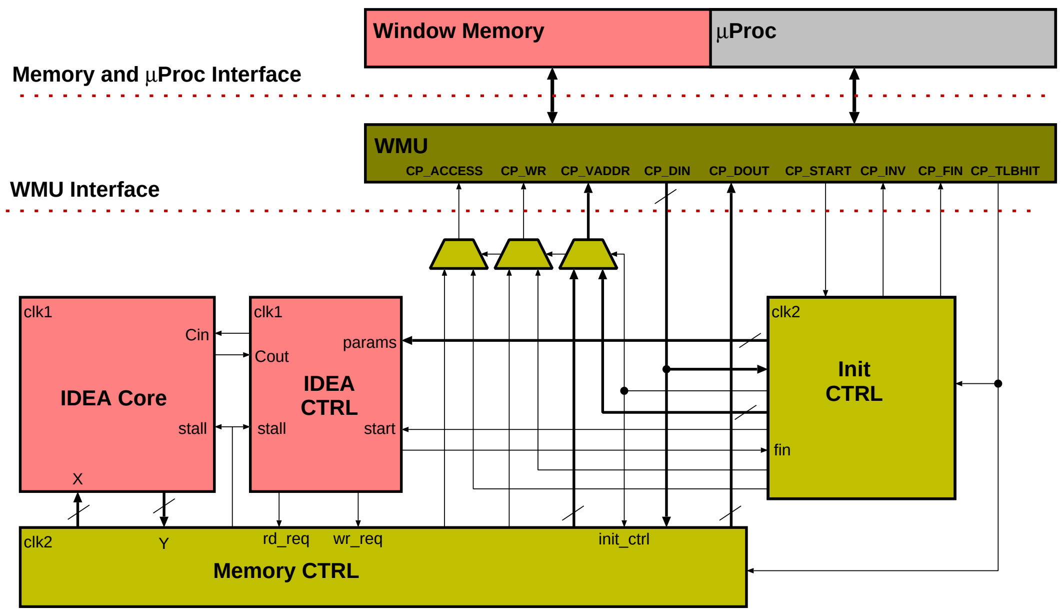

Figure 6. Block diagram of the IDEA coprocessor using a VMW.

coprocessor looks for parameters in a window memory page initially devoted to parameter passing. When the parameters are read, the coprocessor finishes initialisation and continues with normal operation. At the same time it invalidates the parameter-passing page, in this way making it available for future data mapping purposes.

3.3 IDEA Coprocessor Design

The reference IDEA implementation [19] has been used as the basis of the IDEA coprocessor design together with a previous FPGA design in synthesisable VHDL [4]. The IDEA encryption algorithm consists of eight rounds of the core transformation followed by the output transformation. When designing the coprocessor, the eight rounds can be "unrolled" certain number of times, depending on the available reconfigurable logic resources. The computation of a round contains four multipliers (rather resource-demanding in the case of the FPGA implementation), four adders, and several XOR gates. The original pipeline depth of the IDEA round implementation [4] is five stages. The output transformation implementation is only one stage, containing two multipliers and two adders. To obtain a less resource demanding design, the available design of the IDEA core was changed. The multipliers and adders in the IDEA round pipeline are used in a time-multiplexed manner to lower their number (two multipliers and two adders).

In order to adapt the design of the IDEA coprocessor to the VMW interface, a memory access unit and an initialisation unit are designed to comply to the VMW interface specification. Figure 6 shows the four principal design blocks of the IDEA coprocessor. The IDEA Core and the IDEA CTRL blocks represent the algorithm core and its controller. The Memory CTRL and Init CTRL blocks provide actual communication to the core through the virtualisation interface (connection to the WMU). In order to avoid performance penalties due to the relatively complex core design compared to the interface components, the core blocks belong to a different clock domain (`clk1`'s period is an integral fraction of `clk2`'s). The synchronisation mechanism is provided using the stall signal to stall the core until Memory CTRL is ready to read/write the data. Once the coprocessor is started (`CP_START`), control is taken by Init CTRL. This block generates addresses complying to the parameter passing protocol. It reads initialisation parameters and passes them to the IDEA CTRL block (`params`). After all the parameters are passed, it invalidates the parameter passing page (`CP_INV`) making it available for user-space data mapping and starts the IDEA CTRL (`start`). IDEA CTRL controls the computation of IDEA Core and generates memory requests to Memory CTRL (`rd_req`, `wr_req`). Memory CTRL in its turn stalls (`stall`) the core unless it is ready to respond to the requests. For each request, it generates the appropriate WMU interface signals (`CP_ACCESS`, `CP_WR`, `CP_VADDR`, `CP_DOUT`), waits for the TLB hit acknowledgment (`CP_TLBHIT`), and eventually reads the input data lines (`CP_DIN`). When the computation is finished, IDEA CTRL passes the control back to INIT CTRL (`fin`), which informs the WMU about the successful completion of the coprocessor computation (`CP_FIN`).

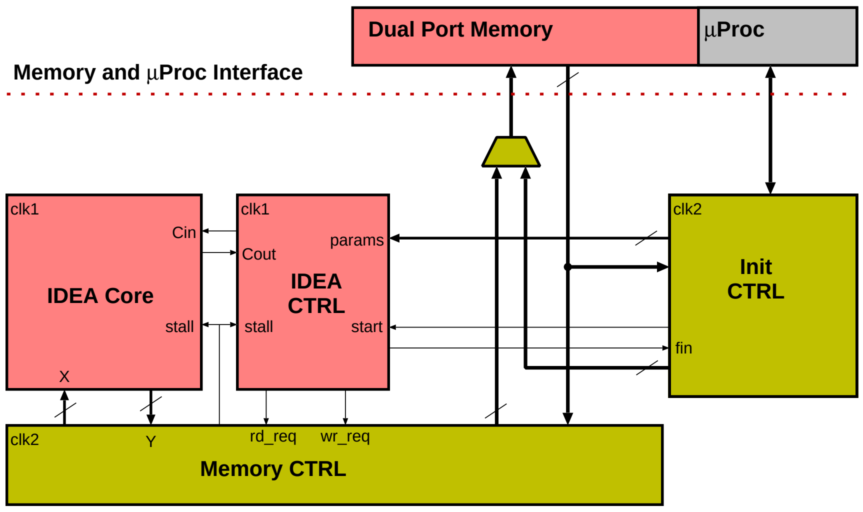

Figure 7. Block diagram of the IDEA coprocessor without VMW.

It can be noticed that the use of a VMW does not add to the complexity of the coprocessor hardware: Figure 7 shows the design blocks of the IDEA coprocessor without VMW. One can notice that the IDEA-specific blocks are unmodified and, although slightly different, both Memory CTRL and Init CTRL must be present in one form or another—the main difference is that they now generate *physical addresses* of the dual port memory. It is not even necessary that generated addresses correspond to the dual port memory addresses used by the processor: the memory accesses can be anywhere in the system memory map. However, having the accesses go through the VMW, memory protection policies can be easily achieved (e.g., preventing the coprocessor to access forbidden memory regions).

On the software side, the IDEA application code is changed in order to use the OS system call for launching the coprocessor. For the purpose of comparison, code excerpts (a) of the original software version, (b) of what could be a typical coprocessor version, and (c) of the VMW-based version are contrasted in Figure 8. Note that the typical coprocessor version requires relatively straightforward but burdensome and system-specific address calculation. On the other side, the coprocessor call in the VMW-based version is almost *as simple and elegant as the original function call*: just before the actual call, an array of the `param` data structure is used to pass parameters and data pointers to the coprocessor. It is initialised with the following convention: the first element of the array is used to pass the number of parameters and arbitrary flags to the coprocessor; the rest of the array elements are data pointers and the pointed data sizes. The data is copied dynamically to/from the window memory as requested/produced by the coprocessor. The task is done by the OS module and is completely hidden from the software programmer.

4 Experimental Results

In this section, we describe an implementation of our VMW system. Later on, we present the results obtained for the IDEA application that is ported to the system.

4.1 Setup

A VMW system has been implemented using a board based on the Altera Excalibur EPXA1 device [1]. The device consists of a fixed part, called *ARM-stripe*, and of reconfigurable logic, called *PLD*. The ARM-stripe includes an ARM processor running at 133MHz, peripherals, and on-chip memories. The board is equipped with 64MB of SDRAM and 4MB of FLASH, and runs the GNU/Linux OS.

The WMU is designed in VHDL as a reusable library element to be synthesised together with a coprocessor. The TLB, the most critical part of the WMU, is implemented using CAMs and RAMs available in the PLD part of the EPXA1 device. As mentioned before, due to the limitations of the technology, the translation is performed in multiple cycles. Note that, although we had to implement the WMU in FPGA for these experiments, the WMUs could and should, in principle, become standard components implemented on the ASIC platform in the same way as the ARM MMU is implemented in the ARM-stripe today. At present, four cycles are needed from the moment when the coprocessor generates an access to the moment when the data is read or written. The performance impact of multiple translation cycles was minimised thanks to the pipelining of the IDEA coprocessor and to the use of different clock periods between the translation hardware and the application.

```
/* Pure software version */
...
idea_cipher_fun(A, B, n64);
...
```

(a)

```
/* Typical coprocessor version */
...
data_chunk = DP_SIZE / 2; data_pt = 0;
while (data_pt < n64 * 8) {
copy(A + data_pt, DP_BASE, data_chunk);
idea_cipher_coprocessor();
copy(DP_BASE + data_chunk, B + data_pt, data_chunk);
data_pt += data_chunk;
}
...
```

(b)

```
/* VMW-based coprocessor version */
...
param[0].u.params_no = 3;
param[0].v.flags = 0;
param[1].u.address = A;
param[1].v.size = n64 * 8;
param[2].u.address = B;
param[2].v.size = n64 * 8;
idea_cipher_coprocessor(param);
...
```

(c)

Figure 8. Different invocations of the IDEA function: the VMW-based coprocessor version is to all practical purposes identical to the pure software version and fully system-detail agnostic.

Through the WMU, the coprocessor is interfaced with the dual-port RAM memory (acting as the window memory) which is an on-chip memory accessible by both the PLD (directly) and the main processor (through an *AMBA Advanced High-performance Bus—AHB*). The dual-port memory has been logically organised in eight 2KB pages – the total size is therefore of 16KB. The dual-port memory has been chosen for VMW because of direct and easy interfacing with the PLD. Larger, single-port memories with arbitrated access would also be usable to implement the window memory.

The VMW is implemented as a Linux kernel module aware of the hardware characteristics of the particular system. Using the module on other similar systems with different size of the dual-port memory (e.g., the Altera devices EPXA4 and EPXA10) would require only changing some constants and recompiling the module. The user application would immediately benefit from a larger window without any need of modifications or recompilation.

4.2 Measurements

The IDEA coprocessor is synthesised from the VHDL code based on the design described in Section 3. The synthesis of the IDEA Core and IDEA CTRL blocks, due to the complex operators involved, results in a maximal running frequency of 6MHz. The Memory CTRL and INIT CTRL blocks, together with the WMU, are running at the frequency of 24MHz, thus four times the frequency of the core. The original C code was manually modified to make use of the OS service provided by the VMW manager and described in Section 3.

Figure 9 shows execution times of the IDEA application, for different input data sizes. The results are shown for pure software, typical coprocessor (with no WMU nor OS support), and VMW-based versions of the benchmark. Pure software and VMW-based versions are both running on top of the OS, whereas the typical coprocessor does not even have the OS. One can notice that the IDEA coprocessor achieves significant speed up comparing to the software case. As previously indicated, exploiting IDEA's parallelism in hardware was limited by the finite FPGA resources of the device used. With larger FPGA, additional speed up could be obtained. What really matters here are the differences between coprocessors with and without VMW.

For the VMW-based version, three components of the execution time are measured: (1) hardware execution time (time spent in the coprocessor and in the WMU, required for computation, memory accesses, and virtual memory translations), (2) software execution time for the window memory management (i.e., time spent in the OS transferring data from/to user-space memory; spending this amount of time is also necessary in the typical coprocessor case), and (3) software execution time for the WMU management (i.e., time spent in the OS checking which address has generated the fault, selecting a page for eviction and updating the translation table). It should be noticed that in the case of the VMW-based versions, as the data set size grows up and page misses appear (from 4KB onwards), more time is spent in the OS but the speed up is only moderately affected. It is important to stress that all of the experiments are performed by simply changing the input data size, without need of modifying neither the IDEA code, nor the IDEA coprocessor design. In particular, no modifications are needed *even for datasets which cannot be stored at once in the physically available dual-port memory*. Programming is made easier (both in C and VHDL) because no explicit reference to the dual-port memory and its physical characteristics are required.

A few conclusions can be drawn from Figure 9. First, the presence of our virtualisation layer adds portability benefits and still provides significant advantage over the pure software version (even if the difference of running frequencies for the ARM processor and the PLD is not negligible). Second, the introduced overhead can be considered acceptable: the software execution time for WMU management can be seen in the figure and it is between 5–7% of the total execution time. The hardware execution time includes the overhead of address translation. This overhead is not always negligible (around 20%) but it is only due to our FPGA implementation of the WMU—to make the overhead negligible, one should seriously consider that the WMU becomes a

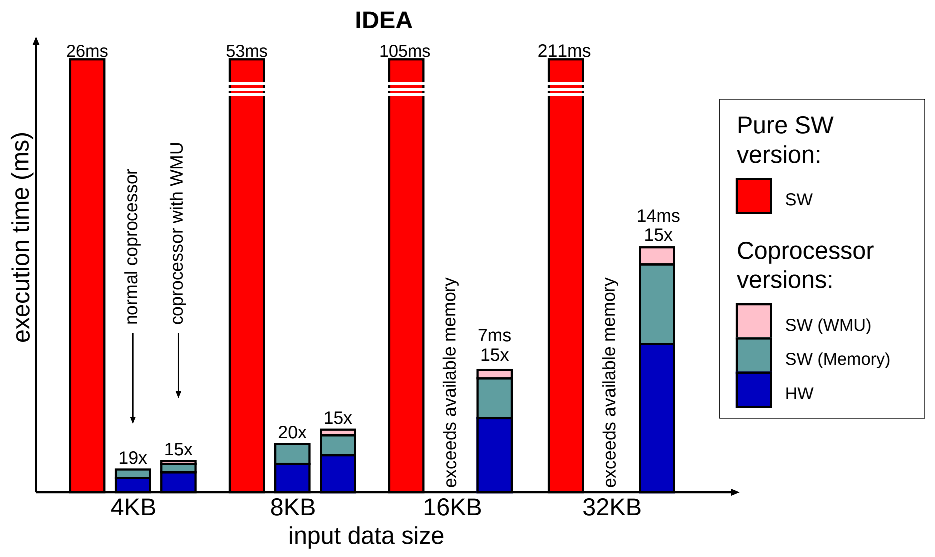

Figure 9. Measurements on the `IDEA` kernel.

standard VLSI part present on a SoC (exactly as the MMU which is already present on the chip we use). Designing a pipelined WMU could mask almost completely the translation overhead. The largest fraction of overhead is actually due to managing the window memory: Note that part of this overhead component consists of compulsory page misses and would be unavoidable even if no virtualisation was applied. Real page misses can be improved by smarter memory allocation and prefetching techniques—the latter allowing overlapping of processor and coprocessor execution.

If the same experiments were to be performed on a different platform this would require porting the WMU HW and the VMW SW, *but would not require any changes to the IDEA coprocessor HDL description nor to the IDEA application C code*.

5 Related Work

The design of cryptography coprocessors has been widely accepted as a way of boosting the execution of cryptography applications. More specifically, large number of IDEA coprocessor designes have been reported in the past [6, 25, 24, 4], intended for both FPGA and ASIC technologies. All of these designs are made having in mind the maximal speed up, while the interfacing complexity and ease of programming are not the primary concern. On the contrary, we present the adaptation of an existing design that is easily interfaced to hardware and software parts of the rest of the system.

Memory abstraction and communication interfaces definition are active field of research, motivated by IP-reuse and component-based system design. Many standardisation efforts are made in order to facilitate IP interconnection—e.g., standardised buses [2]. Another industry standard [16] provides a bus abstraction which makes the details of the underlying interface transparent to the designer. Some authors show ways of automatically generating memory wrappers and interfacing IP designs [9]. Others automate the interconnection of IP designs to a wide variety of interface architectures [14]. The main originality of our idea, with respect to these works, is not in the specific standardisation and abstraction of the memory interface details (signals, protocols, etc.) between generic producers and consumers, but in the dynamic allocation of the interfacing memory, buffer, or communication ports between a processor and a coprocessor—that is in the implication of the OS in the process.

Similarly, extensive literature exists on the design and allocation of application-specific memory systems, typically for ASIC design (e.g., [7, 22]). Mostly, these are compiler-based static techniques consisting in software transformations to exploit better a given memory hierarchy, and in design methodologies for customising the ASIC memory hierarchy itself for specific applications. The former techniques can be used proficiently to enhance the design of coprocessor such as those addressed here, but are rather independent from the actual interface details we handle. On the other hand, a few works have a dynamic flavour and could therefore be used to improve the interface memory allocator—they are fully complementary to the present techniques [15]. In the area of memory systems for reconfigurable systems, works such as [13] study the generation of optimal access patterns for coprocessors within SoC architectures; the focus is not in portability and abstraction from architectural details, as in this paper. We use simple access patterns for validation, but any access pattern could be used in conjunction with the WMU and their address generation techniques are complementary to our work.

Closer to our concerns is a different form of hardware virtualisation which has received some attention recently. With motivations similar to ours, researchers have considered the OS support required for managing the reconfigurable lattice across tasks [27]—to screen the user from the problems introduced by the finite amount of available reconfigurable logic. Reconfigurable hardware virtualisation has also been addressed [5, 8]. For example, an architecture has been introduced [8] that allows the OS to share dynamically the reconfigurable logic between applications. The resource is virtualised and hardware support is developed in order to support the mapping between the virtual and the physical resource. The type of virtualisation we introduce addresses the processor/lattice interface logic rather than the reconfigurable lattice itself; the two problems are therefore orthogonal and complementary—future system may have to implement solutions for both. Several approaches exist that introduce OS extensions supporting user-controlled interfacing of reconfigurable hardware (e.g., a task communication scheme based on message passing [21], or communication through memory slot interface [17]). While these approaches expose the communication to the programmer, our approach completely deliberates the programmer of communication details. It is the ultimate role of the OS module to provide this transparency.

6 Conclusions

In this paper we present an IDEA coprocessor designed for the *Virtual Memory Window* system (VMW). The VMW allows the reconfigurable application-specific coprocessor to access virtual addresses of the user-space memory. In this way, the IDEA coprocessor designed in synthesisable VHDL is perfectly portable accross different platforms. Furthermore, the software part of the application is implemented using a straightforward programming paradigm that hides all unnecessary details and data transfers from the programmer. Instead, the coprocessor data is allocated dynamically and copied to/from the coprocessor-accessible memory by an OS module.

We intended to show on a realistic application that a minimal performance penalty is paid for this comfort, much in the same way as nobody views nowadays a virtual memory system as a significant source of performance loss. Similarly, as virtual memory systems make it possible to achieve ease of programming and portability for software, a VMW achieves the same for the hardware description of a coprocessor and its software invocation. To validate the approach, the IDEA coprocessor was implemented on a real VMW-based system (Linux extended with a VMW module running on an ARM-based board surrounded by FPGA). We have shown that the virtualisation layer overhead is acceptable and does not compromise the benefits of the spatial execution in the coprocessor: a remarkable speed up compared to software-only version of the IDEA application is achieved, with only minimal changes in the application code.

We believe that the presented approach not only constitutes an efficient way of designing portable hardware and of writing the corresponding software in a straightforward manner; it also brings reconfigurable computing closer to general-purpose computing. We suggest a way to the increased programmability of reconfigurable systems by accepting and adapting concepts of the general-purpose systems: the concept of the OS involvement in the virtual memory management motivated the conception of the virtual memory window for reconfigurable coprocessors.

The overhead due to the translation and management of multiple data copies, albeit small, partially hides the inherent speed up of IDEA, as achievable by specialised hardware execution. This suggests that future research should address lowering these overheads in order to further expose the speed-up potentials: we are addressing speculative prefetching by the VMW manager to parallelise memory transfers with the coprocessor operation.

7 Acknowledgements

The authors would like to acknowledge and appreciate the immeasurable help and support provided by Ludovic Righetti, Cédric Gaudin, and Jean-Luc Beuchat.

References

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

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

[3] J.-L. Beuchat. *Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001. Thèse No 2426.

[4] J.-L. Beuchat. Modular multiplication for FPGA implementation of the IDEA block cipher. In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 412–422. IEEE Computer Society, June 2003.

[5] E. Caspi, M. Chu, R. Huang, J. Yeh, A. DeHon, and J. Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction snd tutorial. In *Proceedings of the 10th International on Field-*

Programmable Logic and Applications, Villach, Austria, Aug. 2000.

[6] E. Caspi and N. Weaver. IDEA as a benchmark for reconfigurable computing. Technical report, UC at Berkeley, BRASS Research Group, Berkeley, CA, Dec. 1996. `http://www.cs.berkeley.edu/projects/brass`.

[7] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecapelle. *Custom Memory Management Methodology*. Kluwer Academic, Boston, Mass., 1998.

[8] 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.

[9] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya. Automatic generation of embedded memory wrapper for multiprocessor SoC. In *Proceedings of the 39th Design Automation Conference*, New Orleans, La., June 2002.

[10] 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.

[11] 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.

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

[13] M. Herz, R. Hartenstein, M. Miranda, E. Brockmeyer, and F. Catthoor. Memory addressing organisation for stream-based reconfigurable computing. In *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems*, Dubrovnik, Croatia, Sept. 2002.

[14] T.-L. Lee and N. W. Bergmann. An interface methodology for retargetable FPGA peripherals. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nev., June 2003.

[15] M. Leeman, D. Atienza, C. Ykman, F. Catthoor, J. M. Mendias, and G. Deconcinck. Methodology for refinement and optimization of dynamic memory management for embedded systems in multimedia applications. In *IEEE Workshop on Signal Processing Systems*, Seoul, Korea, Aug. 2003.

[16] C. K. Lennard, P. Schaumont, G. De Jong, A. Haverinen, and P. Hardee. Standards for system-level design: Practical reality or solution in search of a question? In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 576–583, Paris, Mar. 2000.

[17] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwow, M. Wong, and K. Lee. Pilchard—a reconfigurable computing platform with memory slot interface. In *Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 2001.

[18] H. Lipmaa. IDEA: A cipher for multimedia architectures? In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography*, volume 1556, pages 248–263, Kingston, Ontario, Canada, Aug. 1998. Springer-Verlag.

[19] MediaCrypt. *IDEA Algorithm*, 2003. `http://www.mediacrypt.com/`.

[20] O. Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 2002.

[21] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*, Paris, June 2003.

[22] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip*. Kluwer Academic, Boston, Mass., 1999.

[23] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–80, San Jose, Calif., Nov. 1994.

[24] S. L. C. Salomao, V. C. Alves, and E. M. C. Filho. HiPCrypto: A high-performance VLSI cryptographic chip. In *Proceedings of the Eleventh Annual IEEE International ASIC Conference*, pages 7–11, Rochester, New York, USA, Sept. 1998.

[25] R. R. Taylor and S. C. Goldstein. A high-performance flexible architecture for cryptography. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–245, Aug. 1999.

[26] M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41th Design Automation Conference*, San Diego, CA, June 2004.

[27] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2003.

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