Virtual Memory Window for Application-Specific Reconfigurable Coprocessors

Miljan Vuletić
Miljan.Vuletic@epfl.ch

Laura Pozzi
Laura.Pozzi@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Swiss Federal Institute of Technology Lausanne
Processor Architecture Laboratory
EPFL I&C LAP, IN-F Ecublens, 1015 Lausanne, Switzerland

ABSTRACT

Reconfigurable *Systems-on-Chip* (SoCs) on the market consist of full-fledged processors and large *Field-Programmable Gate-Arrays* (FPGAs). The latter can be used to implement the system glue logic, various peripherals, and application-specific coprocessors. Using FPGAs for application-specific coprocessors has certain speedup potentials, but it is less present in practice because of the complexity of interfacing the software application with the coprocessor. Another obstacle is the lack of portability across different systems. In this work, we present a virtualisation layer consisting of an operating-system extension and a hardware component. It lowers the complexity of interfacing and increases portability potentials, while it also allows the coprocessor to access the user virtual memory through a virtual memory window. The burden of moving data between processor and coprocessor is shifted from the programmer to the operating system. Since the virtualisation layer components hide physical details of the system, user designed hardware and software become perfectly portable. A reconfigurable SoC running Linux is used to prove the viability of the concept. Two applications are ported to the system for testing the approach, with their critical functions mapped to the specific coprocessors. We show a significant speedup compared to the software versions, while limited penalty is paid for virtualisation.

Categories and Subject Descriptors: C.0 [General]: Hardware/software interfaces

General Terms: Performance, Design, Management.

Keywords: Reconfigurable Computing, OS, Coprocessors, Codesign.

1. INTRODUCTION

Reconfigurable arrays might increase their relevance in future deep sub-micron technologies. This is due to increasing mask costs and the consequent need of designing individual *Application-Specific Integrated Circuits* (ASICs) as adaptations of generic platforms. However, in the foreseeable future, FPGAs will not be able to show speed or area efficiency comparable to general processors implemented in ASICs. Therefore, the bulk of computation in future high-performance SoCs will have to be performed by blending the two paradigms—standard processors augmented with reconfigurable application-specific parts [6, 7, 15]. Major vendors of reconfigurable devices now offer systems consisting of processor cores surrounded by peripherals, on-chip memories, and large amounts of reconfigurable logic [1, 17] which may include special features such as embedded memories and arithmetic blocks suited for signal processing (e.g., Stratix family [1]).

While partitioning applications (between hardware and software) for such devices, designers need to interface the application-specific coprocessor and account for different architectural details—bus hierarchies and protocols, shared and/or multi-ported memories, I/O ports, etc. For instance, programmers should be aware of 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 partition transfers scheduled. Unpredictable memory accesses (e.g., processing objects on the heap) significantly complicate this conceptually easy task. Besides the design complexity, changing a host platform requires heavily redesigning both the software and hardware parts.

We introduce a shallow platform-specific hardware and an *Operating System* (OS) module that reduce the burden of software programmers and hardware designers. With them, coprocessors can access the user virtual memory through a virtual memory window. User applications and coprocessors become fully platform independent with only a limited penalty. Our contribution reduces the complexity of the programming and hardware-design paradigms and improves the portability of applications for reconfigurable platforms.

This paper is organised as follows: We specify in more detail our goals in Section 2 and introduce our basic idea. In Section 3 we show how such idea can be implemented in practice. The experimental setup used to demonstrate our system and the corresponding results are presented in Section 4. In Section 5 we discuss similarities and complementarities of our work with other memory, interface, and reconfigurable-hardware virtualisation issues. Finally, conclusions are drawn in Section 6.

2. GOALS

The programmer of a computing platform running an OS is abstracted from the characteristics of the memory system [8]: he/she generates memory accesses ignoring whether the required main memory physically exists. The addresses known to the programmer are *virtual* and 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. The disadvantage is that the automatic allocation of pages by the operating system is, in general, suboptimal. In principle, an experienced programmer could obtain better results by managing directly the memory hierarchy, but in most cases people accept a small performance loss for the above-mentioned advantages.

We aim to extend these advantages to application-specific coprocessors (for instance, implemented in the FPGA on a reconfigurable SoC) working on behalf of a user-space application. Without loss of generality, we will concentrate on a virtual memory window built using a dual-port memory accessible by both the reconfigurable lattice and the processor. Our goal is to have an application (in a high-level language—C or C++) and the corresponding coprocessor (in a hardware description language—VHDL or Verilog) completely independent of the underlying hardware. An appropriately augmented OS, a compiler, and a synthesiser must be sufficient to port the accelerated application across different systems.

2.1 Virtual Memory Window

Analogously to virtual memory management, the programmer of a reconfigurable computer should design data exchanges between the processor (i.e., the application software) and the coprocessor (i.e., the reconfigurable hardware) without any knowledge of the underlying physical system. Similarly, the coprocessor designer should be exposed to the same abstraction and generate abstract addresses rather than specifying physical addresses of the directly-accessible memory. Data objects should be addressed transparently from platform limitations.

To allow this abstraction, a memory accessible directly by the FPGA is used as a coprocessor's *Virtual Memory Window* (VMW) to the virtual memory of the user application. As in the case of VMM, two elements are added to the basic system: (1) hardware device that performs the translation between the virtual addresses of abstract objects/elements and the corresponding physical addresses. We call this hardware, similar to a classic MMU, *Window Management Unit* (WMU); (2) support in the OS that allocates dynamically virtual memory regions and ensures that they are available to the coprocessor. Similarly to the VMM, a *Virtual Memory Window manager* (VMW manager) handles the translation unit and the content of the window memory. The WMU sends an interrupt to the OS when the VMW manager needs to provide data to the coprocessor through the window. Figure 1 shows how the VMW integrates to a virtual memory system. 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 (WMU) and the window memory, supported by the OS part (VMW manager) that maintains data transfers from/to the user memory.

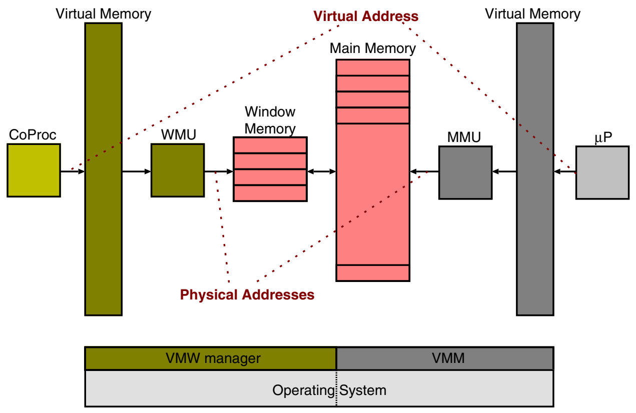

Figure 1: Virtual Memory Window for Coprocessor.

```
/* Software version */
add_vectors(A, B, C, SIZE); ...

/* Typical coprocessor version */
data_chunk = DP_SIZE / 3; data_pt = 0;
while (data_pt < SIZE) {
   copy(A + data_pt, DP_BASE, data_chunk);
   copy(B + data_pt, DP_BASE + data_chunk, data_chunk);
   add_vectors_coprocessor();
   copy(DP_BASE + 2*data_chunk, C + data_pt, data_chunk);
   data_pt += data_chunk;
} ...

/* VMW-based coprocessor version */
add_vectors_coprocessor(A, B, C, SIZE);
```

Figure 2: Motivating example

2.2 Motivating Example

Figure 2 shows simplified pseudo-code excerpts of a trivial application that invokes either a software function or a hardware coprocessor to add two vectors (A and B) and store the result into a third one (C). The application is ported to three different systems: (1) pure software, (2) typical coprocessor, and (3) VMW-based coprocessor system. In the case of the typical coprocessor version, it can be seen that the programmer needs to take care about unnecessary platform-related details (a similar task burdens the hardware designer). On the contrary, the VMW-based version completely resembles the pure software version and provides a clean and transparent interface to the coprocessor.

Besides simple interfacing, the VMW-based coprocessor system has another significant advantage: it makes use of the same virtual addresses (i.e., pointers) to access data, exactly as the user application does. Thus, the VMW-based system is capable of processing dynamically allocated data (e.g., objects scattered on the heap, linked lists) without any additional burden on the programmer's side. This feature can support use of application-specific coprocessors in object-oriented and runtime environments (e.g., Java Virtual Machine).

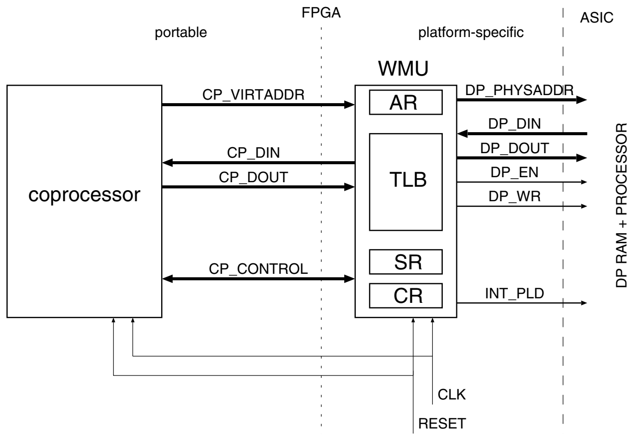

Figure 3: Coprocessor and WMU communication.

3. VIRTUALISATION COMPONENTS

We discuss here the three components that provide virtual memory abstraction to the coprocessor: (1) the standard OS service used to invoke the coprocessor, (2) the translation and interfacing hardware (WMU), and (3) the translation manager (VMW manager).

3.1 Coprocessor Invocation Service

A system call that is provided to software designers is called `FPGA_EXECUTE`. It passes data pointers and parameters, initialises the WMU, launches the coprocessor, and puts the calling process in the sleep mode (avoiding consistency problems of simultaneous accesses to multiple data copies). `FPGA_EXECUTE` also informs the OS about the objects which require dynamic allocation. The software designer passes references to objects and their sizes to the coprocessor as-they-are, without any particular preparation; the hardware designer implements a coprocessor having in mind no specific data addresses—it fetches all necessary references through a standardised initialisation protocol. Finally, the coprocessor processes the objects with no concerns about their location in memory—translation of generated addresses is done by WMU and VMW manager.

3.2 Hardware Translation

All coprocessor memory accesses pass through the WMU, which is the coprocessor's interface to the system. If possible, virtual addresses demanded by the coprocessor are translated by the WMU to real addresses of the window memory region. Otherwise, an interrupt is generated and the OS handling is requested. Although it is excluded from the virtual memory mapping, the window memory region is managed by the OS and divided into pages to allow multiple virtual mappings.

Figure 3 shows how the virtual addresses generated by a standardised coprocessor are translated by the WMU. The WMU in the figure reflects the one implemented for our real system, described in Section 4. The interface between the coprocessor and the WMU is quite simple: It consists of address lines (`CP_VIRTADDR`), data lines (`CP_DIN` and `CP_DOUT`), and control lines (`CP_CONTROL`). Platform-specific signals (the window RAM access lines `DP_PHYSADDR`, `DP_DIN`, `DP_DOUT`, `DP_EN`, `DP_WR`) travel towards the rest of the system, and they would differ across different platforms. Inside the WMU, note 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 [8]. 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 key part of the WMU is the TLB, performing address translation for coprocessor accesses. Its design is platform-specific as it reflects the organisation of the window memory region accessible by the coprocessor. As in typical VMM systems, the upper part of the coprocessor address (most significant bits) is matched to the patterns representing virtual page numbers stored in the translation table. If no match is found, the coprocessor operation is stalled and the OS management is requested. The TLB also contains invalidity and dirtiness information, like a typical MMU does [8].

Standard 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 the coprocessor initiates a memory access; (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—after initiating a memory access, the coprocessor should never continue its operation until this signal appears; (5) `CP_FIN`, the coprocessor completion signal, indicates to the WMU that the coprocessor has finished its operation.

Note that the interest here is not much in the implementation of a wrapper between two memory access protocols—one standardised and platform-independent and the other platform and memory specific; this is a well-studied topic in system-level design, as discussed in Section 5. The originality of our approach lays in the dynamic allocation of memory resources (i.e., shared or dual-port memory) between processor and coprocessor, which makes it possible for the application programmer to ignore the physical details of the resource. Such result can be achieved transparently through the involvement of the OS as discussed below.

3.3 Window Management

The memory is logically organised in pages, as in typical memory systems. Multiple operating modes (i.e., different number of pages in the window memory) are supported by the WMU. Objects accessed by the coprocessor are mapped to these pages . The OS keeps track of the occupied pages and the corresponding objects. Not necessarily all of the objects processed by the coprocessor reside in the memory at the same time. At every point in time, the memory access patterns of the coprocessor determine the occupation of the available pages.

The VMW manager responds to the WMU requests. The OS determines the cause of the interrupt by examining the state of the WMU. There are two possible requests:

Page Fault. If the WMU signals a page fault, it means that the coprocessor attempted an access of an address not currently in the window memory. The OS rearranges the current mapping to the window memory in order to resolve it. It may happen that all pages are in use and in this case a page is selected for eviction (different replacement policies are possible—first-in first-out, least recently used, random). If the page is dirty, its contents are copied back

```
cycle 1:
  CP_ADDR <= ptr_a; -- object A[]
  CP_ACCESS <= '1'; CP_WR <= '0';

cycle 2:
  reg_a <= CP_DIN;
  CP_ADDR <= ptr_b; -- object B[]
  CP_ACCESS <= '1'; CP_WR <= '0';

cycle 3:
  reg_b := CP_DIN;
  reg_c := reg_a + reg_b;
  CP_ADDR <= ptr_c; -- object C[]
  CP_DOUT <= reg_c; CP_ACCESS <= '1';
  CP_WR <= '1';
  ptr_{a,b,c} <= ptr_{a,b,c} + 1;
```

Figure 4: Example of VHDL-like coprocessor code.

to the user-space memory and the page is newly allocated for the missing data; the missing object part is copied from the user-space memory and the WMU state updated. Afterward, the OS allows the WMU to restart translation and lets the coprocessor exit from the stalled state.

End of Operation. Once the coprocessor finishes regularly its task, it signalises through the WMU the end of operation to the main processor. The window manager copies back to user space all the dirty data currently residing in the window memory. The coprocessor should be ready waiting for the next `FPGA_EXECUTE` call.

Speculative actions such as prefetching could be used in order to avoid translation faults. The manager could detect coprocessor's memory access patterns, predict its future actions, and prefetch the speculative pages. Although the interface management task is similar to a classic VMM, the application of such techniques for coprocessor interfacing is novel and brings new advantages.

3.4 Example

The coprocessor code using the WMU (in Figure 4) computes the addition of two arrays: $C[i] = A[i] + B[i]$. For simplicity, the figure omits the implementation details of the finite state machine that switches between the three cycles, and no pipelining is assumed.

It is important to note that *no physical address appears in the code*. All of the generated addresses (`ptr_{a,b,c}`) are virtual and provided through the initialisation. The WMU automatically translates this information into physical addresses, if possible, or invokes the OS, if the translation data are unavailable. This feature of the coprocessor code has several important consequences. First, no effort needs to be made by the coprocessor designer in order to perform physical address calculation—a tiresome task. More important, the software needs not be modified if the datasets to be exchanged exceed the memory available on the interface: the coprocessor can address arbitrarily large data. Finally, both the HDL and C code are now portable. The code is transparent not only to the address modality of the RAM (e.g., access rules)—as in many wrapper-based abstract interfaces such as [5]—but also to the overall memory size and allocation policy.

Figure 5 shows how the C file which originally computed $C[i] = A[i]+B[i]$ is modified to add calls to the FPGA, as described in Section 3.1. Essentially, `FPGA_EXECUTE` replaces a call `add_vectors(A, B, C, SIZE)` where nonscalar parameters are prepared and passed by reference (see Figure 2).

```
int A[]; int B[]; int C[]; ...

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);
```

Figure 5: Example of application C code.

4. DEMONSTRATION

A VMW system is 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 Linux OS.

The WMU is designed in VHDL to be synthesised together with a coprocessor. The TLB, the most critical part of the WMU, is implemented using content addressable and RAM memories available in the PLD part of the EPXA1 device. Due to the limitations of the technology, the translation is performed in multiple cycles. Although we had to implement the WMU in FPGA for these experiments, WMUs should, in principle, become standard components implemented on the ASIC platform in the same way as MMUs. Currently, 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 could be overcome by pipelining.

Through the WMU, the coprocessor is interfaced with the dual-port RAM memory, an on-chip memory accessible by both PLD (directly) and the main processor (through an AMBA Advanced High-performance Bus—AHB). Depending on the WMU operating mode, the memory is logically organised in 2–32 pages with respective page sizes 8–0.5KB (the total size is therefore 16KB). It has been chosen for the window memory because of direct and easy interfacing with PLD.

The VMW manager is implemented as a Linux kernel module for the particular system. Using the module on the system with different sizes of dual-port memory (e.g., the Altera devices EPXA4 and EPXA10) would require only recompiling the module. The user application would immediately benefit without need to recompile.

4.1 Measurements

The viability of our approach was proven on two designs: a cryptography application, IDEA (running at 6MHz), and a common multimedia benchmark, `adpcmdecode` (running at 40MHz). For both, the critical parts were implemented in VHDL as standard coprocessors using the WMU interface. 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 6 shows the execution times of the benchmarks. The IDEA results are shown for pure software, for a typical coprocessor (without OS), and for a VMW-based version of the benchmark, with different input data sizes. The complex IDEA coprocessor core runs at 6MHz and has 3 pipeline stages. The WMU and the IDEA memory subsystem run at 24MHz and the synchronisation with the IDEA core is pro-

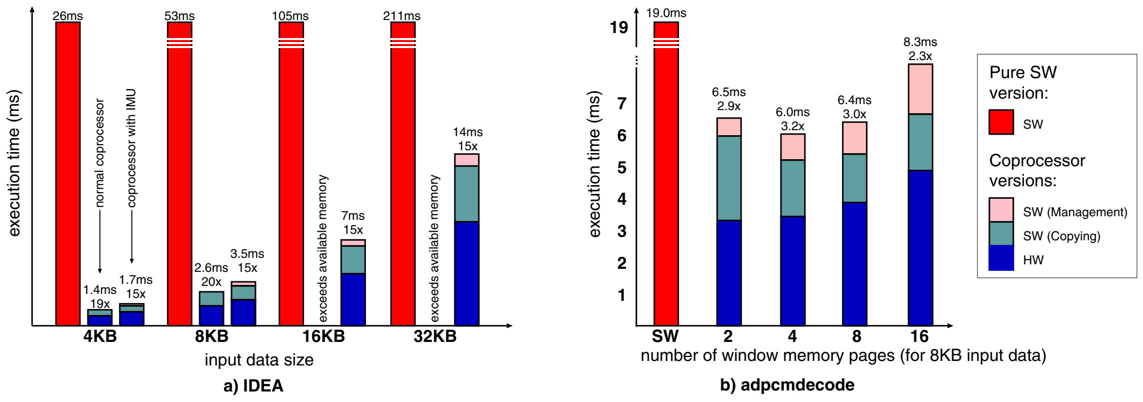

Figure 6: Measurements on `IDEA` and `adpcm` kernels.

vided by a stall mechanism. IDEA's hardware parallelism is limited by the PLD resources of the device used; with larger PLDs, additional speedup could be obtained. The results for `adpcmdecode` are shown for pure software and for VMW-based versions of the application, using different WMU operating modes. The `adpcmdecode` coprocessor and the WMU are running at the frequency of 40MHz. Both `adpcmdecode` versions are running on top of the OS. The `adpcmdecode` application produces four times more data than it consumes (e.g, one input page of data produces four output pages). Despite the incurred overhead of the VMW-based versions, both coprocessors achieve significant speedup compared to the software case (especially in the case of the IDEA benchmark).

For the VMW-based versions, 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 window memory copying—time spent in transferring data from/to user-space memory; and (3) software execution time for the WMU management—time spent in checking which address has generated the fault, selecting a page for eviction, and updating the translation table. In the VMW case, the software periods are spent in the OS module. For IDEA (VMW-based version), when the data set size grows, capacity misses appear (from 8KB onwards). Additional time is spent in the OS for the management but the speedup is only moderately affected. Programming is made easier (both in C and VHDL) because no explicit reference to the dual-port memory is required: It is important to stress that all of the experiments are performed by simply changing the input data size, without the need of modifying neither the application code, nor the coprocessor design. In particular, no modifications are needed *even for datasets which cannot be stored at once in the physically available dual-port memory.*

For the VMW-based version of `adpcmdecode`, changing the number of window memory pages (i.e., changing WMU operating modes) within a reasonable range does not dramatically affect the speedup of the coprocessor. As expected, the management time increases with the number of pages, while the copying time is almost constant (except when the window memory has only two pages, where the behaviour of `adpcmdecode` and the simple allocation policy trigger conflict misses and some additional copies are required).

A few conclusions can be drawn from Figure 6. 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 5–12% of the total execution time (for an optimal number of pages). The hardware execution time includes the overhead of address translation and the OS response time. This overhead is not always negligible (in the IDEA case around 20%) but it can be reduced: one should consider making the WMU a standard VLSI part present on a SoC (exactly as the MMU which is already present on the chip we use). Although a significant amount of time is spent in copying to/from the window memory, a considerable part of this time is contributed by compulsory page misses and would be unavoidable even if no virtualisation was applied. The number of page misses can be reduced by smarter memory allocation and prefetching techniques—the latter allowing overlapping of processor and coprocessor execution.

To conclude, one can notice that 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 coprocessor HDL description nor to the application C code.*

5. RELATED WORK

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 [12] 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 [5]. In [10], an interfacing layer is presented to automate the connection of IP designs to a wide variety of interface architectures. The main originality of our idea, with respect to these works, is not in the 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., [3, 14]). 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 [11]. In the area of memory systems for reconfigurable systems, works such as [9] 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. Although we only use simple access patterns for validation, any access pattern could be used in conjunction with the WMU. In this way, 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 [16]—the purpose is to screen the user from the problems introduced by the finite amount of available reconfigurable logic. Similarly, reconfigurable hardware virtualisation is addressed in [4], where an architecture is introduced to have the OS sharing 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 interfacing rather than the reconfigurable lattice itself; the two problems are therefore orthogonal and complementary—future system may have to implement solutions for both. Finally, in [13], an OS for reconfigurable platforms is proposed that suggests a task communication scheme based on message passing. It exposes the communication to the programmer and it differs from our approach.

6. CONCLUSIONS

In this paper we add a Virtual Memory Window for virtual memory accesses to a reconfigurable computing platform. It provides a straightforward programming paradigm and makes reconfigurable applications completely portable.

The idea is not to improve the performance of the reconfigurable system; as in most related computer architecture ideas such as virtual memory management, the goal is to pay a minimal performance tag for the ease of programming and portability advantages. To quantify the overall benefits, we have tested the approach on a real system equipped with an operating system; we ran a simple multimedia application and a complex cryptographic algorithm, both enhanced with application-specific coprocessors of different complexity. The overhead incurred due to the presence of the virtualisation layer is generally limited, but we are aiming to reduce it further. In both cases the coprocessors achieve a significant speedup compared to software-only execution, with minimal changes in the application code.

We believe that this first step is a key issue for the future of reconfigurable computing. It helps bringing reconfigurable hardware up to the programming paradigm of general computing—a goal which can be most easily achieved by involving the OS. After lowering the overhead of the translation process, research should address the development of efficient allocation algorithms in the OS. The goal is to expose almost completely the inherent speed-up achievable by specialised hardware execution without the inherent complexity of dealing with the details of the physical components.

7. REFERENCES

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

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

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

[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] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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