Operating System Support for Interface Virtualisation of Reconfigurable Coprocessors

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

ABSTRACT

Reconfigurable *Systems-on-Chip* (SoC) on the market consists 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 speed up potentials, but it is less frequent 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 provides portability potentials. The virtualisation layer shifts the burden of moving data between processor and coprocessor from the programmer to the operating system. The operating system relies on a specially designed hardware that interfaces a coprocessor to the rest of the system. In this way, both the coprocessor hardware and the software are made completely independent of the physical details of the system, and thus perfectly portable. A reconfigurable SoC running Linux is used to prove the viability of the concept. In order to test the approach, two applications are ported to the system with their critical functions mapped to the coprocessor hardware. We show that a significant speed up is obtained compared to the software versions, while limited penalty is paid for virtualisation.

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. 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 pure software and hardware-accelerated tasks for such devices, designers need to interface the application-specific coprocessor with the rest of the system, taking into account the peculiarities of the interface between the processor and the FPGA—bus hierarchies and protocols, shared and/or multi-ported memories, I/O ports, etc. For instance, programmers need to take explicitly into account the availability and size of shared memory between processor and FPGA; if such memory is smaller than the datasets to be exchanged, the datasets need to be partitioned and a schedule for loading them onto the shared memory developed. In some cases this can be burdensome but conceptually easy—e.g., in streaming applications it would require loading a fragment of the input stream on the shared memory, calling the coprocessor, copying back the new fragment of output stream, and repeating the process until finished. Other cases with more unpredictable accesses are much more difficult to manage. On top of the design complexity, changing a host platform would require redesigning to a significant degree both the software and hardware parts to fit the new device interface mechanism.

Our contribution reduces the complexity of the programming and hardware design paradigms and improves the portability of applications for reconfigurable-computing platforms: it shows how a shallow platform-specific hardware layer and some cooperation from the Operating System (OS) can reduce the burden of the programmers and designers and make user applications and coprocessors fully platform independent with a limited performance penalty.

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 equipped with 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 therefore *virtual* and they describe an extremely large memory system with

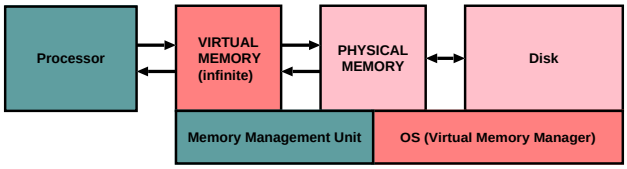

Figure 1: Virtual memory system.

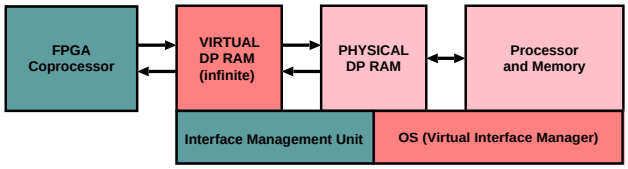

Figure 2: Virtualised interface.

no relation to the real one. As shown in Figure 1, 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.

Our goal is to extend these advantages to the interface between the processor and a coprocessor, for instance implemented in FPGA on a reconfigurable SoC. Without loss of generality, we will concentrate on an interface built around 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 (e.g., C or C++) and the corresponding coprocessor(s) in a hardware description language (e.g., VHDL or Verilog) completely independent of the target hardware. An appropriately augmented OS, a compiler, and a synthesiser must be sufficient to port the accelerated application across different systems.

2.1 Virtual Interface Management

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 physical system. A typical approach would assume the use of a dual-port or a shared memory, but the programmer should ignore its size and location in the system memory map. Similarly, the coprocessor designer should be exposed to the same abstraction and generate abstract addresses rather than generating specific physical addresses for particular memory banks available on a given platform. Data items such as vectors, for example, should be addressed with respect to their natural size, independently of platform limitations.

As in the case of VMM, two elements should be added to the basic system: (1) A hardware device that performs the translation between the addresses of abstract object/elements and the corresponding physical addresses. We call this hardware *Interface Management Unit* (IMU) and it has a strong similarity to a classic MMU. (2) A support in the OS that allocates dynamically interface objects and ensures that they are available to the coprocessor when required. As the VMM does, a *Virtual Interface Manager* (VIM) handles the translation unit and the content of the interface memory. The IMU sends an interrupt to the OS when the VIM needs to provide data to the coprocessor through the interface. Figure 2 shows a practical instance of interface virtualisation where the coprocessor reads data from a dual-port memory (DP RAM, accessible by the main processor too).

```
/* 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;
}
...

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

Figure 3: Motivating example

2.2 Motivating Example

Figure 3 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 the third one (C). The application is ported to three different systems: (1) pure software, (2) typical coprocessor, and (3) VIM-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 VIM-based version completely resembles the pure software version and provides a clean and transparent interface to the coprocessor.

3. COMPONENTS FOR INTERFACE VIRTUALISATION

We discuss here the three components required for interface virtualisation: (1) the standard services used to invoke the coprocessor, (2) the hardware for interfacing a coprocessor to the processor, and (3) the manager of such interface. The first and the last items are software extensions to the OS, while the second is a hardware component.

3.1 OS Coprocessor Invocation Services

Three system calls are provided to software designers. First, `FPGA_LOAD` loads a coprocessor definition in the reconfigurable hardware and ensures the exclusive use of the resource. The argument of the call is a pointer to the configuration bit-stream. Second, `FPGA_MAP_OBJECT` allocates the data used by the coprocessor. The arguments of the call are: (a) the object identifier (a number agreed by the hard-

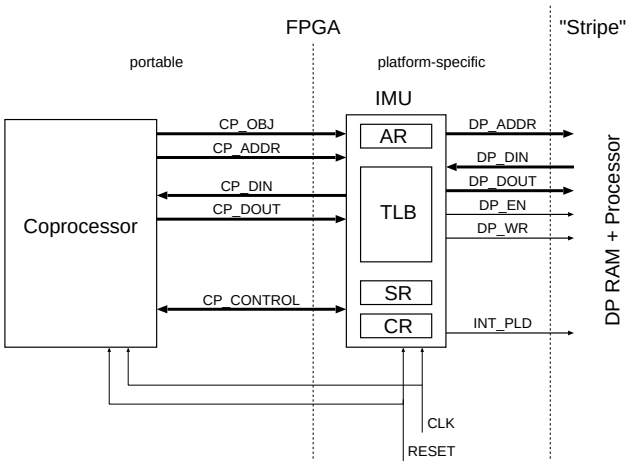

Figure 4: Communication between coprocessor and IMU.

ware and software designers), (b) a pointer to the data, (c) the data size, and optionally (d) some flags used for optimisation purposes. Finally, `FPGA_EXECUTE` performs the mapping, passes scalar parameters, initialises the IMU, launches the coprocessor, and puts the calling process in an interruptible sleep mode.

`FPGA_LOAD` and `FPGA_EXECUTE` are part of any processor or coprocessor handshaking system and exist, in one form or another, in any SoC of this sort. `FPGA_MAP_OBJECT` is specific to our scheme and informs the OS of the data for which it will have to provide dynamic allocation. In a sense, they are equivalent to software parameter passing by reference. In this way, an arrangement between a software and hardware designer is actually made with a call to this service: The software designer declares the data to be processed by the coprocessor; the hardware designer implements a coprocessor having in mind the programmer-declared data (i.e., mapped objects). The coprocessor can processes the mapped data with no concerns about their location in memory.

3.2 Hardware Interface

All coprocessor memory accesses pass through the IMU, which is the coprocessor interface to the system. If possible, virtual addresses demanded by the coprocessor are translated by the IMU to real addresses of a reserved memory region (e.g., a dual-port memory excluded from the virtual memory mapping). Otherwise, an interrupt is generated and the OS event handling is requested. Although it is excluded from the virtual memory mapping, the reserved memory region is managed by the OS and divided into pages.

Figure 4 shows how the virtual addresses generated by a standardised coprocessor are translated by the IMU. The IMU in the figure is for the real system we have built and which is described in Section 4. The interface between the coprocessor and the IMU is quite simple: It consists of address lines (`CP_OBJ` and `CP_ADDR`), data lines (`CP_DIN` and `CP_DOUT`), and control lines (`CP_CONTROL`). Towards the rest of the system there are platform-specific signals (the dual-port RAM access lines `DP_ADDR`, `DP_DIN`, `DP_DOUT`, `DP_EN`, `DP_WR`, in the case shown in Figure 4), and these would differ across different platforms. Inside the IMU, 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 IMU with a conventional MMU [8]. Apart from typical status and control registers (`SR` and `CR`), there is an address register (`AR`) that holds the address of the coprocessor memory access performed most recently. By examining this register, the OS can determine which memory access possibly caused an access fault. Based on this information the appropriate interface management action can be taken.

The key part of the IMU is actually the TLB that performs address translation for coprocessor accesses. Its design is platform-specific as it reflects the organisation of the memory region accessible by the coprocessor. As in typical VMM systems, an upper part (most significant bits) of the coprocessor address is matched to the patterns in the translation table. If a match is found, the physical address is formed out of the translation information and the lower part (less significant bits) of the coprocessor address. 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 in typical VMM systems [8].

Control signals between the coprocessor and the IMU are the following: (1) `CP_START`, the coprocessor start signal, issued by the IMU once a user initiates the execution; (2) `CP_ACCESS`, the coprocessor access signal, indicates that there is an access 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 a memory access, the coprocessor should first wait for this signal to appear; (5) `CP_FIN`, the coprocessor completion signal, indicates to the IMU that the coprocessor has finished its operation.

Besides accessing the memory, the IMU provides a generic way to pass parameters to the coprocessor. Once its operation is started, the coprocessor looks for parameters in a memory page designated 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 data mapping purposes.

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 interfacing resources (i.e., shared or dual-port memory) between processor and coprocessor, which makes it possible for the application programmer to ignore the physical extent of the resource. Such result can most transparently be achieved through the involvement of the OS as discussed below.

3.3 Interface Management

The memory is logically organised in pages, as in typical memory systems. Datasets accessed by the coprocessor are mapped to these pages. The OS keeps track of the pages each dataset currently occupies. Not necessarily all of the datasets used by the coprocessor reside in the memory at the same time. At some point in time, the memory access patterns of the coprocessor are those that determine the

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

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

cycle 3:
  reg_b := CP_DIN;
  reg_c := reg_a + reg_b;
  CP_OBJ <= 2; -- object C[]
  CP_ADDR <= reg_i; CP_DOUT <= reg_c; CP_ACCESS <= '1';
  CP_WR <= '1';
  reg_i <= reg_i + 1;
```

Figure 5: Code snippet of an elementary coprocessor using the virtualisation interface. No address calculation is necessary, nor it is necessary to know the available memory size.

occupation of the available pages.

The interface manager responds to the requests coming from the IMU. The OS determines the cause of the interrupt by examining the state of the IMU. There are two possible requests:

Page Fault. If the IMU signals a page fault, it means that the coprocessor attempted an access of a dataset part not currently in the dual-port memory. The OS rearranges the current mapping to the dual-port 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. If the page is dirty its contents are copied back to the user-space memory and the page is newly allocated for the missing data; the missing dataset part is therefore made available and the translation hardware reflects the changes. Afterward, the OS allows the IMU to restart the translation and lets the coprocessor exit from the stalled state.

End of Operation. Once the coprocessor finishes regularly its task, through the IMU it signalises the end of operation to the main processor. The interface manager copies back to user space all the dirty data currently residing in the dual-port memory. The coprocessor should be ready and waiting for new execution, if another `FPGA_EXECUTE` call appears.

When no page is available for allocation, several replacement policies are possible (e.g., first-in first-out, least recently used, random). Also, speculative actions as prefetching could be used in order to avoid translation misses. To allow fine tuning of actions performed by the interface manager, the use of optimisation hints passed as parameters to the OS services is envisioned. Although the interface management task is similar to a classic VMM, the application of such techniques to the allocation of interface resources is novel and brings new advantages.

3.4 Example

The coprocessor code shown in Figure 5, written in VHDL-like syntax, 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*. A vector identifier (0, 1, and 2, in this example)

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

FPGA_LOAD(ADD_bitstream);

FPGA_MAP_OBJECT(0, A, SIZE, IN);
FPGA_MAP_OBJECT(1, B, SIZE, IN);
FPGA_MAP_OBJECT(2, C, SIZE, OUT);

FPGA_EXECUTE(SIZE);
```

Figure 6: Example of application C code. The semantics is similar to a function call with parameters passed by reference. There is no dependence on the available memory size.

and the corresponding index (*reg_i* in this example) constitute a virtual address and are sent to the interface. The VIM automatically translates this information into physical addresses, if possible, or invokes the OS, if the 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 to be changed 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, synchronicity/asynchronicity)—as in many wrapper-based abstract interfaces such as [5]—but also to the overall memory size and allocation policy.

Figure 6 shows how the C file which originally computed $C[i] = A[i] + B[i]$ needs to be modified in order 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 passed by reference (see Figure 3).

4. EXPERIMENTAL SETUP AND DEMONSTRATION

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

The IMU is designed in VHDL to be synthesised together with a coprocessor. The TLB, the most critical part of the IMU, 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. Note that, although we had to implement the IMU in FPGA for these experiments, IMUs could and should, in principle, become standard components implemented on the ASIC platform in the same way MMUs are today. 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. The

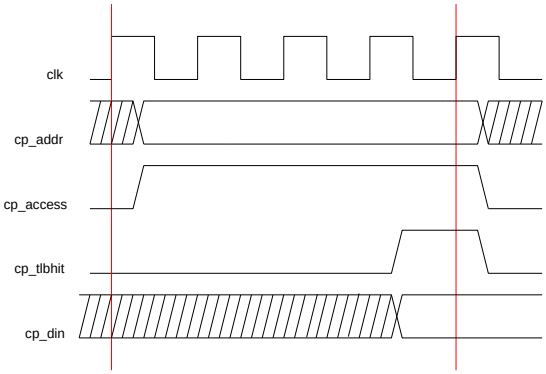

Figure 7: The coprocessor read access. Data is ready on the fourth rising edge of the clock.

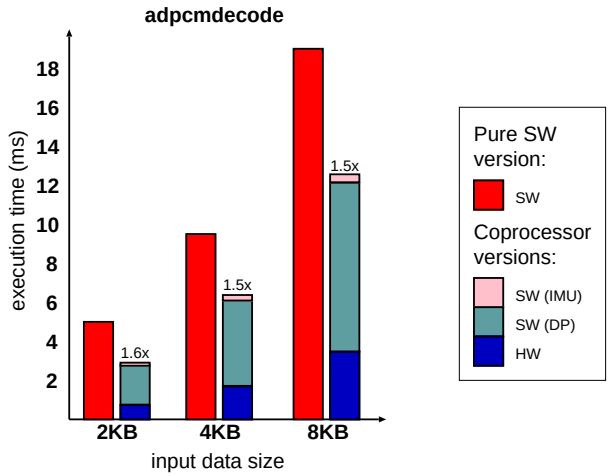

Figure 8: Measurements on `adpcmdecode` kernel. A software implementation, and hardware VIM-based implementation (the coprocessor and the IMU).

timing diagram for the current implementation is shown in Figure 7.

Through the IMU, 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). It is logically organised in eight 2KB pages (the total size is therefore of 16KB). In principle, there is no particular need for a dual-port memory, since the main processor and the coprocessor never access it at the same time. On the other hand, it has been chosen because of direct and easy interfacing with PLD.

The VIM is implemented as a Linux kernel module tuned to the hardware characteristics of the particular system. Using the module on the system with different size of the 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 common multimedia benchmark, `adpcmdecode` (running at 40MHz), and a cryptographic application, IDEA (running at 6MHz). For both, the critical parts were implemented in VHDL as standard coprocessors using the virtual interface provided by the IMU. The original C code was manually modified to make use of the OS services provided by the VIM and described in Section 3.

Figure 8 shows execution times for pure software and VIM-based versions of `adpcmdecode` for different input data sizes. The `adpcmdecode` coprocessor and the IMU are running at the frequency of 40MHz. Pure software and VIM-based versions are both running on top of the OS. The `adpcmdecode` produces 4 times the input data size. In this way, for an input data size of 2 KB, which fits a single page, the output data size is 4 pages. In this case, all data can fit the dual-port RAM and the application execution completes without causing page faults. For all other input sizes, page faults occur.

Figure 9 shows execution times for pure software, typical coprocessor (with no OS), and VIM-based versions of the IDEA cryptography application. A complex coprocessor core running at 6MHz with 3 pipeline stages is designed for IDEA. The IMU and IDEA's memory subsystem are running at 24MHz and the synchronisation with the IDEA core is provided by a stall mechanism. The IDEA coprocessor achieves considerable speedup comparing to the software case. Exploiting IDEA's parallelism in hardware was limited by the limited PLD resources of the device used. With larger PLD, additional speedup could be obtained.

For the VIM-based version, three components of the execution time are measured: (1) hardware execution time (time spent in the coprocessor and in the IMU, required for computation, memory accesses, and virtual memory translations), (2) software execution time for the dual-port RAM management (time spent in the OS transferring data from/to user-space memory), and (3) software execution time for the IMU management (time spent in the OS checking which address has generated the fault and updating the translation table). It should be noticed that for the typical hardware and the VIM-based versions, the speedup is comparable when no translation misses require intervention of the OS. In the case of the VIM-based versions, as the data set size grows up and misses appear (from 4KB onwards, for both applications), more time is spent in the OS but the speedup 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 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 DP-RAM*. Programming is made easier (both in C and VHDL) because no explicit reference to the dual-port memory is required.

A few conclusions can be drawn from Figure 8 and Figure 9. First, the presence of our virtualisation layer, while adding portability benefits, 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 IMU management can be seen in the figure and it is up to 2.5% of the total execution time. The hardware execution time includes address translation, whose overhead is unfortunately not always negligible (in the IDEA case around 20%); we

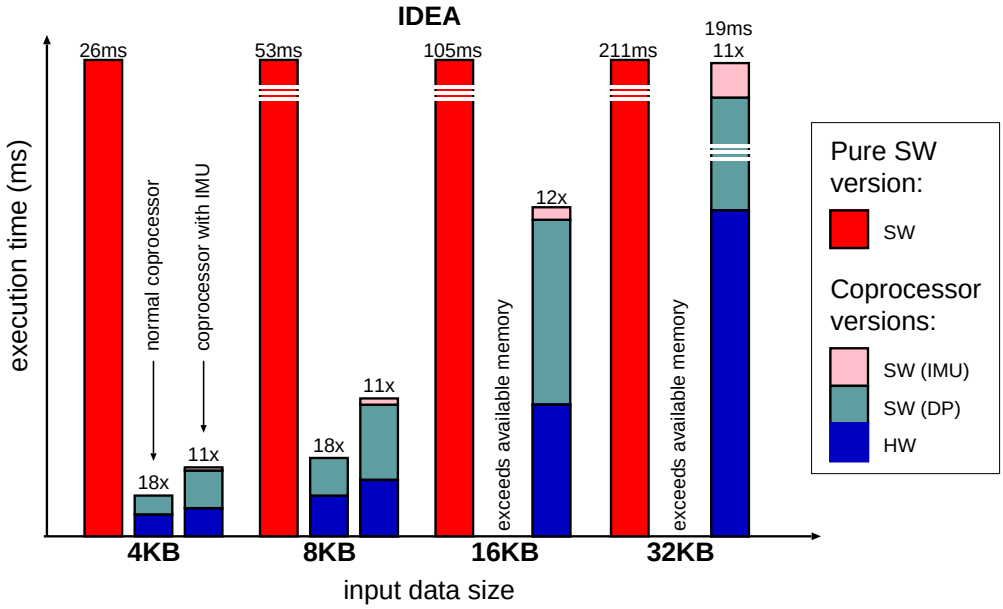

Figure 9: Measurements on `IDEA` kernel. A pure software implementation, a normal coprocessor without our virtual interface, and a VIM-based coprocessor with the IMU.

are now working on a pipelined implementation of the IMU which is expected to mask almost completely the translation overhead. Also, one should consider that eventually the IMU should be implemented in VLSI technology exactly as the MMU which is already present on the chip we use. The largest fraction of overhead is actually due to managing the dual-port memory. Note that part of this overhead component consists of compulsory page misses and would be unavoidable even if no virtualisation was applied.

The significant overhead in the dual-port RAM management between a normal coprocessor and a VIM-based is largely caused by our simple implementation of the VIM which makes two transfers each time a page is loaded or unloaded from the dual-port memory. We are currently removing this limitation. Real page misses can be improved by smarter memory allocation and prefetching techniques—the latter allowing overlapping of processor and coprocessor execution.

Of course, if the same experiments were to be performed on a different hardware platform this would require porting the IMU hardware and the VIM software, *but would not require any changes 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 (for instance, [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 in particular 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. We use simple access patterns for validation, but any access pattern could be used in conjunction with an IMU 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 [16]—that 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

allowing the OS to share dynamically the reconfigurable logic between applications. The resource is virtualised and special hardware support has been 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. 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 thus it differs from our approach.

6. CONCLUSIONS

In this paper we add a Virtual Interface Manager to a reconfigurable computing platform. The purpose is twofold: achieve a much more straightforward programming paradigm and ease the portability of applications.

The idea is not to improve the performance of the reconfigurable system; rather, 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 a full-fledged 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 interface is generally limited and we are working toward further reducing it. In both cases the coprocessors achieve a significant speed-up 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. ACKNOWLEDGEMENTS

The authors acknowledge the immeasurable help and support provided by Cédric Gaudin and Jean-Luc Beuchat.

8. 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/`.