UNIFYING SOFTWARE AND HARDWARE OF MULTITHREADED RECONFIGURABLE APPLICATIONS WITHIN OPERATING SYSTEM PROCESSES

THÈSE N^{O} 3626 (2006)

PRÉSENTÉE LE 22 SEPTEMBRE 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire d'architecture de processeurs

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Miljan VULETIĆ

Ingénieur diplômé de l'Université de Belgrade, Yougoslavie
de nationalité serbe et monténégrine

acceptée sur proposition du jury:

Prof. E. Sanchez, président du jury
Prof. P. Ienne, directeur de thèse
Prof. D. Andrews, rapporteur
Prof. G. Brebner, rapporteur
Prof. W. Zwaenepoel, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Marici i Slobodanu

Acknowledgements

First of all, I wish to express my sincere thanks to my advisor, Paolo, for his counsel, guidelines, and help that he provided during all these years I spent at EPFL. We had many cheerful discussions that sprang the ideas presented in this thesis and late-night (or early-morning) paper submissions that eventually resulted in publications (well, most of them). Apart from being a great thesis advisor, Paolo was a steady companion for a pint of ale and a handful of conversation, after long working days at conferences and meetings. These rare relaxing moments I really appreciate.

Special thanks are due to David Andrews and Gordon Brebner, both of them being the members of my thesis committee. The discussions I had with them helped in evaluating my work within the research community and, at the same time, in answering some challenging related questions. I would also like to thank Willy Zwaenepoel for his participation in the thesis committee and Eduardo Sanchez for presiding the thesis committee and for giving inspiring counsels on reconfigurable computing.

I extend my thanks to Laura Pozzi, who offered valuable help while I was focusing my research topics, and to Nuria Pazos Escudero, who gave useful remarks while I was finishing the thesis writing. Jean-Luc Beuchat, at the very beginning, and Walter Stechele and his group, at the very end, helped in applying the ideas presented in the thesis to real-world problems of software and hardware interfacing. I acknowledge and appreciate the hard work of the students that I followed, whose semester, internship, and diploma projects fostered the practical proof of my thesis.

Of all wonderful moments spent at the Processor Architecture Laboratory (LAP), I will most surely remember the outdoor activities (barbecues, hiking, and skiing) and friendly chats (of rather nontechnical content), fulfilled with laughter and enthusiasm of former and current LAP members. I would like to thank them all for these good moments.

Although hard work is the essence of a PhD student being, it was not just everything. My friends from Lausanne and all over (luckily, there are many to be recognised, whereas mentioning all of them would be error prone) showed me different and pretty successful ways (not to confuse with beer and wine) how to stop thinking about computer engineering. I admire their friendly attitude and social initiative, even if I was difficult to persuade for leaving my research duties (except on a sunny day with powder snow or for an important match of *E2050* or *Fusion Forte* football teams).

While my parents, Marica and Slobodan, and my sister, Jelena, were everlasting origin of long-haul care, "Made-in-Serbia" warmth, and family love, for which I am grateful and I will forever reciprocate, my wife, Chantal, was neverending source of short-haul encouragement, thorough understanding, and stellar love. She brightened my Lausanne sky like the most glittering and beloved star of Earendil, and to her go all my love and gratitude.

Abstract

Novel reconfigurable *System-on-Chip* (SoC) devices offer combining software with application-specific hardware accelerators to speed up applications. However, by mixing user software and user hardware, principal programming abstractions and system-software commodities are usually lost, since hardware accelerators (1) do not have execution context—it is typically the programmer who is supposed to provide it, for each accelerator, (2) do not have virtual memory abstraction—it is again programmer who shall communicate data from user software space to user hardware, even if it is usually burdensome (or sometimes impossible!), (3) cannot invoke system services (e.g., to allocate memory, open files, communicate), and (4) are not easily portable—they depend mostly on system-level interfacing, although they logically belong to the application level.

We introduce a unified *Operating System* (OS) process for codesigned reconfigurable applications that provides (1) unified memory abstraction for software and hardware application parts, (2) execution transfers from software to hardware and vice versa, thus enabling hardware accelerators to use systems services and callback other software and hardware functions, and (3) multithreaded execution of multiple software and hardware threads. The unified OS process ensures portability of codesigned applications, by providing standardised means of interfacing.

Having just-another abstraction layer usually affects performance: we show that the runtime optimisations in the system layer supporting the unified OS process can minimise the performance loss and even outperform typical approaches. The unified OS process also fosters unrestricted automated synthesis of software to hardware, thus allowing unlimited migration of application components. We demonstrate the advantages of the unified OS process in practice, for Linux systems running on Xilinx Virtex-II Pro and Altera Excalibur reconfigurable devices.

Keywords: HW/SW Codesign, Hardware Accelerators, OS Support, Virtual Memory, Execution Context for Codesigned Applications, Reconfigurable Computing.

Résumé

Les systèmes intégrés sur puce ("*Systems-on-Chip*") reconfigurables offrent la possibilité d'accélérer des applications en combinant du logiciel avec des accélérateurs spécifiques implémentés en matériel. Cependant, la combinaison du logiciel utilisateur avec le matériel utilisateur ne permet plus de bénéficier des abstractions principales de programmation et des facilités d'utilisation du système d'exploitation. En effet, les accélérateurs d'applications implémentés en matériel : (1) n'ont pas de contexte d'exécution (c'est au programmeur d'assurer ce contexte pour chaque accélérateur) ; (2) ne peuvent pas profiter de la mémoire virtuelle (c'est à nouveau le programmeur qui doit transférer les données depuis l'espace de mémoire utilisateur à l'espace de mémoire matériel, même si c'est pénible voire même impossible, selon les cas!) ; (3) ne peuvent pas appeler les services du système d'exploitation; et (4) ne sont pas facilement portable (ils dépendent des interfaces du système, même s'ils appartiennent sémantiquement au niveau d'application).

Pour les applications conçues en logiciel et en matériel reconfigurable, nous proposons un concept de processus du système d'exploitation *unifié*. Ce concept permet : (1) l'abstraction de mémoire unifiant les parties de l'application logicielles et matérielles ; (2) les transferts d'exécution du logiciel vers le matériel et vice-versa, afin que les accélérateurs matériels puissent appeler les services systèmes et d'autres fonctions logicielles ou matérielles ; et (3) l'exécution de plusieurs processus légers ("*multithreaded execution*") en logiciel et en matériel. De plus, l'utilisation du processus du système d'exploitation unifié assure la portabilité des applications conçues partiellement en logiciel et en matériel, offrant ainsi des moyens standard d'interfaçage.

En général, le fait d'introduire une nouvelle couche d'abstraction influence significativement la performance des applications: contrairement à cette attente, nous montrons que les optimisations en cours d'exécution effectuées dans la couche du système contenant le processus unifié permettent de minimiser la perte de performance et même, dans certain cas, de dépasser les approches typiques. De plus, le processus unifié rend possible la synthèse automatique et sans restriction du logiciel jusqu'au matériel ; c'est-à-dire, il est possible de migrer sans restriction les composants de l'application du logiciel vers le matériel ou inversement. Nous démontrons par des cas pratiques les avantages du processus unifié mentionnés ci-dessus sur un système Linux s'exécutant sur deux plateformes reconfigurables.

Les mots-clés: Accélérateurs en Matériel, Contexte d'Exécution des Applications Conçues en Logiciel et en Matériel, Ordinateurs Reconfigurable.

Contents

List of Figures

Chapter 1

Introduction

'The time has come,' the Walrus said,
'To talk of many things:
Of shoes—and ships—and sealing-wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings.'
—Lewis Carroll, *Through the Looking-glass*

THIS thesis shows that an *Operating System* (OS) providing a unified process context for running codesigned hardware and software applications can (1) simplify software and hardware interfacing, (2) support advanced programming paradigms, and (3) enable runtime optimisations and unrestricted automated synthesis; all these benefits can be achieved for an affordable cost. In this introduction, we first recall major abstractions that system software provides to software-only user applications; then, we introduce our contribution, through the discussion on the lack of such abstractions for codesigned applications; finally, we present the organisation of the thesis.

1.1 User Applications and OS Processes

User applications are rarely run bare on the underlying system hardware. An abstraction layer created through system software provides the execution environment, eases programming, increases portability, improves security, and releases application programmers from managing and sharing system hardware resources (i.e., processors, memories, mass storage devices, input/output peripherals) [110, 116]. Such abstraction is to date either missing or incomplete for codesigned hardware and software applications. This thesis proposes a solution providing the unified and complete abstraction for codesigned applications.

A *process* (or a task) is one of the basic concepts in the OS design: it represents an instance of a running user program [12, 116, 117]. Multiple processes can coexist and run in a computer system at the same time; they can share physical resources based on the OS services but still be screened from each other. Figure 1.1 shows several

system and user processes (software-only) running upon a Unix-like OS [13]. In the figure, we distinguish the following categories: *system hardware*, *system software*, and *user software*.

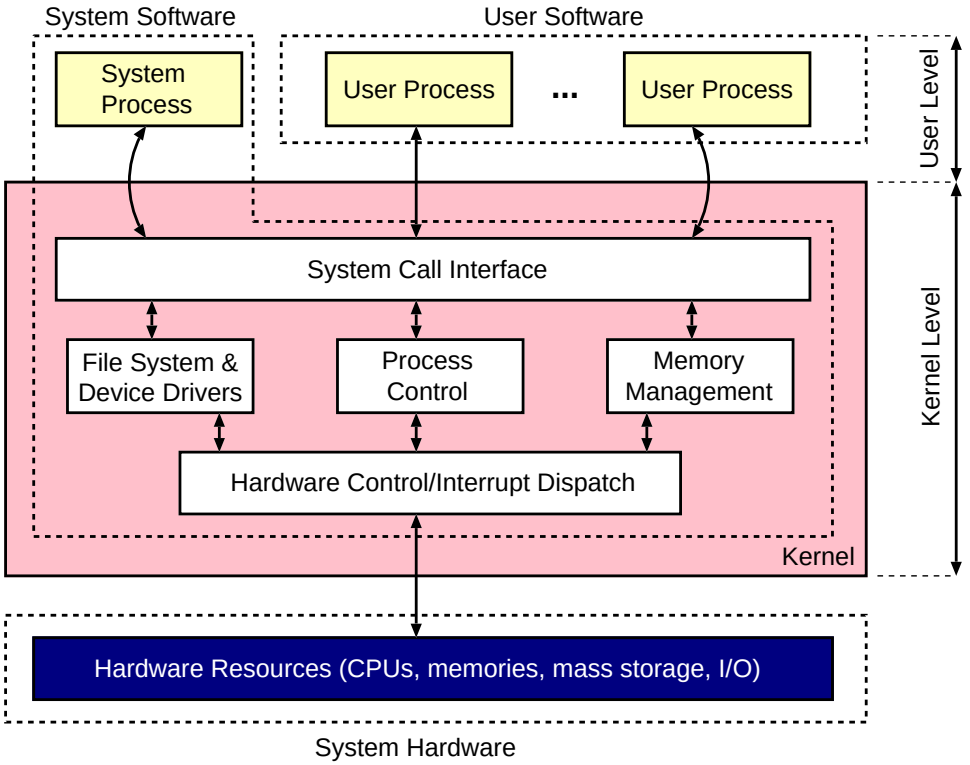


Figure 1.1: User software and system software running on system hardware. System software completely screens user software from system hardware.

System hardware represents hardware components of a computer system. The system hardware may comprise different resource types: computational resources (*Central Processing Units*—CPUs), memory resources (including mass storage devices), communication resources (human-computer interfaces, network interfaces), etc.

System software is responsible for controlling, integrating, and managing the individual hardware components of a computer system [126]. It runs on a CPU either as a *kernel* (i.e., software layer wrapping hardware components of the system [110, 116], running in a privileged mode of the CPU execution [4, 81, 105, 107]) or as a system process (running in an unprivileged mode of the CPU execution).

User software represents a user application, written and compiled by computer system users; it always runs on the CPU in user mode of execution; its use of system resources is restricted and managed by system software. The user software runs in the abstract context of an OS process and uses hardware resources exclusively through system services. The OS system call interface completely hides interfacing particularities and protects the system hardware from end users.

While running within the context of a process, user applications benefit from the virtual memory abstraction. Thanks to the system software support, any user

program has an illusion of having the whole machine for itself: the process context comprises its own memory address space and the machine state. Wherever the code and data physically reside, user applications see the image of the linear address space; there is no notion of the physical memory capacity and organisation. Beside standardising the runtime environment, the OS process provides user applications with dynamic memory management facilities, through stack and heap mechanisms [104].

The benefits of user programs using system services are multiple: (1) parallelism even on uniprocessor systems—if a process is blocked in a system call waiting on an event, other processes can use the CPU for computation, (2) portability across different platforms—if architecturally different computer systems support the same system call interface, one should be able to port simply application code just by recompiling, (3) abstracted access to hardware—there is no need to know particular characteristics of hardware devices, (4) optimised resource sharing—the system software can optimise the resource usage based on the process behaviour. Although the overall performance of the system shown in Figure 1.1 is often suboptimal, the benefits of the abstraction are overwhelming.

1.2 Codesigned Applications in Software-centric Systems

We call *codesigned hardware and software applications* the applications that have some of their parts running on CPUs and the rest running in specialised hardware (i.e., hardware specifically designed to speed up and parallelise the execution of performance demanding tasks). Codesigned applications blend two models of computation [38]: (1) temporal computation (scarce hardware resources such as ALUs, shifters, multipliers are reused in time by the instruction sequence being executed); (2) spatial computation (where abundant hardware resources are wired and deployed in space—silicon area—to fit more closely the nature of the application, thus maximising the processing parallelism). Standard processors (especially those intended for embedded applications—typically smaller and slower than cutting-edge CPUs but more power efficient) are often extended with application-specific hardware accelerators [51, 58, 128] to achieve performance goals and meet power constraints. The approach becomes widely used, especially with recent developments in designing versatile *Systems-on-Chip* (SoC), the market growth of consumer appliances, and *Field Programmable Gate Arrays* (FPGAs) [3, 130] coming of age.

By extending the CPU with the application-specific hardware, codesigned applications consist of heterogeneous code: (a) the software part consisting of CPU instructions—typically generated by a compiler from a specification (program) written in a high-level programming language, and (b) the hardware part consisting of hardwired logic or FPGA configuration [38, 133]—typically generated by a synthesiser from a specification (description) written in a *Hardware Description Language* (HDL). Figure 1.2 shows a heterogeneous-code user program executing within the context of an OS process. The process executes on the CPU and, at one point

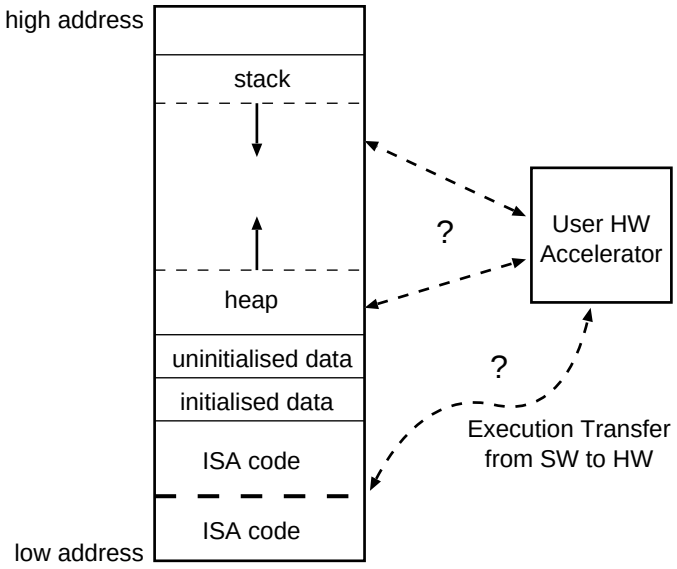


Figure 1.2: Memory space of mixed software and hardware process. Hardware accelerator does not belong to the context of the software process: it neither shares memory abstraction with software nor it uses standardised means of execution transfers.

in time, transfers the execution to the hardware accelerator; when the accelerator finishes the computation, it returns back the control to the software part. The hardware part of the code does not belong to the process context; there is neither virtual memory support, nor standardised means of hardware invocation—software programmers and hardware designers have to solve the interfacing for a particular architecture they use. This thesis addresses the problem of *user hardware* missing the abstractions that are already available to user software.

User hardware—application specific hardware accelerators running on behalf of user software—is semantically linked to a particular user application, since software and hardware parts of the application exhibit jointly some specified functionality and operate on the same data. We may think of FPGAs as a system-level resource available for implementing both user hardware (the FPGA is a computational resource for application-specific accelerators programmed with configuration in place of the ISA—*Instruction Set Architecture*—code) and system hardware (the FPGA is a control-building resource for system devices such as peripheral controllers).

The Y-chart in Figure 1.3 shows the different levels of abstraction—provided and managed by the OS—visible to software and hardware parts of a traditional codesigned application running in a software-centric system. User software does not recognise user hardware as its peer. Abstraction levels (such as virtual memory and system-service interface) typically available to user software do not exist for user hardware.

The missing abstraction imposes limits on: (1) interfacing hardware and software parts of codesigned applications—while writing user software and while designing user hardware, software programmers and hardware designers have to be aware of specific interfacing details; (2) available programming paradigms—advanced programming concepts such as multithreading assume unified memory space and sys-

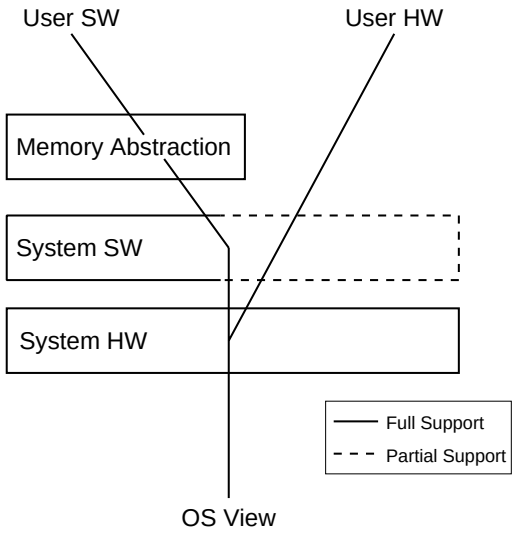


Figure 1.3: Levels of abstraction for traditional codesigned applications in a software-centric system. User hardware misses system abstractions available to user software.

tem-enforced memory consistency; (3) portability of codesigned applications across different platforms—it is burdensome for programmers to preserve their hardware-agnostic, high-level programming approaches, and it is challenging for hardware designers to write accelerators that can run across different platforms, without any change in the HDL code; (4) user hardware ability to use results of system services—if there is no memory abstraction present, this is either cumbersome or even impossible (as an example, we can think of dynamic memory allocation). The thesis proposes a solution to remove the imposed limits.

1.3 Seamless Interfacing of Software and Hardware

Our goal is to provide the missing abstractions for user hardware and bring user software and user hardware to the same conceptual level. We propose a unified process context for heterogeneous-code programs (consisting of user software and user hardware) to achieve transparent software and hardware interfacing and their seamless integration. In this way, by delegating platform-specific tasks to a system-level virtualisation layer, we also increase the portability of codesigned applications. The virtualisation layer provides unified memory abstraction for software and hardware processes and hides platform details from users as much as general-purpose computers do. It consists of an OS extension relying on a system hardware extension that provides (1) unified virtual memory, (2) execution transfers from software to hardware and vice versa (such that user hardware becomes able to callback software and use system services), and (3) multithreaded execution, for user software and hardware accelerators running within the same OS process. The presence of the virtualisation layer brings platform-agnostic interfacing and enables (1) dynamic optimisations of codesigned applications, and (2) unrestricted automated synthesis from high-level programming languages.

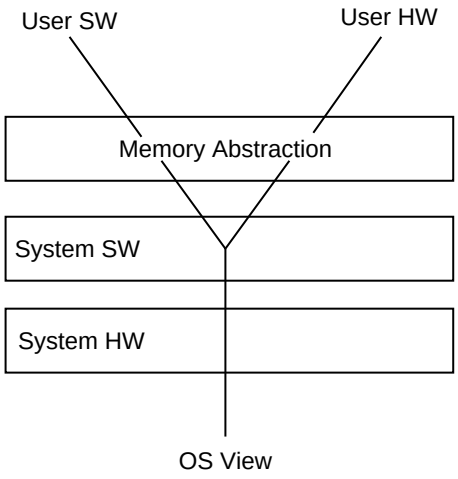


Figure 1.4: User software and user hardware as peers in a proposed system supporting codesigned applications. User hardware can run together with user software in the same execution context.

In contrast to Figure 1.3, Figure 1.4 shows the unified abstraction levels that we introduce visible by both user software and user hardware components of codesigned applications. We delegate the system specific tasks to the OS, which makes possible to extend the incomplete or missing abstraction layers from Figure 1.3 to the full boxes in Figure 1.4. The extended abstraction minimises the programmer and designer efforts, when interfacing user software and user hardware of codesigned applications. It also enables simultaneous execution of user hardware and user software within the same OS process. Although the introduction of an additional abstraction usually brings overheads, we show how its presence can be turned into additional advantages. More importantly, the advantages are achieved without any user intervention. Similar things happen in an OS, where system software screens the user from a number of runtime optimisations [12, 117].

1.4 Thesis Organisation

Our contribution, starting from a simple idea of unifying the execution context for software and hardware parts of codesigned applications, provides the first general evaluation of this concept in practice—on real reconfigurable systems. We show that—even with the overheads of our mixed software-and-hardware approach—having unified memory for user software and user hardware is beneficial, in the terms of simplified hardware design and software programming, while the advantageous performance of spatial execution is only moderately affected. When building future systems for codesigned applications, designers can immediately rely on our results and our overhead analysis.

In Chapter 2, we illustrate problems of hardware and software interfacing to motivate our contribution. In Chapter 3, we present the state-of-the-art and discuss related hardware and software interfacing approaches, already-proposed OS-based extensions for reconfigurable applications, and existing portability solutions that in-

what	where
unified memory	Chapter 4
execution transfers	Chapter 6
multithreading	Chapter 7

Table 1.1: Our principal contributions and the corresponding chapters.

crease platform independence for codesigned applications. In the following chapters, we discuss different aspects of our contribution.

Table 1.1 summarises our principal contributions, by showing *what we add* (the first column) to form unified OS processes for running codesigned applications and *where we describe* (the second column) the corresponding implementation. Our contribution (marked in Table 1.1 by the numbers of the corresponding chapters—Chapter 4, 6, and 7) provides to OS processes running codesigned applications the abstractions, services, and programming paradigms already available to software-only processes.

We show an implementation of the unified virtual memory for mixed software and hardware processes in Chapter 4; there, we also explain the extensions of our implementation that enable dynamic, runtime optimisations to improve performance transparently to the end user—the virtual memory abstraction we propose not only introduces overheads but also brings additional advantages. We introduce a performance metric for heterogeneous computing in Chapter 5; afterwards, we perform experimental measurements on reconfigurable SoC platforms to show and discuss advantages and limited overheads of our unified memory scheme. In Chapter 6, we introduce the OS extensions supporting user hardware callbacks to user software—hardware becomes capable of invoking system calls directly; then, we show how these extensions, together with the unified memory abstraction, enable unrestricted automated synthesis of hardware accelerators from high-level programming languages. Having unified memory abstraction between software and hardware is essential for supporting multithreaded programming paradigm. In Chapter 7, we extend the virtualisation layer to support multithreading for codesigned applications. Finally, we conclude the thesis in Chapter 8 and give directions for future work.

Chapter 2

Missing Abstractions for Codesigned Applications

Wovon man nicht sprechen kann, darüber muß man schweigen.

What we cannot speak of, we must pass over in silence.
—Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

IN this chapter, we motivate the introduction of a unified process context for running codesigned applications by showing hardware and software interfacing problems in the case of typical existing architectures. The unified process abstraction that we introduce releases software programmers and hardware designers from interfacing problems. We discuss the system requirements to support this abstraction.

2.1 Process Model of Computation

An OS process—in most of its implementations—provides to a user program running within its context a model of computation based on the *random-access machine* [102]—an abstract machine definition from theoretical computer science. The model features a CPU (with separate control and processing units) interconnected to a random-access memory (as shown in Figure 2.1). The CPU executes a program (a sequence of instructions) and operates on data. The program and the data are all stored in the random-access memory. The computation performed by the CPU from Figure 2.1—in its simplest implementation—is purely temporal [38]: hardware resources (such as ALUs) are reused in time by the instruction sequence being executed.

As Figure 2.1 shows, the process model provides a unique, virtual memory address space. The architectural improvements of the physical underlying machine (such as memory hierarchy, multiple issue, pipelined and out-of-order execution [56]) are typically hidden from ordinary users; at most, only system software and compilers—especially in the case of VLIW machines [41]—are concerned. A memory manager component of the system software (e.g., the *Virtual Memory*

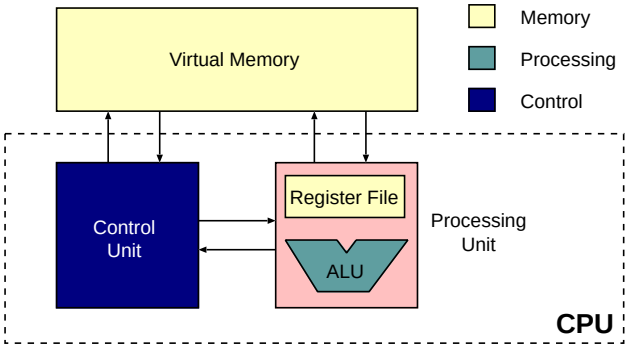


Figure 2.1: Process model of computation: a stored-program computer. The control unit fetches and executes the program stored in the random access memory. The program operates on data also stored in the same memory.

Manager—VMM—of the OS) provides the virtual memory abstraction, no matter what is the organisation of the underlying physical memory.

Figure 2.2 depicts the temporal computation machine from Figure 2.1 extended with a spatial computation engine—an application-specific hardware accelerator. In contrast to the CPU from Figure 2.1, we can notice the following differences: (1) the hardware accelerator employs custom processing units (*Data-Flow processing Units*—DFUs) deployed in space and tailored to fit as much as possible a specific application data-flow; (2) there is no centralised register file in the hardware accelerator but the data local to the computation are stored in distributed registers corresponding to the data-flow; (3) the number and the size of memory ports are custom to the application; (4) there is no centralised control, it is rather distributed across the accelerator; (5) the configuration (in the case of reconfigurable hardware accelerators) or hardwired logic gates (in the case of ASIC hardware accelerators) determine the behaviour of the accelerator, not the sequence of programming instructions.

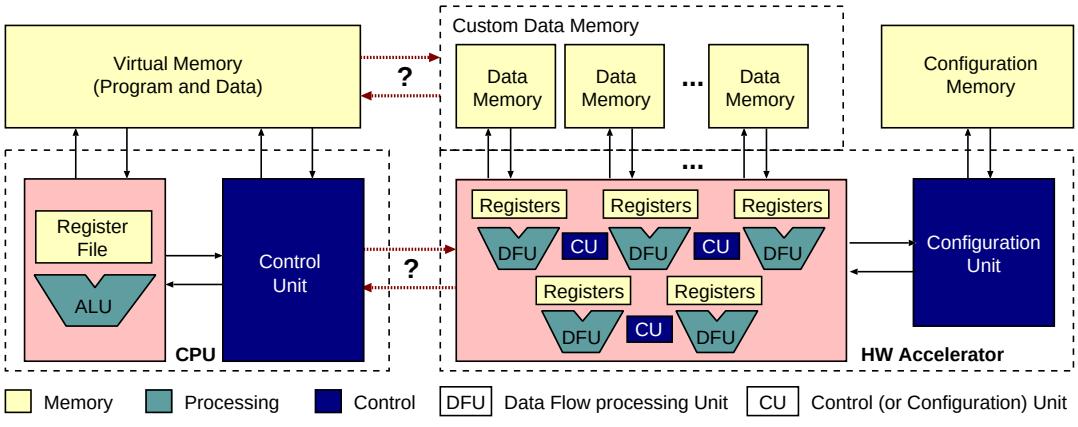


Figure 2.2: Blending temporal and spatial ways of computation. The hardware accelerator performs application-specific computation. There is no common address space between software and hardware. Furthermore, there is no standardised way of execution transfers.

Although the machine from Figure 2.2 may run a given application faster than the temporal machine from Figure 2.1, we notice that the virtual memory abstraction and the neat programming environment have disappeared: (1) user software and user hardware do not share the memory address space—it is typically on the application programmer to arrange the communication and data transfers; (2) there is no standardised way of execution transfer—it is again on the application programmer to explicitly control the hardware accelerator; (3) system services are only partially available to user hardware—programmers may use software wrappers to call system services on behalf of the user hardware. In the following sections, we illustrate the software programming and hardware design issues for typical existing architectures running codesigned applications [17, 72, 112].

2.2 Typical Architectures: Accessing Memory

Having disjoint memory address spaces for user software and user hardware exposes data communication to users; it is the task of software programmers and hardware designers to arrange memory transfers from software to hardware address spaces and vice versa. In this section, we show typical arrangements for hardware accelerator accesses to the memory.

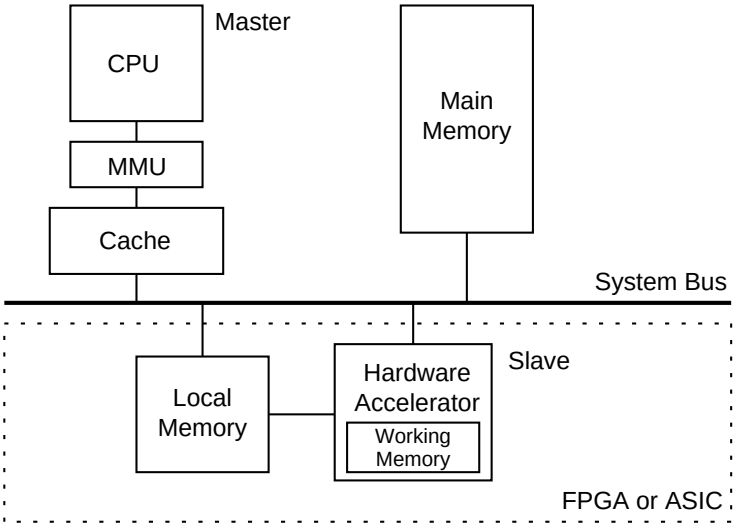


Figure 2.3: Typical hardware accelerator accessing local memory. While user software has an ideal image of the memory provided by virtual memory mechanisms, user hardware generates physical addresses of the local memory. It is the task of the programmer to arrange the communication between user software and user hardware.

Figure 2.3 shows a possible implementation of the machine from Figure 2.2: the hardware accelerator (user hardware) directly accesses a local on-chip memory to perform the computation. The user software, running on the CPU, has a perfect, linear image of the memory provided by the virtual memory manager of the OS [116]. For speeding up the execution, a system hardware unit called *Memory Management Unit* (MMU), and often integrated within the CPU [56] supports the translation of

virtual memory addresses. In contrast to the user software, the user hardware is directly interfaced to the system hardware and generates physical memory addresses for fast accesses to the local memory. The programmer controls the accelerator and accesses its local memory through a memory mapped region (this assumes using the *mmap* system call of the OS).

```
/* Software-only version */

idea_block A[n64], B[n64];
...
idea_cipher_sw(A, B, n64);
...
```

(a)

```
/* HW accelerator version accessing local memory */

idea_block *buff =
  mmap(0, sizeof(idea_block), LMEM_PHYADDR);
...
data_chunk = LMEM_SIZE / 2; data_ptr = 0;
while (data_ptr < n64 * sizeof(idea_block)) {
  memcpy(buf, A + data_ptr, data_chunk);
  IDEA_CTRL_REG = START;
  while(*IDEA_STATUS_REG != FINISH);
  IDEA_STATUS_REG = INIT;
  memcpy(B + data_pt, buf + data_chunk, data_chunk);
  data_ptr += data_chunk;
}
...
```

(b)

Figure 2.4: Programming for the IDEA cryptography application: software-only version (a) and HW accelerator version (b). Software-only case is neat and clean. The presence of the hardware accelerator demands programmer activities for controlling the accelerator, partitioning the data, and scheduling data transfers.

If the memory size is limited, the programmer is responsible for partitioning the data and scheduling the data transfers. Figure 2.4 compares programming of the IDEA cryptography application [82] for its software-only and codesigned implementations. The software-only version (in Figure 2.4a) just invokes the encryption function by passing the pointers to the input and output IDEA blocks. On the other side, using the hardware accelerator (in Figure 2.4b) demands partitioning the data to fit the local memory, transferring the data explicitly from the main memory to the local memory and the other way around, and iterating until the computation is finished. Although it is not a difficult task, it is quite burdensome and demands programmer's knowledge of the hardware memory access pattern. In principle, the local memory serves as a software managed cache or scratchpad [113].

Figure 2.5 shows another approach with a hardware accelerator capable of initiating master transactions on the system bus and directly accessing the main memory. The user software is responsible for controlling the accelerator and passing the physical addresses of a fixed memory region, previously reserved by the OS. Since user hardware generates physical addresses of the main memory, an erroneously-designed or malicious accelerator may cause nondeterministic behaviour and crashes of the whole system.

With the assumption that a large amount of the physical memory is available (which may not be always true, especially in the embedded applications), the programming is made simpler (as Figure 2.6b shows for an image processing application [15]) and closer to the pure software (shown in Figure 2.6a); there is no more need to partition and copy data iteratively. However, single accesses to the main

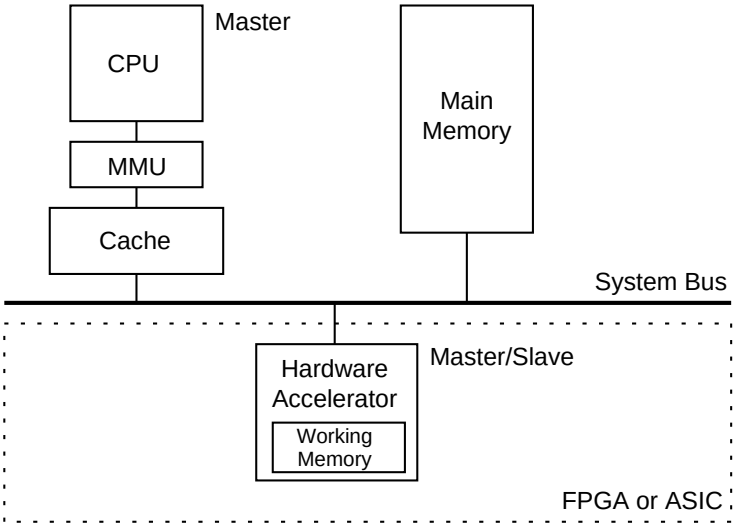


Figure 2.5: Typical hardware accelerator accessing main memory. While user software has an ideal image of the memory provided by virtual memory mechanisms, user hardware generates physical addresses of the main memory. It is the task of the programmer and hardware designer to arrange the communication properly.

memory are rather expensive. To overcome this drawback, the hardware designer has to manage and implement burst accesses to the main memory, which imposes creating buffers and local memory management on the hardware side: the programmer's burden from Figure 2.4b has not disappeared but *is just shifted to the hardware designer*; instead of managing the memory in user software, it becomes the task of user hardware. In the case of applications that exhibit multiple input and output data streams (as is the case with the contrast enhancement application from Figure 2.6b—four input images and one output image), the hardware designer would have to manage the corresponding number of input and output data buffers—filled and emptied by burst transfers—and synchronise them with the accelerator.

Our contribution liberates software programmers and hardware designers from burdensome memory transfers. We delegate the memory interfacing tasks to system software and system hardware. Chapter 4 explains a novel system architecture providing unified memory abstraction for user software and user hardware.

Performance Analysis. Figure 2.7 sketches possible execution timelines of the two presented typical approaches for a given application. The overall *Execution Time* (ET) of the approach with the local memory is the sum of the *Copy Time* (CT) and pure *Hardware execution Time* (HT). The overall execution time (ET) of the approach with the main memory consists of hardware executions interleaved (or partially overlapped if bursts are supported) with master memory accesses. If we assume identical computation cores of the hardware accelerators, the pure hardware execution time is the same: the overall performance figure depends on the effectiveness of memory transfers.

It is debatable which of the two approaches is better. A programmer responsible for data transfers to the local memory can overlap computation with data transfers (by dividing the local memory in two halves—the first processed by the hardware and

```
/* Software-only version */

unsigned char outimg[imgsize],
  inpimg[4][imgsize];
  ...
  contrast_enhancement_sw
    (outimg, inpimg, winsize, imgsize);
  ...
```

(a)

```
/* HW accelerator version
   accessing main memory */

unsigned char *resimg =
  mmap(0, imgsize, RES_PHYADDR);
unsigned char *inpimg[i] =
  mmap(0, winsize,INPi_PHYADDR);
...
memcpy(inpimg[i], cam_out[frame_i], winsize);
CONTRAST_CTRL_REG = START;
while(*CONTRAST_STATUS_REG != FINISH);
CONTRAST_STATUS_REG = INIT;
memcpy(outimg, resimg, winsize);
...
```

(b)

Figure 2.6: Programming for the contrast enhancement application: software-only version (a) and HW accelerator version (b). Software-only case is neat and clean. The presence of the hardware accelerator demands controlling the accelerator and using *mmap()* system call.

the second used for copying) or use a DMA (although this would mean descending from the user-level to system programming) to speed up the process. A hardware designer responsible for memory accesses to the main memory can use burst accesses and hardware-managed buffers to improve the performance. Whichever of the two approaches an application architect chooses, the memory management tasks, which are normally delegated to the system, burden the user-level software and hardware: pushing the management of the memory hierarchy from the programmer toward the virtual memory manager and the cache controller is the analogous assignment of the general computer architecture.

2.3 Typical Architectures: Callbacks to Software

There are some cases in which hardware accelerators may want to call back software. For example, hardware may request software to invoke a system call, service an exception, demand an external computation, display the accelerator status, or send a message to some other application part (Chapter 6 shows callbacks to software required for mapping high-level languages to hardware).

Programmers and designers wanting to support hardware callbacks to software encounter two principal difficulties: (1) writing additional code to service callbacks is necessary, and (2) parameter passing conventions are not standardised but rather chosen in an *ad-hoc* manner. The additional obstacle is the lack of the unified memory between software and hardware: if there is no memory abstraction present, some software return values can be completely useless to hardware (we can take *malloc()* function as an example).

While the runtime environment implements a *calling sequence*—the sequence of instructions and data arrangements necessary to perform a function call [2]—for software-only applications (usually through stack management), no such arrangement exists for codesigned applications; the programmer has to write the code that

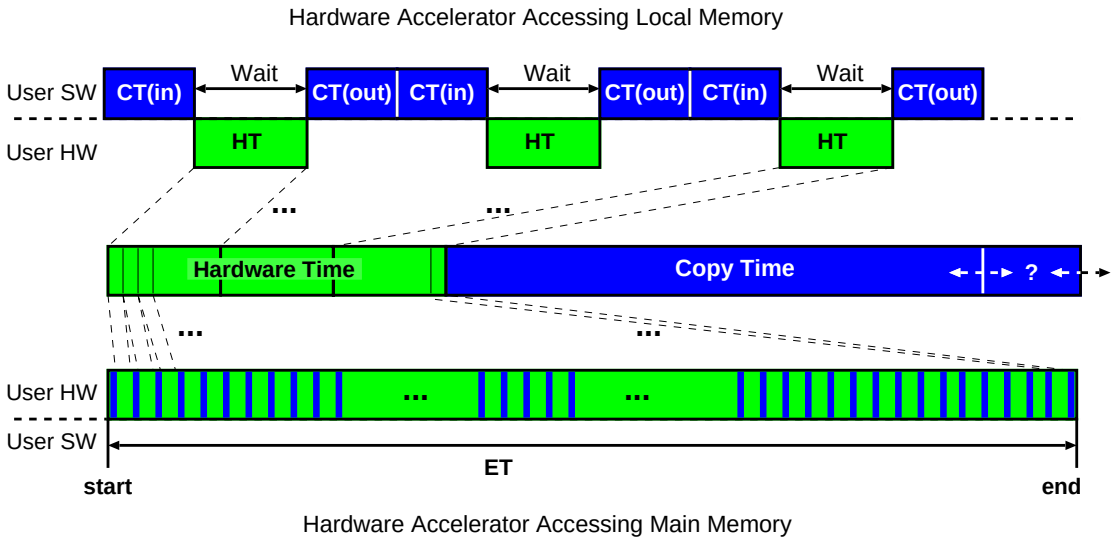


Figure 2.7: Execution timelines of typical hardware accelerators. The execution time of the typical hardware accelerator accessing local memory consists of memory transfer intervals (in user software) and hardware execution intervals (in user hardware). The execution time of the typical hardware accelerator accessing main memory consists of interleaved short memory-access intervals (using system bus transactions) and hardware-executions intervals. If one assumes the same computation cores, summing up hardware intervals for both accelerators gives the same amount of time spent in hardware execution. What matters to the overall performance is the effectiveness of memory accesses.

supports parameter exchange and hardware callbacks to software. Figure 2.8 shows an example code servicing multiple callback requests generated by a hardware accelerator. After launching the accelerator, the program loops until the `FINISH` signal arrives. Depending on a callback identification (`FPCOMP_ID` for a floating point computation, `MSEND_ID` for sending a message) received from the hardware, the programmer prepares invocation parameters and calls the appropriate software function. Once the function returns, the programmer passes the return value—supposedly non-void—to the accelerator and resumes its execution. The nonexistent memory abstraction brings additional complexity (similarly to what Figure 2.4b shows) to the code.

Our contribution shifts the parameter passing and function invocation tasks from the user to the system. With the execution transfers supported by the OS and assuming a unified memory abstraction, hardware accelerators become capable of using any system service or function from the standard library. Chapter 6 demonstrates how an OS can provide transparent execution transfers for hardware accelerators invoking software functions, be it system services or library calls.

2.4 Typical Architectures: Multithreading

The lack of unified memory and system-level support for codesigned applications narrows available programming paradigms. For example, running multithreaded

```
/* Typical HW accelerator calling back software */
...
HWACC_CTRL_REG = START;
while (*HWACC_STATUS_REG != FINISH) {
  switch(*HWACC_CBACK_ID) {
    case FPCOMP_ID: ...
      fpcomp(p1,p2); ...
      HWACC_CTRL_REG = RESUME;
      break;
    case MSEND_ID: ...
      msend(p1)); ...
      HWACC_CTRL_REG = RESUME;
      break;
    ...
    default:
      ... printf("Unknown callback ID."); ...
      break;
    }
  }
  ...
```

Figure 2.8: Servicing hardware accelerator callbacks to software. The programmer has to examine the callback identifier and, then, to dispatch the execution toward the corresponding function.

codesigned applications (in software-centric systems) demands additional activities on the programmer side to perform memory transfers and enforce memory consistency.

Figure 2.9a shows a simple program computing the sum of two vectors, using a POSIX-like [88] thread management. In the master thread, the programmer declares the vector pointers and the identifier of the slave thread function, initialises the vectors, and creates the slave thread by invoking the thread creation function (*thread_create*). Thread creation is similar to a function call, except that the caller and the callee continue their execution simultaneously (from the programmer's perspective) and perform their work in parallel. After doing some work simultaneously, the master eventually synchronises with the slave through the join primitive (i.e., it waits until the slave returns). The two threads share the virtual memory address space and they use the same memory pointers. Once the computation is finished, the master thread can immediately access the results through its pointer to the result vector `C`.

The fact that the threads share the same virtual memory address space (as Figure 2.10a indicates) is one of the crucial concepts of multithreading: the threads share the same memory, while having separate execution stacks.

We suppose now that the designer decides to move the vector-addition slave thread to hardware execution. Similarly to what we have seen in Section 2.2, without system-level support for threads executed in hardware, the programmer needs to take explicit care of the communication between the application software and hardware components. The master thread is unchanged, while a wrapper thread is now needed in order to control and transfer data to the hardware accelerator which is now responsible for the computation (Figure 2.9b shows a solution for integration of the hardware accelerator, using a software wrapper thread). The wrapper thread initialises the accelerator, copies data to the accelerator local memory, and launches

```
/* Master Thread */
void main() {
  int *A, *B, *C;
  int n;
  int thr_id;
  ...
  read(A, n);
  read(B, n);
  thr_id = thread_create
    (add_vect, A, B, C, n);
  do_some_work_meanwhile();
  thread_join(thr_id);
  ...
}

/* Slave Thread */
void add_vect(int *A, int *B, int *C, int n) {
  int i;
  for(i = 0; i < n; i++)
    C[i] = A[i] + B[i];
  }
}
```

(a)

```
/* Master Thread */
void main() {
  int *A, *B, *C;
  int n;
  int thr_id;
  ...
  read(A, n);
  read(B, n);
  thr_id = thread_create(add_vectors, A, B, C, n);
  do_some_work_meanwhile();
  thread_join(thr_id);
  ...
}

/* Wrapper Thread */
void add_vect(int *A, int *B, int *C, int n) {
  int d_chunk = BUF_SIZE / 3;
  int *d_ptr = 0;
  /* initialise accelerator */
  write(HWACC_CTRL, INIT);

  while (d_ptr < n) {
    copy(A + d_ptr, BUF_BASE, d_chunk);
    copy(B + d_ptr, BUF_BASE + d_chunk, d_chunk);
    /* launch accelerator */
    write(HWACC_CTRL, ADD_VECT);
    while () {
      if (read(HWACC_STATUS) == FINISHED) {
        copy(BUF_BASE + 2*d_chunk,
          C + d_ptr, d_chunk);
        break;
      } else {
        do_some_work_meanwhile();
      }
    }
    d_ptr += d_chunk;
  } ...
}
```

(b)

Figure 2.9: Multithreaded code for software-only (a) and hardware accelerator version (b) of codesigned application. In the software-only case, the threads use the same memory pointers. In the codesigned case, the programmer has to write a wrapper.

the computation. Since the input data does not necessarily fit to the local memory of the accelerator, the wrapper iteratively copies data back and forth, until all the data are processed. The hardware accelerator and the software threads do not share the same memory address space (as Figure 2.10b shows).

For a codesigned application with multiple threads in hardware, the programmer has to create a wrapper thread per hardware accelerator. Changing the HDL code of a hardware accelerator and its memory access pattern may require changes in the wrapper thread—the interfacing details burden software programmers and hardware designers.

Our contribution enables wrapper-free and access-pattern-independent multithreaded execution of user software and user hardware. Chapter 7 presents the system-level extensions to enforce memory consistency and support synchronisation, thus enabling multithreaded programming paradigm for codesigned applications.

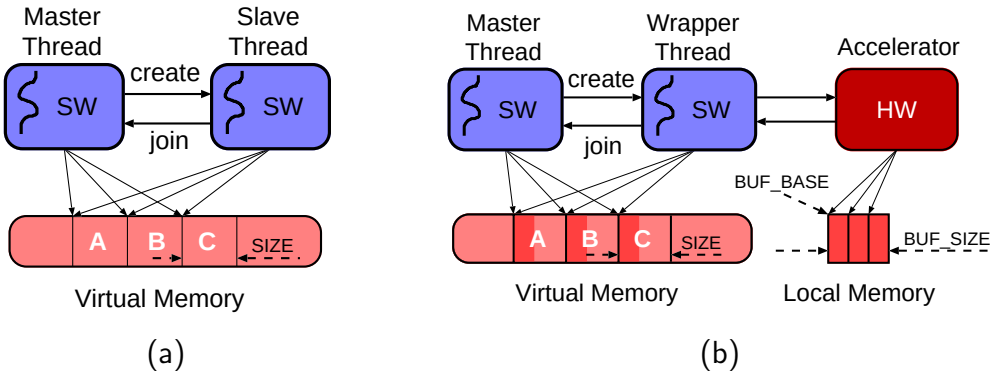


Figure 2.10: Multithreading memory perspective for software-only (a) and hardware accelerator (b) version of the codesigned application. In the software-only case, the threads share the same memory. In the case with the hardware accelerator, software and hardware address spaces are separate.

2.5 Unified Process Context for Codesigned Applications

To overcome the interfacing burdens and to enable user hardware invoking system services and software functions, we propose a system-supported, unified process context for machines that combine temporal and spatial models of computation (as Figure 2.11 shows). Having unified processes for codesigned applications simplifies the interfacing, memory communication, and execution transfers between user software and user hardware, thus allowing seamless integration of software and hardware application parts. The introduced high-level abstraction does not mean abandoning the performance benefits of application-specific execution and custom-memory architectures (as we will show later in this thesis).

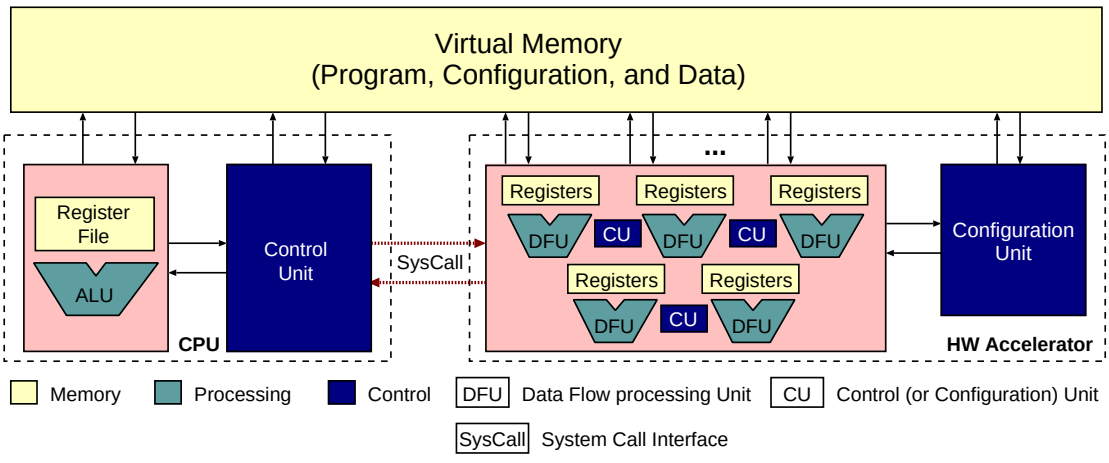


Figure 2.11: A heterogeneous-code unified-memory computer. The CPU executes the ISA code and performs temporal computation; the hardware accelerator performs spatial computation. Both share unified memory abstraction (where program, configuration, and data are stored) and use the system support for execution transfers between software and hardware application parts.

We propose making hardware accelerators capable of (1) accessing virtual memory and sharing the address space with software to enable transparent memory communication and screen user software and user hardware from the interfacing details, and (2) implementing a common calling sequence with software to support transparent execution transfers and user hardware callbacks to user software.

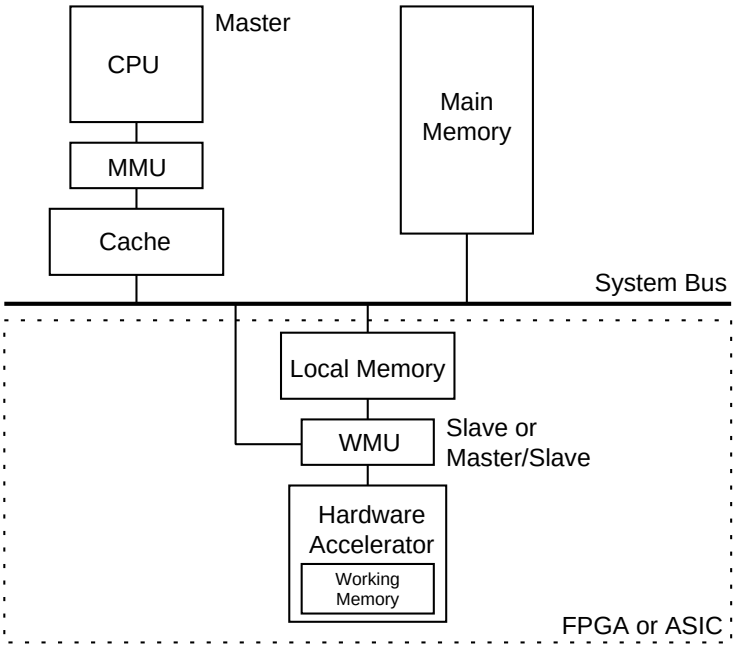


Figure 2.12: Hardware accelerator capable of accessing virtual memory of user process through *user hardWare memory Management Unit* (WMU), in a similar way as user software does through MMUs.

Figure 2.12 shows an architecture that allows an application-specific hardware accelerator to run in the process context of their peer software. Not only the unified process can simplify hardware and software interfacing but it can allow hardware accelerators to benefit—*transparently and without any need for user intervention*—from spatial and temporal locality of memory accesses (a well-known and largely-exploited concept from general-purpose computing).

Assuming the system-level support for unified processes, programming such a system is straightforward. In the programming presented in Figure 2.13, whatever the size of the data to process, the programmer can just pass the data pointers (`A` and `B`) and the number of blocks to encrypt (`n64`) to the hardware accelerator. There is no need to partition the data and schedule the transfers, as was the case for the typical approach shown in Figure 2.4b. It is the responsibility of system software and system hardware, as it is the case in the general-purpose computing systems.

On the hardware side, having the virtual memory abstraction allows the hardware designer to write HDL code independent of the memory size and location. It is the task of the system to ensure that the requested data is brought to the local memory acting as a cache. There is no need for either burst accesses or explicit buffering. The action is completely invisible for the hardware designer and, again,

```
/* Virtual memory-enabled hardware version */
idea_block A[n64], B[n64];
...
idea_cipher_hw(A, B, n64);
...
```

Figure 2.13: Programming for the IDEA cryptography application with a virtual-memory-enabled hardware accelerator. The accelerator is capable of accessing the data to process through the virtual memory pointers to the user address space.

it is on the system to perform it transparently. In a similar manner, the memory hierarchy in general-purpose systems is managed transparently to the end user (e.g., cache block transfers and virtual address translations).

Figure 2.14 compares excerpts of a VHDL-like code for both the typical and the virtual-memory-enabled IDEA hardware accelerators. While writing the code for the typical hardware accelerator, the hardware designer has to (1) use physical addresses of the local memory, (2) be aware of the memory size, and (3) arrange the memory partitioning in accordance with the programmer. The code is inherently platform dependent. While writing the code for the virtual-memory-enabled hardware accelerator, the hardware designer does not care about these tasks. The accelerator generates virtual memory addresses and the system provides translation and synchronisation: the user code becomes portable.

```
-- Initialisation
-- with platform-dependent addresses
ptr_a <= LMEM_BASE;
ptr_b <= LMEM_BASE + LMEM_SIZE/2;
-- Computation
cycle 1:
  -- partition of A[]
  LMEM_PHYADDR <= ptr_a;
  LMEM_ACCESS <= '1';
  LMEM_WR <= '0';
cycle 2:
  reg_a <= DATAIN;
cycle 3:
  reg_b := IDEA(reg_a);
  -- an element in a partition of B[]
  LMEM_PHYADDR <= ptr_b;
  DATAOUT <= reg_b;
  LMEM_ACCESS <= '1';
  LMEM_WR <= '1';
  ptr_{a,b} <= ptr_{a,b} + 1;
  if (ptr_b = LMEM_BASE + LMEM_SIZE) then
  -- finished for a data chunk
    partial_finish;
  else
    cycle 1;
  end if;
```

(a)

```
-- Initialisation with runtime-dependent
-- virtual memory pointers
ptr_a <= A;
ptr_b <= B;
i := 0;
-- Computation
cycle 1:
  -- object A[]
  VMEM_VIRTADDR <= ptr_a;
  VMEM_ACCESS <= '1';
  VMEM_WR <= '0';
cycle 2:
  reg_a <= DATAIN;
cycle 3:
  reg_b := IDEA(reg_a);
  -- any element in B[]
  VMEM_VIRTADDR <= ptr_b;
  DATAOUT <= reg_b;
  VMEM_ACCESS <= '1';
  VMEM_WR <= '1';
  ptr_{a,b},i <= ptr_{a,b},i + 1;
  if (i = n64) then
  -- finished for the entire vectors
    finish;
  else
    cycle 1;
  end if;
```

(b)

Figure 2.14: Platform-dependent (a) and portable (b) VHDL-like code of the hardware accelerator. The platform dependent code reflects the limited size of the local memory and uses physical memory addresses. The portable code uses virtual memory addresses with no notion where the data actually reside.

System Requirements. To achieve our goal of having unified process context for codesigned applications, we need (1) system hardware support for user hardware invocation, virtual address translation, and memory coherence enforcement, and (2) system software support for steering these activities and enabling the interoperation with user software. In general-purpose computer systems, memory management units (MMUs) and cache controllers [33, 56] represent the system hardware responsible for virtual address translation and transparent and coherent memory hierarchy. The OS kernel, in turn, represents the system software responsible for managing the hardware and for providing the process model of computation to user software. It is the responsibility of system designers to decide how to partition—between system software and system hardware—the task of providing the process and memory abstractions. Researchers have explored different approaches [95], many of them being demonstrated in practice [1, 16, 74].

We choose a mixed software and hardware scheme that employs (1) a hardware translation engine (the *user hardWare memory Management Unit*—WMU—from Figure 2.12) and (2) an OS extension. The scheme slightly trades off performance for applicability to a wide range of reconfigurable SoCs; there are no hardware requirements regarding the system-bus capability to support memory coherence. Our approach is not limited to reconfigurable SoCs, although we primarily target these devices in our case studies. Following the state-of-the-art (in Chapter 3), we present details of our architecture, and demonstrate its prevailing benefits and limited drawbacks in Chapters 4, 5, 6, and 7. We also show that performance is not significantly affected, despite the inherent overhead of our scheme.

Chapter 3

State of the Art

> Grâce à l'art, au lieu de voir un seul monde, le nôtre, nous le voyons se multiplier et autant qu'il y a d'artistes originaux, autant nous avons de mondes à notre disposition, plus différents les uns des autres que ceux qui roulent dans l'infini, et bien des siècles après qu'est éteint le foyer dont il émanait, qu'il s'appelât Rembrandt ou Ver Meer, nous envoient encore leur rayon spécial.
>
> —Marcel Proust, *Le Temps Retrouvé*

BOTH research and industry have tackled a large number of issues from hardware and software codesign, bus wrappers and memory subsystems for embedded computing, programming paradigms and OS-support for reconfigurable computing, and general hardware and software interfacing. In this chapter, we present the state-of-the-art and distinguish our contribution from related works.

We first show in Section 3.1 how our research relates to industry standardisation efforts for providing IP-level design reuse and portability, and to research on memory wrappers and memory subsystems for embedded computing. Then, in Section 3.2, we compare our work to major approaches of reconfigurable computing for extending standard CPUs with application-specific hardware accelerators. In the same section we also discuss related work on parallel programming paradigms for reconfigurable computing, and we present the status of the research on OSs for reconfigurable computing. In Section 3.3, we show existing approaches also offering portable and seamless interfacing between software and hardware. Section 3.4 presents related work that motivated our user-transparent dynamic optimisation technique. Finally, Section 3.5 compares existing work on high-level synthesis with our unrestricted automated approach.

3.1 Memory Wrappers and Subsystems

The increasing need for IP-reuse and component-based system design motivates abundant industrial and research activities regarding memory abstractions and communication interfaces. On the industry side, many standardisation efforts have facilitated interconnecting IP blocks that come from different sources. For example, AMBA [8] and CoreConnect [59] are some well-known industry standards for on-chip bus interconnections. Going one step further, the *Virtual Component Interface*

(VCI) is a standard [71] that separates bus-specific interfacing logic from the internal functionality of IP blocks; in this way, it hides the details of the underlying bus interface from the IP designer. On the research side, many researchers have addressed memory wrappers and transparent bus interconnections. For example, Lyonnard et al. [78] and Gharsalli et al. [49] propose automatic generation of application-specific interfaces and memory wrappers for IP designs. Similarly, Lee and Bergmann [69] introduce an interfacing layer that automates connecting IP designs to a wide variety of interface architectures. The main originality of our idea, with respect to the standardisation efforts and wrapper-related works, is not in the 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 hardware accelerator—that is the implication of the OS in the process.

Extensive literature exists on the design and allocation of application-specific memory systems, typically for ASIC end SoC designs (e.g., Catthoor et al. [28] discuss methodologies for custom memory management, while Panda, Dutt, and Nicolau [91] survey different memory issues for SoCs). In most cases, the existing approaches are compiler-based static techniques consisting of (1) design methodologies for customising the ASIC memory hierarchy for specific applications and (2) software transformations to exploit better a given memory hierarchy. The latter techniques are independent from the actual interface we handle, and their proficient use can enhance the design of hardware accelerators, including virtual-memory-enabled ones. In contrast to well-established static techniques, a few works have a dynamic flavour. For example, the work of Leeman et al. [70] on refining dynamic memory management for embedded systems is fully complementary to our approach. Since our system layer—providing the unified memory for software and hardware—is dynamic in its nature, one could consider using Leeman's methodology to improve the dynamic behaviour of our memory allocation process.

In the area of memory systems for reconfigurable computing, Herz et al. [57] have studied the generation of optimal access patterns for coprocessors within SoC architectures; their focus is not in abstraction from architectural details and portability, as it is the case in this thesis. Although we only use simple access patterns for validation, hardware designers can use any access pattern in conjunction with the unified memory. In this way, their address generation techniques are complementary to our work.

3.2 Reconfigurable Computing

In this section, we first relate our architecture with two major architectural approaches in reconfigurable computing that blend temporal and spatial computation. Then, we compare our approach to parallel execution in reconfigurable SoCs with recent developments in multithreading for reconfigurable applications. Finally, we survey existing OS extensions—complementary to ours—that address system software support for reconfigurable applications.

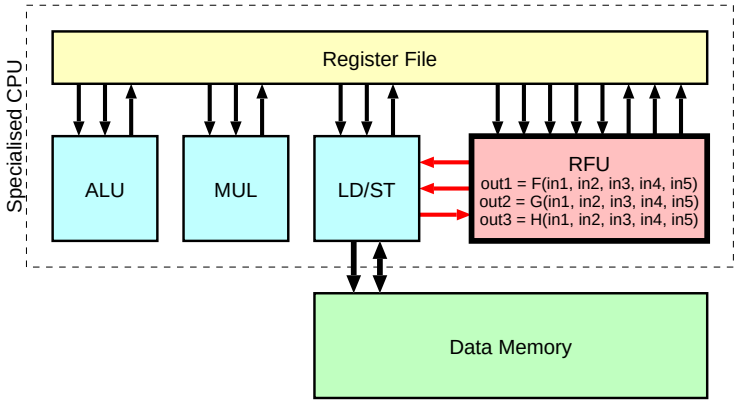


Figure 3.1: Specialised CPU extended with *Reconfigurable Functional Unit* (RFU). The RFU implements a special instruction built by collapsing multiple ISA instructions into the complex one.

3.2.1 Reconfigurable Accelerator Integration

Two of the most common approaches that mix temporal computation within a CPU and spatial computation within reconfigurable hardware are: (1) fine-grained, specialising the CPU data-path with an application-specific reconfigurable functional unit (RFU in Figure 3.1); and (2) coarse-grained, specialising the application execution off the CPU, with an application-specific reconfigurable accelerator (acting like a coprocessor in Figure 3.2).

Past research has demonstrated the feasibility of extending a CPU data-path with a reconfigurable functional unit (e.g., [10, 98, 132]). The approach is also well-known and studied in the ASIC world [58]. Such extensions demand creating new ISA instructions for the given CPU. Apart from the need for automation when creating the new instructions, using the extensions effectively requires some compiler modifications (as, for example, Ye et al. [132] demonstrate). Our research does not address this approach, as we rather target reconfigurable SoCs with nonextensible CPU cores.

Closer to our concerns are systems adopting the coarse-grained approach—with hardware accelerators acting like coprocessors. We have shown in Section 2.5 our target model of computation mixing temporal and spatial computation. Other researchers have also considered using hardware accelerators acting like coprocessors (e.g., Hauser and Wawrzynek [55] propose GARP—a MIPS processor with a reconfigurable coprocessor; Budiu et al. [23] introduce ASH—a coprocessor-based implementation of spatial computation; Vassiliadis et al. [118] present MOLEN—a polymorphic processor incorporating general-purpose and custom computing). In contrast to our work, system-level management of memory hierarchy is out of their concerns. For example, Budiu et al. [23] put aside maintaining a coherent view of memory and the OS integration, and rather investigate compilation and synthesis issues. Callahan, Hauser, and Wawrzynek [25], like Vassiliadis et al. [118], mostly rely on resolving the interfacing through compiler technology. In contrast, we solve the interfacing at the system level—no need for compiler customisation—by providing to user hardware the same abstractions that user software already has. None of

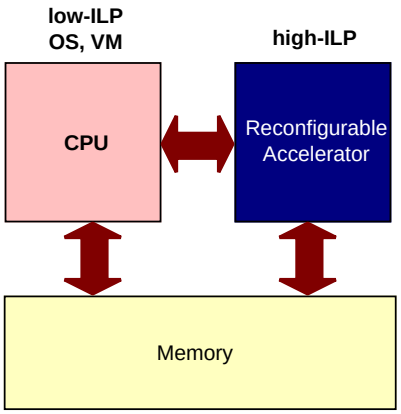


Figure 3.2: CPU used in tandem with coprocessor-like reconfigurable accelerator. The accelerator is loosely coupled with the CPU. It implements a special function or a critical-code section at the coarse-grained level.

these authors addresses the integration of user software and user hardware within an OS process as we do.

3.2.2 Parallel Execution

Numerous research groups have investigated models of parallel execution for reconfigurable computing systems [51]. For example, Brebner et al. [22] propose a hardware-centric parallel programming model targeting mainly the design of networking applications. Closer to ours is the work of Andrews et al. [5, 6]. They have suggested a hybrid SW/HW architecture that enables a multithreaded programming model called *hthreads* (Figure 3.3 shows components of a system using hthreads). Beside the platform-independent compilation, the same authors concentrate on efficient implementation in hardware of primitives for thread synchronisation and scheduling. As Figure 3.3 illustrates, there are special hardware units for management (*Thread Manager*, *Thread Scheduler*, and *CBIS—CPU Bypass Interrupt Scheduler*) and synchronisation (*Mutexes* and *Conditional Variables*), but there is no virtual memory support (hardware threads generate physical addresses of *Shared Memory* from Figure 3.3). In contrast to the described approach, our unified OS process provides a multithreaded programming paradigm with a common virtual memory for software and hardware threads: the programming paradigm it offers is exactly the same as in the software-only case.

Nollet et al. [89] have proposed another model for parallel execution in reconfigurable SoCs. They introduce at the OS level a *Hardware Abstraction Layer* (HAL) responsible for communication between software and hardware (e.g., in Figure 3.4, the communication between the software task P2 and the hardware task PC goes through the HAL). Like our virtualisation layer in the OS, the HAL consists of software and hardware components. Unlike our unified OS process, their solution assumes a specific communication scheme based on message passing. On the hardware side, tasks generate no memory addresses but send messages through a message-passing interface; on the software side, tasks rely on the message-passing

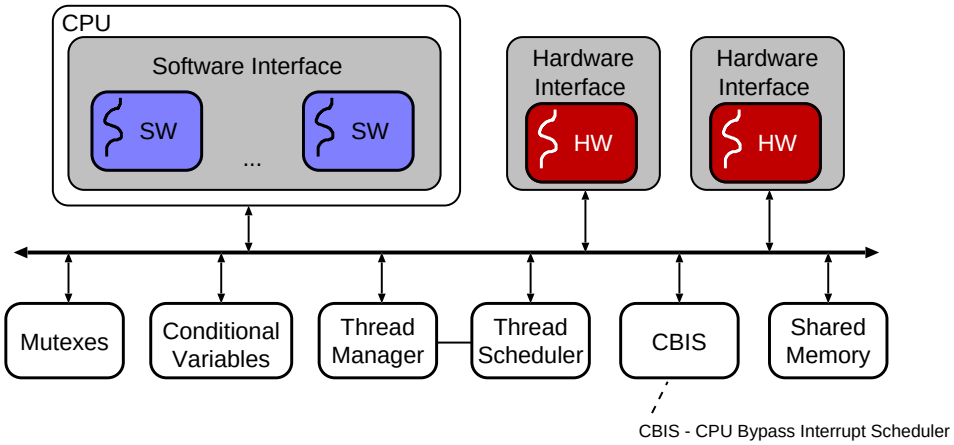


Figure 3.3: System using *hthreads* for codesigned applications. The system hardware provides mechanisms to speed up thread synchronisation (Mutexes and Conditional Variables) and management (Thread Manager, Thread Scheduler and CBIS). There is no virtual memory support.

Application Programming Interface (API). Our scheme is more general and assumes no specific API but allows both user software and user hardware—being within the context of the same OS process—to access and share any location of practically-unlimited virtual memory.

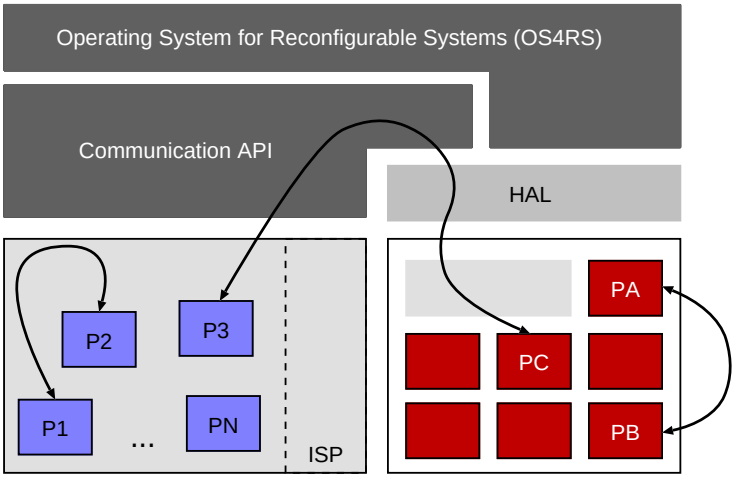


Figure 3.4: Inter-task communication in the OS4RS system. Software and hardware tasks rely on a message passing API to communicate. The communication API and the *Hardware Abstraction Layer* represent the system layer components (implemented in software and hardware) responsible for interfacing.

The Cell processor [42, 94], proposed by an industrial alliance [54, 65], provides high-performance parallel processing on a single chip, with a high-level of flexibility and configurability. For example, the processor can dynamically link its coarse-grained processing units to form a temporary pipeline tailored for a particular application. Figure 3.5 shows the basic architecture of the Cell processor. It contains the main host processor (a 64-bit version of the PowerPC processor) and

eight Synergistic Processing Elements (SPEs) for power-efficient (typical superscalar issues are delegated to software) and high-performance computing. The distinction between general-purpose (PPE) and special computation blocks (SPEs), together with their deployment and bus-based interconnection, relates the architecture of the Cell processor with the architecture of our system—a general-purpose CPU supporting execution of multiple application-specific hardware accelerators.

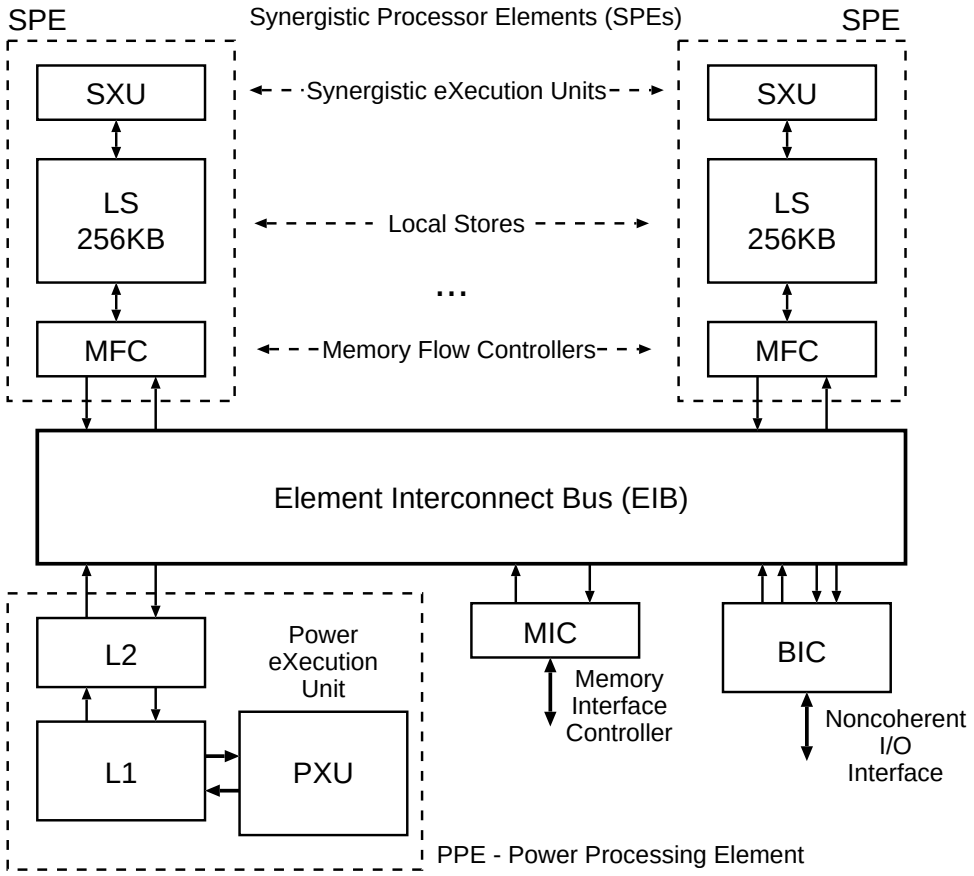


Figure 3.5: Block diagram of the Cell processor. The processor consists of Power Processing Element (PPE—based on a PowerPC processor) interconnected with eight Synergistic Processing Elements (SPEs). The SPEs access their own local address space (local stores), not belonging to the system address space. Local address space is untranslated, unguarded, and noncoherent. The software has to manage local stores explicitly by programming the Memory Flow Controllers (MFCs). The MFCs can access the local stores or the system (off-chip) memory to reach the required data.

Each SPE [42] is a four-way SIMD processor [44] programmed in C or C++ that consists of a processing unit (SXU in Figure 3.5), local memory (named local store), and the *Memory Flow Controller* (MFC). In contrast to our architecture, the local memories of SPEs are out of the system address space, addressable only by their local SPE or MFC; the software is entirely responsible for the local memory management (by programming the appropriate MFCs). The approach trades programming simplicity for achieving better performance. In our case, we delegate transfer activities to the system level and provide the unified address space for all

accelerators. We also address improving performance of memory transfers, but in a user transparent fashion (the end user is completely screened), through runtime optimisations such as prefetching.

3.2.3 Managing FPGA Resources

Different research groups have recognised the significance of managing FPGA real estate. Reconfigurable logic is a computational resource that codesigned applications can use for accelerating their execution. A resource manager determines how the applications running at the same time use the reconfigurable resource. It also makes sure to load an accelerator configuration in the reconfigurable logic before a particular application refers to it. In his work on a virtual hardware OS, Brebner [20, 21] defines swappable logic units and proposes delegating their management to an OS. The ultimate goal is to screen the end user from the problems introduced by the finite amount of available reconfigurable logic. Accordingly, Dales [34] has shown another example of reconfigurable hardware virtualisation. His proposal describes an architecture relying on the OS to share dynamically the reconfigurable logic among applications. In his case, the OS supports mapping from the virtual to the physical resource. The type of virtualisation we introduce addresses the interfacing between the CPU and the reconfigurable accelerators rather than the reconfigurable lattice itself.

Dynamic partitioning of the reconfigurable logic and configuration scheduling are advanced tasks of the resource manager. Walder and Platzner [123] have investigated online scheduling for block-partitioned reconfigurable devices. Similarly, Caspi et al. [26] have examined dynamic scheduling for a stream-oriented computation model. Finally, Fu and Compton [45] have proposed, for a computation model closest to ours, extending an OS with support for scheduling and runtime binding of computation kernels implemented in reconfigurable hardware. In our case, we build the execution environment for reconfigurable hardware accelerators within the unified OS process, and we do not address partitioning and scheduling. Although, for the moment, we assume that the configuration of the FPGA does not change during application lifetime, our future OS extensions may incorporate sophisticated algorithms proposed by the related work. The existing approaches presented in this section are orthogonal and complementary to our work—future systems may have to implement both.

3.3 Hardware and Software Interfacing

Typical solutions [72, 84, 112] for hardware accelerator interfacing with software expose the communication to the programmer through a limited-size interface memory mapped to the user space. For example, Leong et al. [72] use a memory slot interface for communication between the CPU and the hardware accelerator. On the other side, our approach completely liberates the programmer of communication details, since the OS does memory transfers implicitly hiding them from the end user. Caspi et al. [26, 27] and Nollet et al. [89] introduce more sophisticated

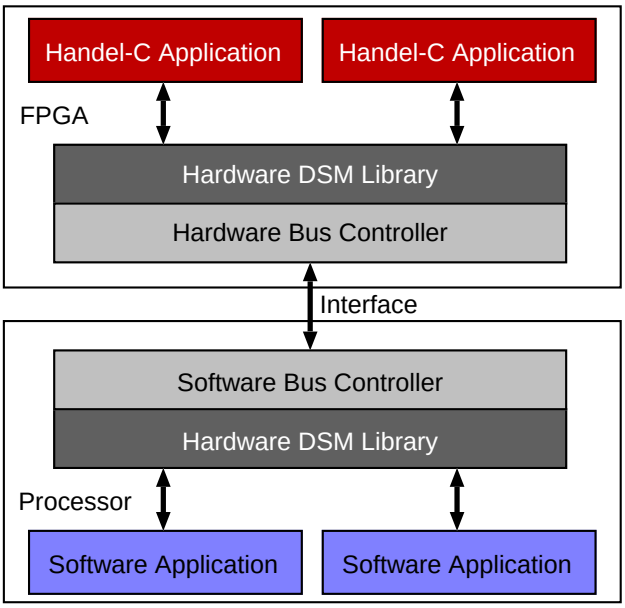


Figure 3.6: Data streaming manager (DSM) for portable codesign. The DSM hides platform details from both software and hardware. It provides stream-oriented ports for communication.

interfacing schemes—providing also parallel execution—than the typical existing solutions offer: (1) Caspi et al. suggest a stream-based programming paradigm for using multiple hardware nodes logically organised in a data flow; (2) Nollet et al. propose a task communication scheme based on message passing (already discussed in Section 3.2.2). These approaches simplify the communication with hardware accelerators but, assuming a specific API, they are less general than our approach.

An industrial solution from Celoxica [30, 114] offers software API (based on the C language) and hardware API (based on the Handel-C language) supported by an intermediate system layer called *Data Streaming Manager* (DSM from Figure 3.6). The DSM provides platform-independent and stream-oriented communication between software and hardware. The approach demands using API-specific data types (the DSM buffers) and communication primitives (the DSM port read and write operations). The data transfers are exposed to the programmer and the coprocessor memory accesses are limited to the sequential access pattern. In our approach, the coprocessor hardware can generate any pattern of virtual memory addresses. Furthermore, our virtual-memory scheme completely screens the programmer from data transfers. It is the ultimate role of the unified OS process to provide this transparency.

A research by Mukherjee et al. [86, 87] related to general-purpose computing introduces coherent network interfaces—peripherals capable of accessing the main memory through a cache that supports a cache-coherence protocol (as Figure 3.7 shows). Their goal is to improve performance of application-specific communication and to simplify programming. Having shared memory—enforced by a cache-coherence protocol—between the CPU and the network interface avoids the need for explicit data transfers, even if fast transfer mechanisms (e.g., Blumrich et al. [18]

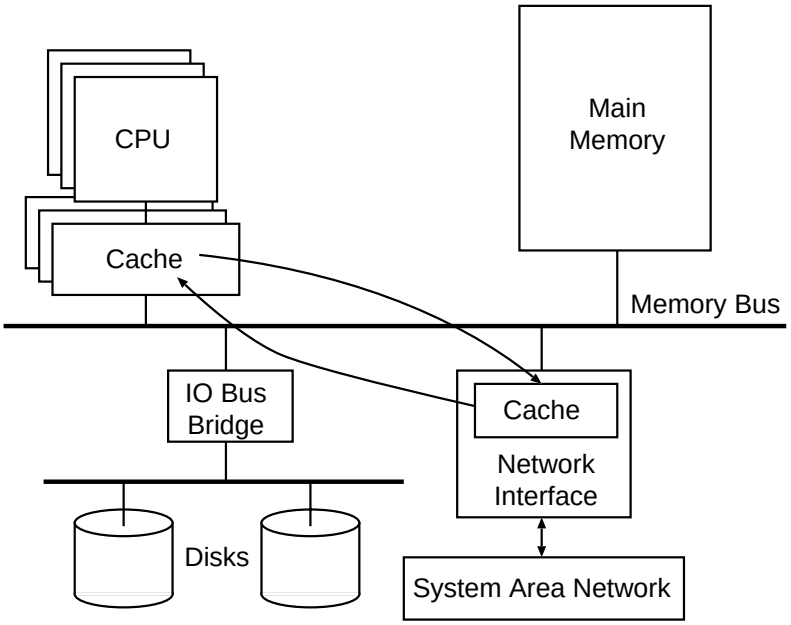


Figure 3.7: Network interface with cache on memory bus. This configuration provides improved performance and simplified programming. The memory bus supports cache coherence.

and Markatos and Katevenis [80] have developed mechanisms to use DMAs from user programs) are available at the user level. Mukherjee and Hill [87] plead—similarly to what we do for hardware accelerators—in favour of considering network interfaces as standard equipment, not as peripheral add-ons. We extend their approach, by proposing a general scheme that distinguishes user hardware (having explicit semantic links with a particular user application) from system hardware (such as mass storage, network interfaces, reconfigurable logic, being implicitly available to all user applications). Our approach also eliminates restrictions regarding virtual-to-physical address translation that the work by Mukherjee et al. assumes. For example, they assume that the operating system allocates pages containing network-interface data structures in physically contiguous memory; on the contrary, our approach has no such restrictions.

Continuing the research on coherent network interfaces, Schoinas and Hill [103] have investigated address translation mechanisms in network interfaces. They find that network interfaces need not have hardware lookup structures, as software schemes are sufficient. However, in our approach we have no specific hardware accelerator a priori in mind; thus, we decide to use hardware lookup structures (e.g., the WMU we have shown in Section 2.5 does the translation locally within a TLB equivalent) to cover general cases. Later on in this thesis we show that the cost is affordable. Having hardware accelerators with caches on the CPU memory bus—as is the case with the coherent network interfaces—would definitely be beneficial; our mixed software and hardware approach introduced in Chapter 4 incurs larger overhead because of the page-level granularity (we analyse the overhead in Section 5.2); nevertheless, we have chosen such an approach since it is easily applicable to reconfigurable SoCs currently available on the market.

3.4 Prefetching Techniques

We employ a dynamic prefetching technique (explained in Section 4.4) in the system abstraction layer supporting unified memory for user software and user hardware. The technique is motivated by extensive related work on prefetching. In this section, we mention just few principal researches on prefetching that is applied, or could be applied, to our approach.

Hardware and software prefetching techniques were originally developed for cache memories to support different memory access patterns. Jouppi [63] has introduced stream buffers, as an extension of tag-based prefetching, for simple but effective prefetching for sequential memory accesses. Following Jouppi's work, Palacharla and Kessler [90] have proposed several enhancements for stream buffers. Other techniques exist that cover nonsequential memory accesses (e.g., Roth, Moshovos, and Sohi [99] address recursive techniques that can fit pointer-chasing memory accesses, while Solihin, Lee, and Torrellas [109] show correlation-based prefetching for user-level threads). In the field of reconfigurable computing, Lange and Koch [66] have used hardware prefetching techniques for configurable processors.

Apart from caching, researchers have used prefetching techniques for efficient virtual memory management. For example, Saulsbury, Dahlgren, and Stenström have relied on a hardware technique to speculatively preload the TLB and avoid page faults. Similarly, Bala, Kaashoek, and Weihl [14] have investigated a software technique to prefetch virtual memory pages for user applications.

The prefetching technique that we use within our unified OS process is in its essence a dynamic software technique with limited hardware support. It is motivated—but not limited to—by Jouppi's stream buffers. We have implemented the dynamic prefetching in such a way that future extensions of our architecture can easily apply the techniques mentioned in this section. The strongest point of our dynamic prefetching is the transparency: neither user software nor user hardware are aware of its presence, the optimisation is completely done at the system level.

3.5 High-level Language Synthesis

A considerable amount of research on synthesis is available in the hardware and software codesign literature [37]. We limit the comparison of our unrestricted synthesis approach only to published results on high-level language synthesis; closer to our case study shown in Section 6.3, we also consider existing approaches of high-level synthesis for virtual machines.

Hardware synthesis from high-level programming languages, especially for dialects of C and C++, has been a challenging topic for a long time [36, 40]. For synthesising hardware from languages related to C and C++, researchers, as well as the industry, have adopted two major approaches: (1) extending the original language [29, 46, 53, 131] and making it closer to hardware description languages (e.g., adding language constructs such as concurrency, reactivity, variable bitwidth, special data types); and (2) restricting the original language [47, 92, 106, 108] to ease the mapping to the register transfer level (e.g., excluding language constructs such as

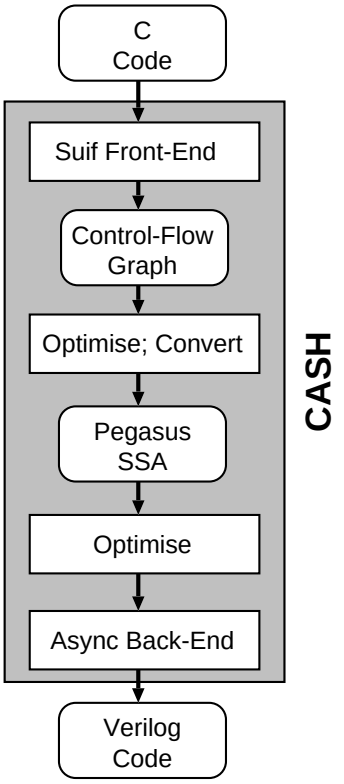


Figure 3.8: Compiler for Application-Specific Hardware (CASH). Using a *Static Single Assignment* (SSA) form and an asynchronous back end producing Verilog code, the compiler synthesises application-specific accelerators from the input C code.

pointers, dynamic allocation, function calls, and recursion). Having our unified OS process with unified virtual memory and execution transfers, there is no restriction on the input code; practically, a synthesis flow based on our unified OS process can treat any high-level language construct. We do not consider special hardware description languages originating from standard programming languages; we suppose that the synthesiser is capable of extracting enough parallelism from the original code; nevertheless, if the programmer decides to write explicitly-parallel code (e.g., using a multithreaded programming paradigm), a synthesis flow based on the unified OS process with multithreading support (we present the multithreading extension to the unified process in Chapter 7) will be able to handle it as well.

In their research on spatial computation, Budiu et al. [23] use a computation model closest to ours. Assuming an asynchronous coprocessor-based model with a monolithic memory shared with the CPU, they develop a compiler flow for generating application-specific hardware from C-program kernels. On average, the Mediabench applications [68] run slower on hardware generated by their flow than on a superscalar machine [24]. However, they report significant power savings (up to three orders of magnitude). Figure 3.8 shows their CASH (*Compiler for Application-Specific Hardware*) synthesis flow. The flow uses Suif [127] as a front-end to transform the input C program into a control-flow graph. The next pass of the flow converts the control-flow graph to an extended SSA (*Static Single Assignment*) in-

termediate representation (Pegasus in Figure 3.8). After the final optimisations, the back-end produces the Verilog output for an asynchronous circuit corresponding to the input C code. Although Budiu et al. ease the translation of C into hardware by assuming a memory hierarchy like in general-purpose systems, the circuits generated by CASH cannot handle system calls. In our case, such limitation does not exist. The unified OS process allows user hardware to invoke system services (as Section 6.2 shows).

We also consider research on automated synthesis of memory subsystems to be related to our work. For example, Wutack et al. [129] use a compile-time analysis to generate distributed memory structures with custom memory managers. On the contrary, our solution is a general-purpose and a centralised one: memory management is performed by the OS at runtime and the virtual memory space is unique for all accelerators. This, however, does not exclude possible future implementations of our system with distributed management of the virtual memory.

The unified OS process enables synthesis of high-level language concepts like memory allocation, function calls, and recursion [122]. We implement a synthesis flow for virtual machines as a case study. Researchers have used software [115] and hardware [67, 73] approaches to accelerate code execution in virtual machines. In the case of *Java Virtual Machines* (JVMs), industrial solutions exist that support limited or complete subset of JVM instructions (Java bytecode) directly in hardware [73] (e.g., PicoJava from Sun, Jazelle from ARM, or DeCaf from Aurora VLSI). The approach improves execution times of Java programs, but it is highly dependent on the used JVM. Other authors have addressed the optimisation problem at a higher level, by implementing Java methods in reconfigurable hardware [43, 67]. However, these solutions are usually dependent on the underlying reconfigurable platform and the used JVM. In contrast to both approaches, our solution relies on the unified OS process for codesigned applications that provides portability and seamless interfacing of reconfigurable accelerators (critical Java bytecode methods migrated to hardware). Our approach can be used with any JVM which conforms to the Java platform standard. Moreover, our extension to the JVM execution environment allows running of these accelerators without any modification of either compiled Java bytecode or Java source.

Chapter 4

Virtual Memory for Hardware Accelerators

> And now for something completely different.
>
> —The Announcer, *Monthy Python's Flying Circus*
>
> Play it again, Sam.
>
> —Rick Blaine (what he never said), *Casablanca*

IN this chapter, we show which system software and system hardware extensions are necessary to provide unified virtual memory for codesigned applications. As already mentioned in Chapter 2, we choose a mixed software-and-hardware approach that enables hardware accelerators sharing the virtual memory with user software. We present our architecture (Section 4.1) and we explain its basic components (system hardware in Section 4.2 and system software in Section 4.3). In Section 4.4, we show simple extensions to our architecture that support system-level runtime optimisations for codesigned applications.

4.1 System Architecture

Our approach primarily relies on system software and system hardware extensions. Figure 4.1 shows the system support required by user software and user hardware for cohabiting within the same user process. The *Memory Management Unit (MMU)* translates memory addresses generated by the user software—executed on the CPU—from virtual addresses to physical addresses of the main memory. The *Virtual Memory Manager (VMM)* of the OS manages and controls the translation.

We add an MMU equivalent (the WMU introduced in Chapter 2) as a system hardware support for the translation of virtual memory addresses generated by the user hardware—executed on the hardware accelerator. We also extend the OS with a virtual memory manager for hardware accelerators (*Virtual Memory for user hardWare*—VMW—manager in Figure 4.1). It manages the translation done by the WMU (similarly as the VMM does with the MMU) and provides a system call interface between software and hardware.

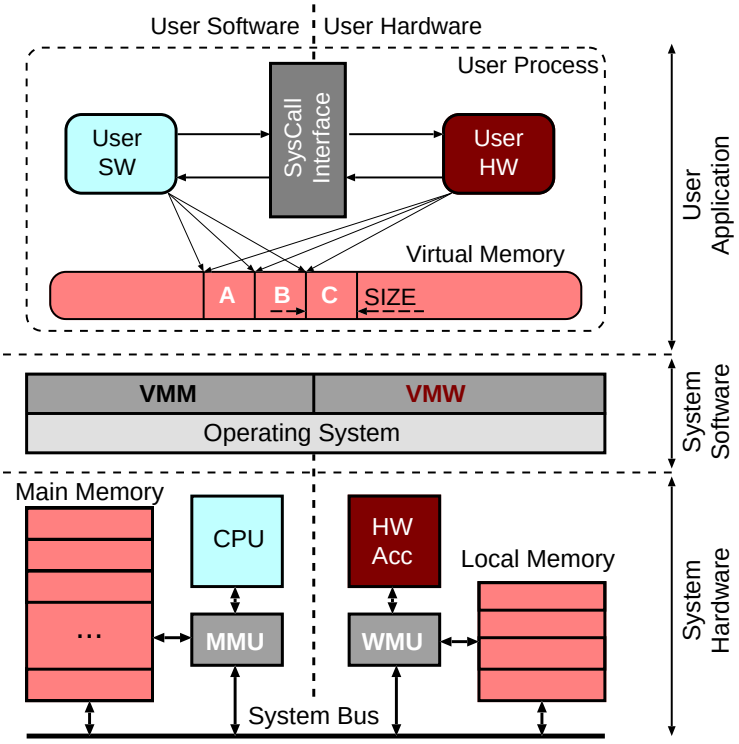


Figure 4.1: Support architecture for unified memory abstraction at the application level. The OS provides the memory abstraction and manages the translation done by the MMU and the WMU.

It is worth noticing that the system bus interconnecting the CPU and the hardware accelerator does not have any support for memory coherence (for the sake of simplicity, Figure 4.1 does not show the real memory hierarchy of the CPU—the caching is, of course, present). We completely rely on the OS to perform this task; although this fact may affect system performance (as discussed in the analysis in Section 5.2), the approach is applicable to a wide range of reconfigurable SoCs. Furthermore, the experimental results (in Section 5.3) show that the introduced overhead is acceptable.

4.2 Hardware Interface to Virtual Memory

The WMU translates virtual addresses generated by user hardware to physical addresses of an on-chip, local memory, acting as a software managed cache (shown in Figure 4.1). Apart the translation, the WMU decouples the design of application-specific hardware accelerators from the host platform: it defines a standardised hardware interface for the virtual memory-enabled hardware accelerators. The WMU also supports parameter exchange between software and hardware, and provides a way for user hardware to call back software and invoke system calls (as we demonstrate in Chapter 6).

The WMU—being a platform specific component, as its interfacing depends on the system bus specification of a particular platform and its implementation

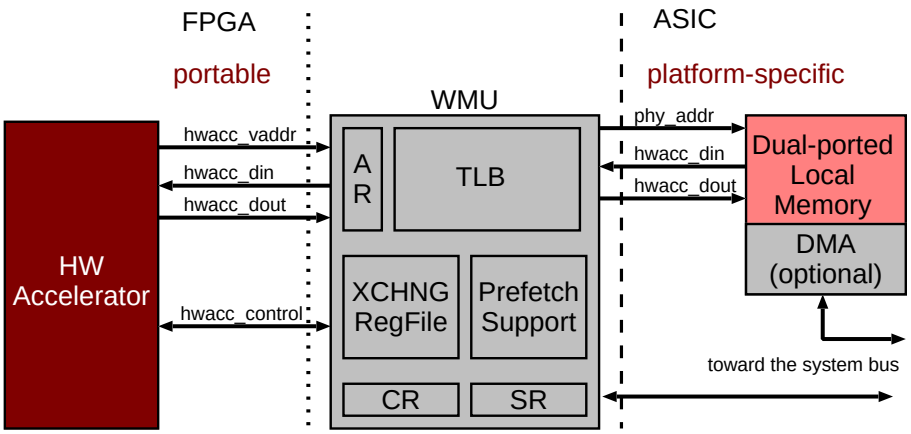


Figure 4.2: WMU structure and WMU interface to user hardware, local memory, and system. The WMU contains a *Translation Lookaside Buffer* (TLB), an exchange register file (XCHNG RegFile), status (SR), control (CR), and address (AR) registers, and a block for speculative prefetching (Prefetch Support). The WMU interface toward the accelerator does not change across different platforms. The interface toward the rest of the system is platform-specific.

depends on the underlying FPGA architecture—has to be ported once per target platform. For the moment, we assume having only one WMU per application. We explore possibilities of having multiple WMUs in Chapter 7. Multiple hardware accelerators could use the WMU (by reconfiguring the FPGA), but only one at a time.

Figure 4.2 shows the structure of the WMU with its interfaces toward the hardware accelerator and toward the rest of the system. A virtual memory-enabled hardware accelerator is interconnected to the WMU. The interface consists of virtual address lines (`hwacc_vaddr`), data lines (`hwacc_din` and `hwacc_dout`), and control lines (`hwacc_control`). Platform-specific signals connect the WMU with the rest of the system. The presence of the *Translation Lookaside Buffer (TLB)* inside the WMU emphasises its similarity with a conventional MMU [56]. The TLB translates the upper part of the address—its most significant bits—issued by the accelerator to a physical page number of the local memory. The WMU can support multiple operation modes, i.e., different local memory sizes, page sizes, and number of pages of the memory. Limitations of the FPGA technology (e.g., lower clock rates than full-custom integrated circuits having the same feature size) and the design methodologies different than in the case of ASICs (e.g., inserting intermediate registers between logic levels facilitates simpler routing and achieving higher clock frequencies) impact the performance of our TLB design. The TLB does the translation in multiple cycles. Appendix A gives more details about the WMU interface showing which signals the hardware designer has to implement when designing a WMU-compliant hardware accelerator.

Apart from the usual control and status registers (`CR`, `SR`), the WMU contains an exchange register file (for parameter passing between software and hardware, and vice versa), prefetching support logic (discussed in Section 4.4), and the address

register (AR) containing the most recent address coming from the accelerator. The WMU translates virtual addresses requested by the accelerator (similarly as the MMU does for the user application running on the CPU) to physical addresses of the local memory divided into pages. If the hardware accelerator tries accessing a page not present in the local memory, the WMU generates an interrupt and requests the OS service. On this event, the hardware accelerator is stalled (as the CPU may be stalled on a cache miss). While the accelerator is stalled, the VMW manager (1) reads the address register to find out the address that generated the fault, (2) transfers the corresponding page from the main memory and updates the TLB state in the WMU, and (3) resumes the accelerator. The optional presence of a DMA in the system can speed up the memory transfer and relieve the CPU from copying. Appendix A explains the parameter exchange protocol from the hardware side—the hardware designer has to follow the protocol for getting the correct invocation parameters from software. On the other side, Appendix B gives more details on the parameter exchange protocol from the software programmer viewpoint.

Having the HDL ports of virtual memory-enabled hardware accelerators predefined by the WMU hardware interface (shown in Figure 4.2), the hardware designer writes the accelerator HDL-code (1) to fetch the function parameters (e.g., pointers, data sizes, constants), (2) to access data using virtual memory addresses and perform the computation, and (3) to return back to the software. Such HDL code of the accelerator [121] (1) does not embody any detail related to the memory interfacing, (2) has no limit on the size of the data to process, (3) is not concerned about physical data location. Appendix D shows design-related details of several virtual memory-enabled hardware accelerators.

4.3 OS Extensions for Virtual Memory

The VMW manager provides two functionalities: (1) a system call to transfer the execution from user software to user hardware; (2) management functions to respond to WMU requests. The system call provided to programmers is called `sys_hwacc`. It passes data pointers or other parameters to the hardware, initialises the WMU, launches the accelerator, and puts the calling process in sleep mode. The accelerator processes the data with no concerns about their location in memory—translation of generated addresses is performed by the WMU and the VMW manager. Figure 4.3 shows the system call in practice, on the example of the IDEA cryptography application. Calling this function is to all practical purposes identical to the pure software version (shown in Figure 2.4) and fully agnostic of system-related details. The OS provides transparent dynamic allocation of memory resources (i.e., the shared local memory) between the processor and the accelerator: now the programmer can avoid explicit data movements.

The local memory is logically organised in pages, as in typical virtual memory systems. The data accessed by the hardware 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 local memory at the same time. At every point in time, the memory access pattern of the accelerator deter-

```
/* Virtual memory-enabled accelerator version */
void idea_cipher_cp(IDEA_block *A, IDEA_block *B, int n64) {
  param.params_no = 3;
  param.flags = 0;
  param.p[0] = A;
  param.p[1] = B;
  param.p[2] = n64;
  sys_hwacc(IDEA_HW, &param);
}
```

Figure 4.3: Calling the IDEA hardware accelerator through the *sys_hwacc* system call. The programmer initialises the parameter passing structure and invokes the system call.

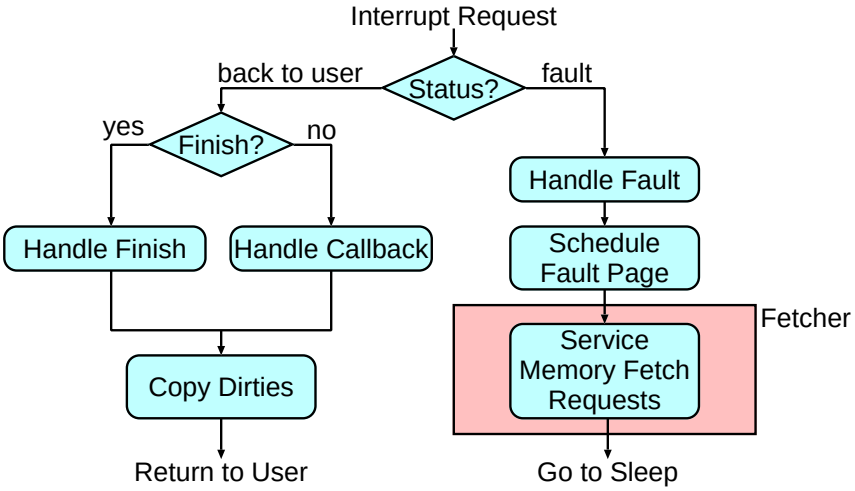


Figure 4.4: Basic VMW interrupt handler. Receiving a back-to-user interrupt, the manager prepares the execution transfer back to user software (either for an accelerator completion or for a software callback). For a fault interrupt, the manager transfers the missing page to the local memory.

mines the occupation of the local memory pages. The accelerator can address any word of the user address space (including stack and heap dynamic memory regions). However, through memory protection policies, the OS prevents the accelerator from accessing forbidden memory regions (such as regions containing software code).

Figure 4.4 shows the basic architecture of the VMW interrupt handler. When an interrupt arrives from the WMU, the manager checks the status register to distinguish the two possible requests: (1) *page fault*—the hardware attempted an access to an address not currently in the local memory; (2) *back to user*—the hardware requests the transfer of the execution back to software.

Page Fault. Since the requested page is not in the local memory, the OS has to rearrange the current memory mapping, in order to resolve the fault. It may happen that all pages of the local memory are already in use. In this case, the VMW manager selects a page for eviction (different replacement policies are possible). The manager ensures that the user memory reflects correctly the state of the local memory; if the page selected for eviction is dirty, its contents are copied back to the user-space memory and the page is anew allocated for the missing data; the missing data page is scheduled for transfer; the manager part called *Fetcher* copies the missing data

from the user-space memory and updates the WMU state; afterward, the OS allows the WMU to restart the translation and lets the accelerator exit from the stalled state and continue the execution.

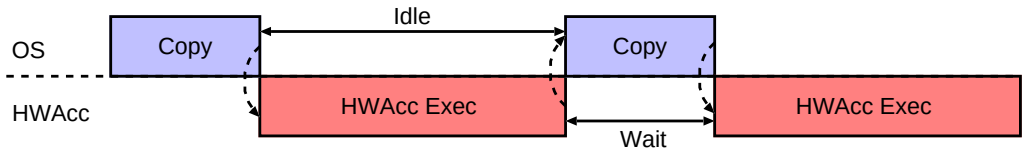
One shall notice a possible overhead—Section 5.2 discusses it further—that may appear for write faults because of page-level granularity of the local memory. The overhead is likely to appear for hardware accelerators with stream-based memory accesses. On a write fault (i.e., the accelerator tries writing to an address not present in the local memory), the OS has to transfer the whole page to the local memory, even if future memory accesses of the accelerator may thoroughly overwrite its contents (i.e., write to each single word of the page). This would mean that the time spent on transferring the page from the main memory may be wasted, just allocating the page in the local memory—without copying the page for a write fault—could suffice; however, such a speculative act would require in-advance knowledge that the accelerator never reads from the page.

Back to User. If such a request arrives from the WMU, the VMW manager checks if the end of the accelerator operation is reported ("`Finish?`" check in Figure 4.4). If true, the manager transfers back the execution to user software, immediately after the `sys_hwacc` system call in the original program (e.g., it would be the function return in Figure 4.3). On the contrary, if the arrived request is for a software callback (i.e., the hardware accelerator invokes a system call or a software function), the manager transfers the execution to the corresponding function, which is out of the normal program execution flow. In Chapter 6, we present how the VMW manager enables the execution transfer. In both cases, before transferring the execution back to software, the VMW manager copies back to the user space all dirty pages currently residing in the local memory. This is a conservative approach—assuming that the callback function might need any page from the local memory–which is not necessarily the most efficient one. A possible optimisation could delay the transfer of a page to the moment when the callback function accesses the page.

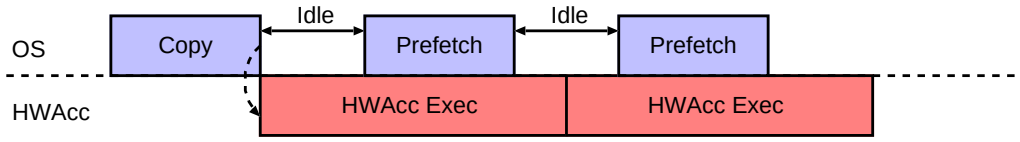
4.4 Dynamic Optimisations in Abstraction Layer

The involvement of system software in execution steering simplifies hardware and software interfacing and unloads software programmers and hardware designers from the burdens of the typical approaches. The presence of the runtime steering in the system software also allows dynamic optimisations that can improve the performance, in a manner completely transparent for users, *without any change in application source code*.

In this section, we present the basic motivation for applying OS-based prefetch techniques. Afterward, we discuss the details of hardware and software requirements to implement a prefetching system for virtual-memory-enabled hardware accelerators. We show the benefits of our approach and showcase several examples of codesigned applications through experiments in Chapter 5.



(a) Basic Execution Timeline



(b) Execution Timeline with Prefetching

Figure 4.5: Execution timeline of virtual-memory-enabled hardware accelerator (a) without and (b) with prefetching. Without prefetching, the VMW manager is idle during the hardware execution, waiting for a memory fault to happen. With prefetching, the VMW manager is active during the hardware execution, trying to anticipate future memory accesses and to eliminate memory faults.

4.4.1 Motivation

The OS is not only limited to providing resource sharing and transparent interfacing; it can survey the execution of the accelerator, optimise communication, and even adapt the interface dynamically. The VMW manager makes such improvements possible without any change in the code of the codesigned application. Although it is intuitively expected that the additional abstraction layer brings overheads, we show in this thesis that it can also lower execution time by taking advantage of run-time information. The dynamic optimisations showcase the strength of delegating the interfacing tasks to the OS.

As opposed to the simple execution model shown in Figure 4.5a, where the main processor is idle during the hardware accelerator busy time, we explore the scenario where the idle time is invested into anticipating and supporting future accelerator execution. With simple hardware support, the OS can predict accelerator memory accesses, schedule prefetches, and thus decrease memory communication latency. While the accelerator is running and the VMW manager is idle, the OS scheduler is free to invoke any available process for execution.

Instead of being idle, the VMW manager could—with a lightweight hardware support in the WMU to detect hardware memory access patterns—survey the execution of the accelerator and anticipate its future requests, thus minimising the number of page faults. Figure 4.5b shows hardware execution time overlapped with the VMW page transfers. With some execution-related information obtained from the WMU, the VMW manager could predict future memory accesses of the accelerator and prefetch virtual memory pages to the local memory accordingly. In the case of correct predictions, the hardware generates no faults: the OS activity completely hides the memory communication latency *without any action on the user side*.

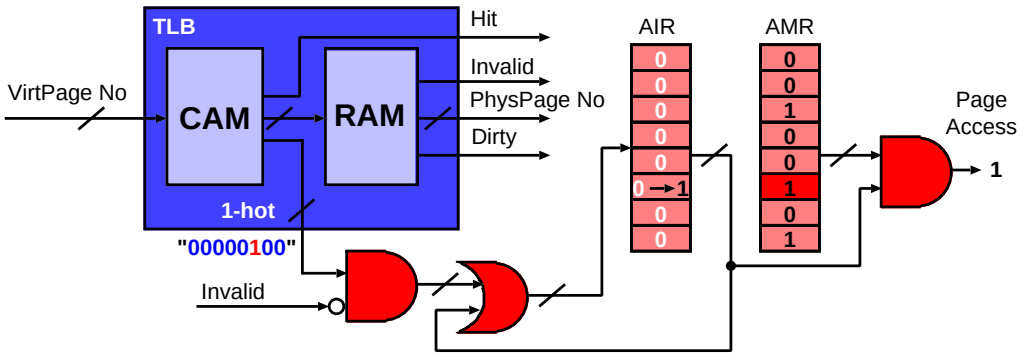


Figure 4.6: Page access detection in the WMU. On a hit in the *Content Addressable Memory (CAM)*, 1-hot bit lines will set the corresponding bit in the *Access Indicator Register (AIR)*. If the mask (*Access Monitor Register—AMR*) allows the access, the WMU raises an interrupt. By reading the AIR register, the OS can determine which pages the accelerator has accessed.

4.4.2 Hardware and Software Extensions

We introduce a simple extension—two 32-bit registers and few tens of logic gates—to the WMU that supports the detection of a page access. Figure 4.6 contains the internal organisation of the WMU related to address translation. The VMW sets the appropriate bits of the *Access Monitor Register (AMR)*, indicating to the WMU which page accesses to report (Table A.1 from Appendix A shows how many pages the WMU typically supports). If there is a match in the *Content Addressable Memory (CAM)*, the 1-hot bit lines are used to set a bit—corresponding to the accessed page—in the *Access Indicator Register (AIR)*. If the mask in the AMR allows (i.e., the VMW manager requests monitoring the access event), the WMU will report the page access by raising an *access* interrupt. While the access interrupt is being handled, there is no need to stop the accelerator; the VMW manager and the accelerator can run in parallel. The OS actions need not be limited to this simple access detection mechanism. A more sophisticated but still reasonably simple hardware can be employed in order to support detection of more complex memory access patterns.

We extend the VMW manager and its interrupt handler (shown in Figure 4.4) to support the prediction of future memory accesses and speculative prefetching (Figure 4.7 shows the extensions). The three main design components of the extended VMW manager [120] are: (1) initialisation and interrupt handling; (2) prediction of future accesses (*Predictor*); (3) fetching of pages from main memory (*Fetcher*). For a *fault* interrupt, after scheduling a transfer of a fault page, the OS invokes the predictor module which predicts future accesses and schedules their transfers. For an *access* interrupt, the OS again invokes the predictor module to validate or confute its past predictions, and schedule future transfers. The predictor uses a simple but effective—in the case of stream-based memory accesses—prefetching policy [120] based on the previous work on stream-buffers for cache memories [63, 90].

The Predictor. It attempts to guess future memory accesses and to schedule page transfers. The only input parameters to the predictor are fault addresses and

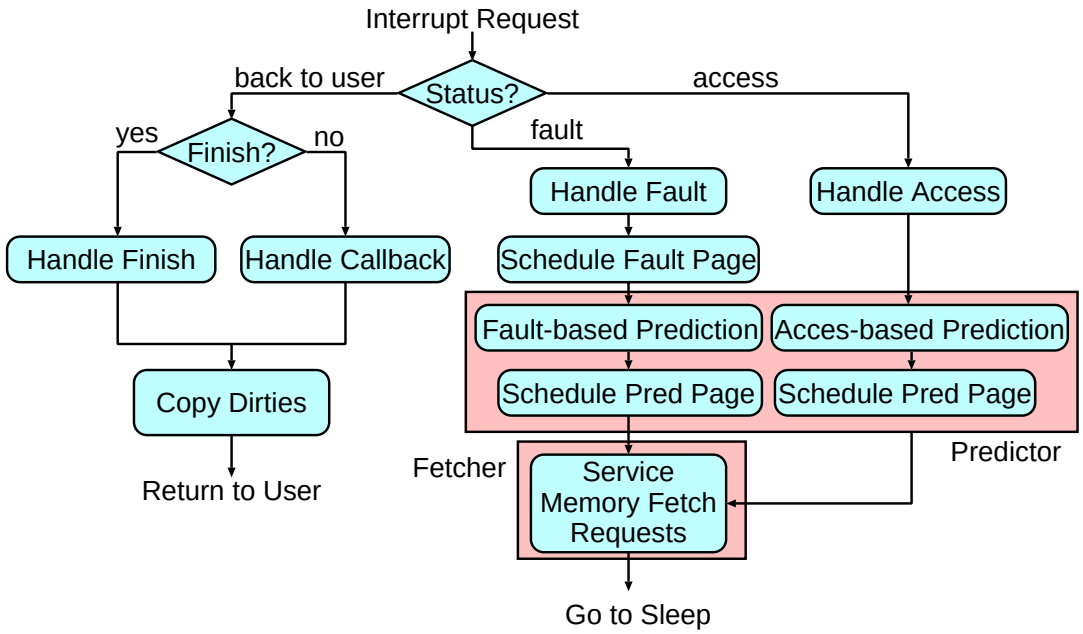


Figure 4.7: Extended VMW manager for speculative transfers. Predictor schedules speculative pages for transfer. Predictions are fault-based (a new stream detected) and access-based (a stream access confirmed). Both initiate speculatively prefetching a new page.

accessed-page numbers—there is no information about the state of the hardware accelerator. The approach is similar to classic prefetching techniques where no information is available about the instructions issued by the main processor [90] but only the addresses on the bus. A simple predictor assumes that for each fault a new stream is detected; thus, it requires a stream buffer allocation (i.e., a circular buffer built of a pair of local memory pages, as Figure 4.8 illustrates) and it schedules a speculative prefetch for the virtual page following the missing one. By setting appropriately the AMR, it ensures that the WMU hardware shall report the first access to the speculatively-loaded page. When the WMU reports the access, the interrupt handler invokes the predictor again and, with this information confirming

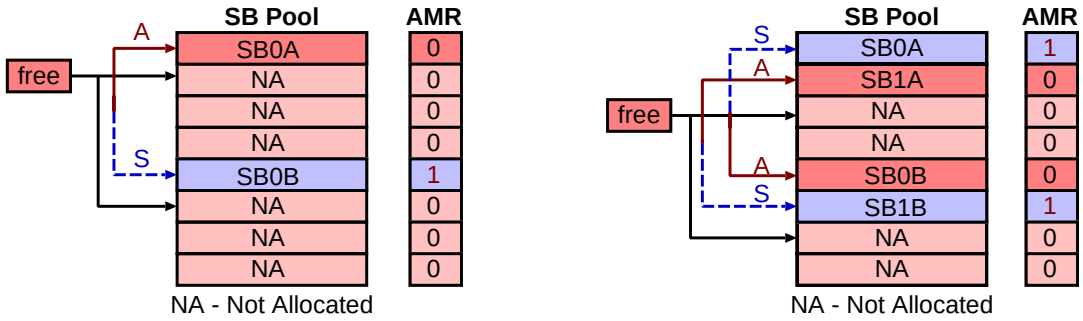


Figure 4.8: Stream Buffer (SB) allocation and the AMR. On a fault, a pair of pages (SB) is allocated. The fault page is active (A); it is being accessed by the accelerator; the prefetched page is speculative (S). The AMR helps to detect accesses to (S) pages.

the speculation, further prefetches are scheduled in the same fashion. Each speculative prefetch is designated to its corresponding stream buffer. While the accelerator accesses the active page (`A` in Figure 4.8), the VMW prefetches the speculative one (`S` in Figure 4.8). Ideally, for a correctly-guessed memory access stream and good timing of the prefetching, only one fault per stream should appear: the prefetching should prevent all other faults to appear.

Although for the moment we use a simple, software-managed and hardware-supported, stream-buffer-based technique [63, 90] that handles efficiently stream-based memory accesses, other techniques that cover nonsequential memory accesses (e.g., recursive [99], and correlation-based techniques [109]) could also be applied, with appropriate extensions in the system software (the VMW manager) and system hardware (the WMU). More importantly, the presented technique is completely transparent to the user. On the experimental results presented in Chapter 5, we show the effectiveness of the prefetching technique.

Chapter 5

Unified Memory Performance and Overheads

Mrki Vuče, podigni brkove,
Da ti vidju toke na prsima,
Da prebrojim zrna od pušakah
Kolika ti toke izlomiše!

O scowling Vuk, lift your moustache for me
and let me see the breastplates on your chest,
that I may count holes from rifle bullets,
see how many of them broke up your plates!

—Njegoš, *The Mountain Wreath*

THIS chapter introduces a performance metric for spatial computation which we use for performance and overhead analysis of existing approaches and our unified memory scheme. Later on, we use the definition of the performance pointers for experimental measurements. The measurements demonstrate the viability of our approach by showing (1) performance improvements the approach provides compared to the software-only applications and (2) limited overheads the approach introduces compared to the existing approaches. The dynamic optimisations in the system abstraction layer facilitate an additional performance improvement—it is achieved without any user intervention on the application code.

5.1 Performance Metric for Heterogeneous Computing

Performance analysis of codesigned applications requires having a common metric. The metrics used for standard processors [50, 56, 83] are just partially applicable to codesigned applications. Such applications have heterogeneous code consisting of user software and user hardware—the existing metrics are only applicable to software parts of codesigned applications. We assume for the moment that the application execution is sequential—user hardware and user software of a given application do not execute in parallel—and, later, we release this restriction in Chapter 7.

We do not address here the problem of how to partition the original application. We assume that the designer has already made the choice of critical code sections

to be mapped to hardware in a way to maximise the overall performance. That is why, if not otherwise stated, we primarily consider the speedup of the critical code section (refer to Section E.3 for an example). In general-purpose systems with a single CPU, we can use the following performance equation for the overall execution time (ET) of a software-only critical section [56]:

$$ET_{SW} = IC \times CPI \times T_{CPU}, \tag{5.1}$$

where IC stands for overall *instruction count* for the software code, CPI is an average *cycles per instruction* for the code execution, and T_{CPU} is the clock cycle time of the CPU (i.e., the CPU clock period). The three parameters of Equation 5.1 are subject to architectural, organisational, and implementation decisions of a CPU architect. They are interdependent: IC depends on ISA and compiler technology, CPI on CPU organisation and memory architecture, and T_{CPU} on hardware technology and CPU organisation.

Recalling the qualitative discussion from Section 2.2, an expression for the overall execution time of the hardware accelerator[1] is

$$ET_{HW} = \sum_{i=1}^{n} HT_i + \sum_{i=1}^{m} CT_i = HT + CT, \tag{5.2}$$

where CT is the total memory copy time, and HT is the total hardware execution time. In the general case, the number of hardware execution intervals (n) and copy intervals (m) are different. Equation 5.2 is convenient for the hardware accelerator where we can easily separate the time spent for memory transfers and the time spent in hardware execution (e.g., typical hardware accelerators with local memory from Figure 2.3 and Figure 2.4b). In Equation 5.2, we neglect the time spent on the software side to manage the memory transfers, calculate buffer addresses, and iterate until the computation is over. The experimental results show (as we present in Section 5.3 of this chapter) that the management time is negligible for the accelerators with simple memory access patterns.

The CPI parameter (from Equation 5.1) in the case of software gives an insight regarding the CPU architecture and organisation. There is no equivalent parameter in Equation 5.2. To build a performance equation for hardware accelerators with the form equivalent to Equation 5.1, we introduce a new metric for spatial computation. We suppose the hardware designer implements the accelerator by parallelising the original software code of the critical section, thus preserving the complexity of the critical section and without changing the size of the problem [31]. In this case, the total number of cycles for the accelerator execution depends on the problem size—the number of data to be processed. This is not immediately visible from Equation 5.2. Neither can the equation show the effectiveness of the performed parallelisation: what is the number of data processed per cycle, how does it relate to the execution support and memory hierarchy? Therefore, we introduce another metric for spatial and data-flow computation that we base on *Cycles Per Datum* (CPD) in place of *Cycles Per Instruction* (CPI) used for CPUs. The CPD

[1] We assume that the time for the execution transfers from software to hardware is negligible.

metric represents the average number of cycles spent by a hardware accelerator for processing the unit datum (a datum of the unit size; in our discussion, we consider it a 32-bit memory word). Using the CPD metric, the overall execution time of the hardware accelerator is

$$ET_{HW} = DC \times CPD \times T_{HW}, \tag{5.3}$$

where DC is the data count to be processed (i.e., the problem size) by the hardware accelerator, expressed in the number of memory words, and T_{HW} represents the hardware clock period—in the general case, T_{HW} is different from T_{CPU}. The DC parameter is not affected by the VLSI (or FPGA) implementation of the critical section, be it semelective or multilective [101, 102] (i.e., no matter whether the accelerator reads a particular input data from the local memory only once, which means it is semelective, or multiple times, which means it is multilective, what counts is the problem size); however, where the semelectivity and multilectivity of the accelerator implementation matter is the CPD parameter. We show examples of analytically calculating CPD from control-flow graphs and an example of its use in Appendix E.

Apart from convenient speedup estimation, the CPD metric gives an insight to the accelerator internal organisation: is the execution pipelined or not, what is the cost (and influence on performance) of memory accesses, what is the number of memory ports? We can draw parallels between CPI and CPD: (1) a computer architect strives to get both close to one; (2) having CPI or CPD smaller than one indicates the presence of multiple issue or multiple memory ports, respectively; (3) both indicate the presence and efficiency of pipelining.

Similarly to an approach for general-purpose systems used for studying the effects of memory hierarchies [56], we write Equation 5.3 in a form that clearly separates the pure hardware execution from the execution support (Section E.2 gives more details on the separation):

$$ET_{HW} \;=\; DC \times (CPD_{HW} + CPD_{memory}) \times T_{HW}, \tag{5.4}$$

where CPD_{HW} is the CPD value for the hardware accelerator assuming the ideal memory hierarchy (i.e., memory accesses introduce no stalls) and CPD_{memory} represents the overhead incurred by the memory transfers. In the following section, we apply the same equation for calculating the overhead of virtual-memory-enabled hardware accelerators.

5.2 Overhead Analysis

In Equation 5.2, we have expressed the overall execution time (ET_{HW}) of a hardware accelerator. The time components HT and CT stand for pure hardware-execution time and copy time. In Equation 5.4, we have developed another form of the performance equation; its factors CPD_{HW} and CPD_{memory} designate cycles per datum for pure hardware-execution (assuming ideal memory) and for the memory access

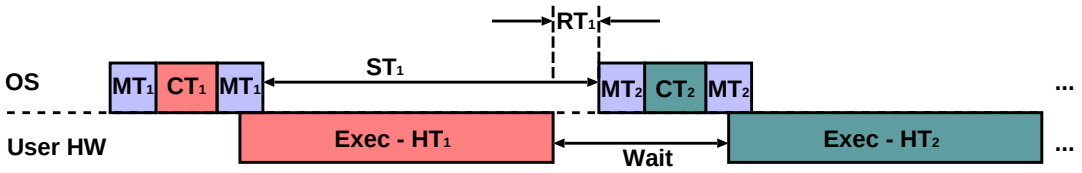


Figure 5.1: The operating system (OS) activities related to the hardware accelerator execution (User HW). The VMW manager of the OS sleeps (ST_i time) during the hardware execution (HT_i time). The hardware stalls on a memory fault and waits for the OS service. After some response time (RT_i), the VMW manager reacts on the interrupt event. It manages the translation data structures (MT_i time) and copies the required page to the local memory (CT_i time). After some additional management, it resumes the hardware execution.

overhead (the price of managing—either in software or hardware—the memory hierarchy).

Based on Equations 5.2 and 5.4, we can express the execution time of virtual-memory-enabled hardware accelerators. Compared to the typical existing approaches, there are several sources of overhead in our scheme. The first one is related to the virtual address translation (limitations of the FPGA technology and design methodologies different than in the case of ASICs result in a TLB which performs the translation in multiple clock cycles, i.e., 4–6 in our implementations). Thus, the *Hardware execution Time* (HT) of virtual-memory-enabled accelerators is longer than in the case of accessing the memory by physical addressing:

$$HT_{virtual} = t \times HT_{typical}, \tag{5.5}$$

where $t > 1$ is the translation overhead. It would be practically one, if the WMUs were implemented in ASIC, as a standard part of reconfigurable SoCs.

Figure 5.1 shows different time components during the execution of a virtual-memory-enabled hardware accelerator. The VMW manager in the OS manages translation data structures (Management Time—MT_i) and copies pages from/to the user memory (Copy Time—CT_i). When finished copying, it updates the data structures MT_i), invokes the kernel scheduler, and sleeps (Sleep Time—ST_i). The hardware accelerator executes (Hardware Time—HT_i) until it finishes or tries accessing a page not present in the local memory. In both cases, it waits for the OS action. The OS responds after some time (Response Time—RT_i).

Having the OS responsible for the translation management brings MT and RT (the sums of corresponding MT_i and RT_i, respectively) as inherent overheads to the performance equation. *Sleep Time* ($ST = HT_{virtual} + RT$) represents the time of the OS idleness, with respect to its actions on behalf of the accelerator. If we assume the same data transfer technique employed for a virtual-memory-enabled accelerator and the corresponding typical one, the overall *execution time* (ET) of the virtual-memory-enabled accelerator is

$$ET_{virtual} = \underbrace{t \times HT_{typical} + RT}_{Sleep\ time\ (ST)} + c \times CT_{typical} + MT, \tag{5.6}$$

where $c > 1$ is the page-level granularity overhead (discussed later in this section). Another way to express the execution time, using the CPD metric is

$$\begin{aligned} ET_{virtual} &= DC \times (CPD_{virtual} + CPD_{vmemory}) \times T_{HW} \\ &= \underbrace{DC \times CPD_{virtual} \times T_{HW}}_{t \times HT_{typical}} \\ &\quad + \underbrace{DC \times CPD_{vmemory} \times T_{HW}}_{c \times CT_{typical} + MT + RT}, \end{aligned} \tag{5.7}$$

from where we can write:

$$CPD_{virtual} = t \times CPD_{typical}, \tag{5.8}$$

and

$$CPD_{vmemory} = \left(c + \frac{MT + RT}{CT_{typical}} \right) \times CPD_{memory}, \tag{5.9}$$

to express explicitly the overheads of our approach.

Thinking in the spirit of Equation E.4 from Section E.2, one can notice that the proposed approach increases the miss rate (by having pages typically smaller than data partition sizes specified explicitly by the programmer) and the miss penalty (by incurring the response time, the management time, and the page-level granularity).

Figure 5.2 explains the overhead of the page-level granularity. In contrast to typical approaches, not necessarily all of the transferred data are used by the accelerator (in the extreme case, as Section 4.3 illustrates for write faults, it may happen that the accelerator reads none of the transferred data). If we assume that an image is layered in the memory row by row and the size of the processing window is smaller than size of the image, going from the row i to the row $i + 1$ of the window means skipping a certain number of outer pixels. However, having no information of the exact size of the data to be processed, the VMW has to transfer these pixels along with the rest of the page to the local memory. The effect is well-known and studied in the OS and cache related research [60, 93]. To tackle different application requirements, our WMU implementation and the VMW manager support different number of pages and sizes of the local memory (refer to Appendix A and Appendix B).

Despite the incurred overhead, the involvement of system software in execution steering simplifies hardware and software interfacing and releases software programmers and hardware designers from the burdens of the typical approaches. As Section 4.4 shows, the runtime steering in the system software allows dynamic optimisations that can improve the performance, in a manner completely transparent for end users, without any change in application source code.

Figure 5.3 shows details of the OS prefetching activities during the execution of a virtual-memory-enabled accelerator. If predictions are correct, the accelerator execution is uninterrupted. In comparison to Figure 5.1, *Sleep time* (ST) is shorter because of the speculative activities of the VMW manager. Furthermore, *Copy time*

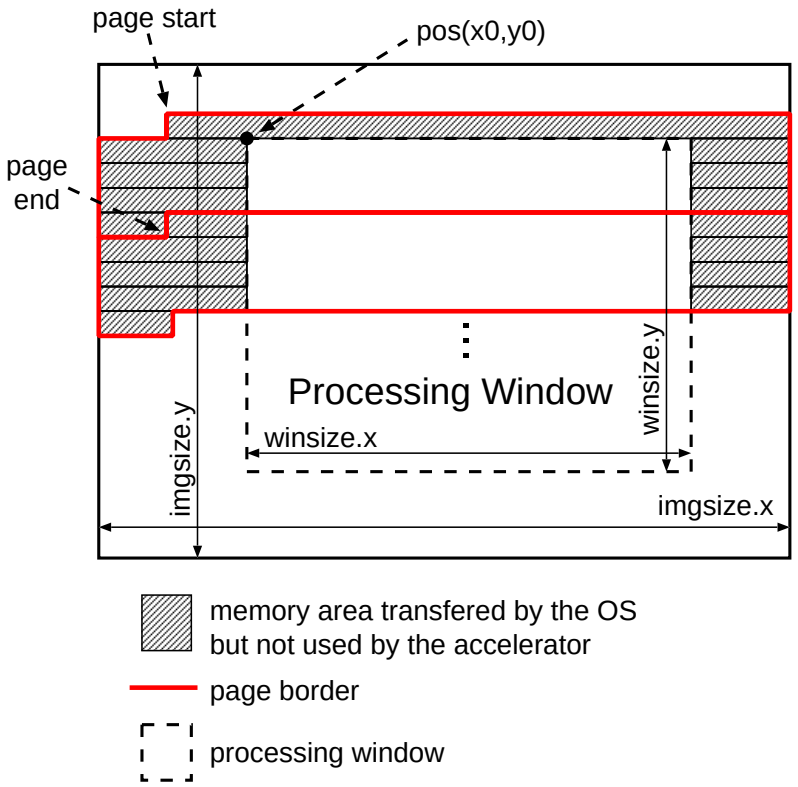


Figure 5.2: Memory layout of an image. Processing a window region smaller than the image size imposes the copying overhead because of the page granularity.

(CT) is overlapped with the hardware execution time (HT): the prefetching can hide memory-to-memory copy latency.

Although Equation 5.6 suggests that shorter ST implies faster overall execution, masking $HT_{typical}$ out of the equation is not intuitively comprehensible. On the contrary, Equation 5.7 featuring the CPD metric sheds more light to the positive effects of prefetching: we use the same hardware accelerator (with the same $CPD_{virtual}$) as in the case without prefetching, but the runtime optimisation makes the memory system more efficient (prefetching can improve $CPD_{vmemory}$), lowering the influence of the inherent overheads (such as page-level granularity). We show the benefits of our approach and apply the developed metric through experiments in Section 5.3.

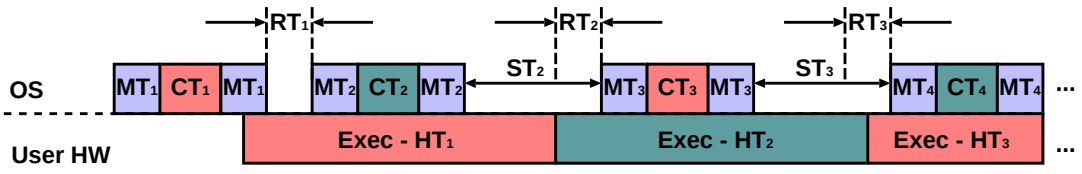


Figure 5.3: The operating system (OS) activities related to the hardware accelerator execution (User HW). Instead of sleeping, the VMW can fetch in advance pages that may be used by the accelerator.

5.3 Experimental Results

We have implemented two different VMW systems—based on Altera (Excalibur series, EPXA1 device [3], with a 133MHz ARM processor) and Xilinx (Virtex-II Pro series, XC2VP30 device [130], with a 300MHz PowerPC processor) reconfigurable devices—with several applications running on them. In this section we present the results for the IDEA cryptography application [82] and the ADPCM voice decoder application [68]. Both applications exercise sequential memory accesses (with one input and one output data stream) within their critical code sections. Chapter 6 and Chapter 7 give additional examples, also showing measurement results for applications with different access patterns, such as processing a window within an image (recall Figure 5.2).

The IDEA algorithm consists of eight rounds of the core transformation followed by the output transformation. When designing the coprocessor, the eight rounds can be "unrolled" a certain number of times, depending on the available reconfigurable logic resources. The computation of a round contains four multiplications, four additions, and several XOR operations. The output transformation consists of two multiplications and two additions. The algorithm exhibits parallelism exploitable by hardware implementation. The ADPCM decoder is simpler and exhibits less parallelism. It produces four times more data than it consumes—one input page produces four output pages.

We have developed the VMW manager as a Linux kernel module and ported it to both platforms. The WMU is designed in VHDL to be synthesised onto FPGA together with a hardware accelerator. Because of the limitations of the FPGA technology, the virtual-to-physical address translation is performed in multiple cycles (4–6 cycles depending on the implementation). We give more details on the reconfigurable platforms we used in Appendix C.

5.3.1 Typical Data Sizes

Figure 5.4 and 5.5 show execution times of the IDEA and ADPCM applications (running on the Altera-based board) for typical input data sizes. The VMW-based versions (with and without prefetching) of the applications achieve significant speedup (especially with prefetching) compared to the software cases, while the overhead is limited compared to the typical accelerators (in the category of accelerators accessing the local memory, as discussed in Section 2.2).

We measure three components (discussed in Section 5.1) for the typical accelerator: (1) $CT_{typical}$, time to copy the exact amount of data for processing; (2) $HT_{typical}$, time to perform the computation in hardware; and (3) $MT_{typical}$, time to update the pointers and iterate until completion—the experiments show that $MT_{typical}$ is negligible for typical hardware accelerators. Also, we measure three components (discussed in Section 5.2) for the virtual-memory-enabled accelerators: (1) CT; (2) ST; and (3) MT. ST consists of the hardware time for executing a virtual-memory-enabled accelerator plus the OS response time (RT). The OS response time is not present when programming the typical accelerator like in Figure 2.4b; the software part of the application is busy waiting for the accelerator to finish; the program

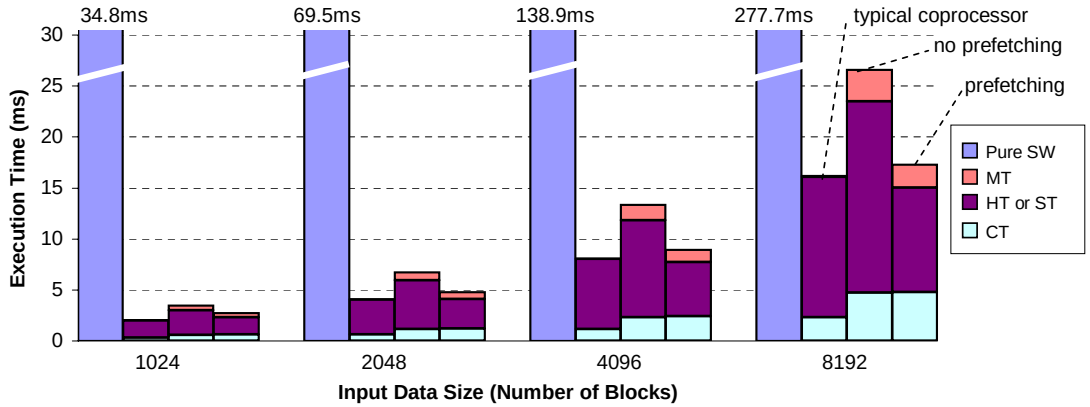


Figure 5.4: IDEA execution times for different number of input blocks. The figure compares the performance of a pure software with hardware-accelerated versions of the IDEA application.

responds faster but unnecessarily wastes the CPU time.

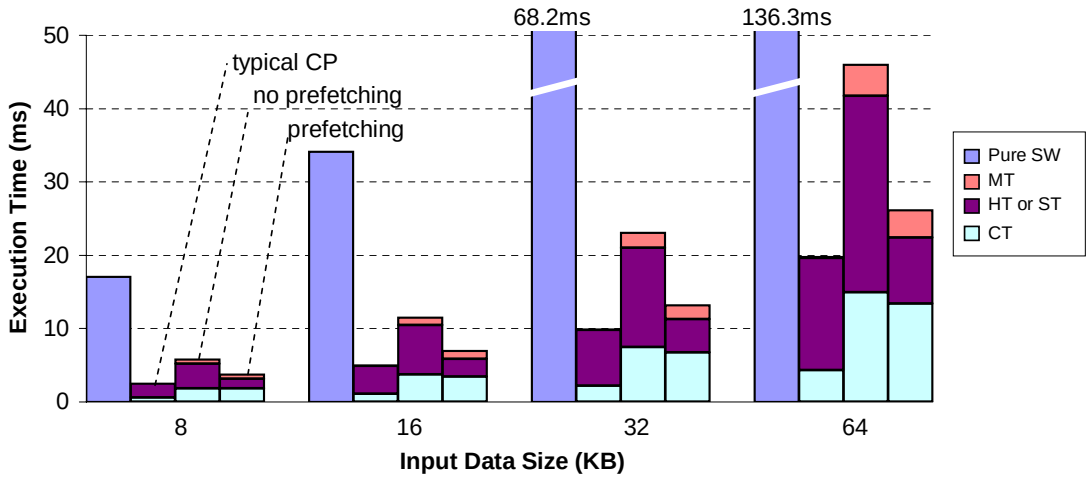


Figure 5.5: ADPCM decoder execution times for different input data sizes. The figure compares the performance of a pure software with hardware-accelerated versions of the ADPCM decoder.

The cost of managing higher abstraction (represented by MT) goes up to 15% of the total execution time, which is acceptable. Current inefficient implementation of the WMU in FPGA (represented by the difference between $HT_{typical}$ and $HT_{virtual}$) results in about 20% longer execution in the VMW case ($t \approx 1.2$). Fortunately, this overhead can be reduced by implementing the WMU as a standard VLSI part on a SoC (exactly as it is the case with the MMUs) or by pipelining the WMU translation. The copy overhead c expressed in Equation 5.6 causes the difference between copy times of the typical and virtual-memory-enabled accelerator.

A significant amount of time is spent in copying to/from the local memory, to which compulsory misses contribute a considerable part and would be unavoidable even if no virtualisation was applied. In the cases with no prefetching, for both IDEA and ADPCM virtual-memory-enabled accelerators, when the data set size

grows, capacity misses appear (e.g., from 1024 blocks onwards in the case of IDEA). Additional time is spent in the OS for management, but the speedup is only moderately affected. Besides capacity misses, the ADPCM decoder also exhibits conflict misses: the simple policy used by the VMW manager for page replacement cannot handle its specific memory access pattern (prefetching eliminates these conflict misses—observing Figure 5.5 shows that the copy times are lower with prefetching).

Figures 5.4 and 5.5 also show that a dynamic optimisation like prefetching shortens execution times and increases the obtained speedup. Prefetching reduces the number of page faults by allowing processor and accelerator execution to overlap. Although running at the same speed, in the prefetching case, the ADPCM decode accelerator finishes its task almost twice as fast compared to the nonprefetching case. As Figure 5.3 indicates, the sleep time decreases because of the VMW manager that now handles access requests in parallel with the accelerator execution. Counterintuitively, the management time slightly decreases: the number of fault-originated interrupts is dramatically lower (e.g., in Figure 5.11, in the case of 32KB input data size it goes down from 48 to only 2). Meanwhile, multiple access-originated interrupts (generated as Figure 4.6 explains) may appear within a relatively short time interval (e.g., two streams usually cross the page boundary at about the same time) and the VMW manager services them at the same cost. This is not the case for the faults. For a fault to appear, the previous fault needs to be already serviced.

Another remark related to the ADPCM decoder is the following: since the VMW manager in the nonprefetching case uses a simple FIFO policy for page eviction, it may happen that a page still being used gets evicted; this will cause an additional fault and, consequently, the manager will need to transfer the page from the main memory once again, before the accelerator continues the execution. On the other hand, the prefetching approach with stream-buffer allocation is less sensitive to the applied page eviction policy because the VMW manager allocates distinct stream-buffers for input and output streams.

We stress once more that the experiments are performed by simply changing the input data size, without any need to modify either the application code, or the accelerator design. In particular, no modifications are needed even for datasets which cannot be stored at once in the physically available local memory: the VMW manager takes care of partitioning and copying transparently in such case.

CPD Discussion. From the measurements and from Equation 5.7, we can determine $CPD_{virtual}$ and $CPD_{vmemory}$. For example, for the IDEA hardware accelerator we find $CPD_{virtual} = 24.4$ and $CPD_{vmemory} = 14.5$. The overall CPD is $CPD_{virtual} + CPD_{vmemory} = 38.9$, from where we remark a significant impact of the $CPD_{vmemory}$ parameter to the performance (the equivalent of almost 15 cycles per datum is spent in memory related operations). On the other hand, the $CPD_{virtual}$ parameter obtained from the measurements closely corresponds to analytically computed value (the calculation in Appendix E gives $CPD_{virtual} = 24.7$). The prefetching in the system layer supporting unified memory significantly improves the memory related overhead giving $CPD_{vmemory} = 0.9$ and $CPD = 25.3$.

The virtual-memory-enabled accelerator suffers from slow translation of virtual memory addresses, because of the current WMU implementation in FPGA (it needs

4–6 cycles for translating a memory address present in the local memory). We can use the CPD metric to estimate the accelerator performance with the ideal WMU (one cycle per successful translation). In this case (as Section E.1 shows), the CPD number for the accelerator is $CPD_{virtual} = 18.6$. The memory access pattern of the accelerator does not change, since we do not alter the memory interface. Thus, we have the same CPD number for the memory overhead $CPD_{vmemory} = 14.5$. If we assume $DC = 16384$ (meaning 8192 input IDEA blocks read through a 32-bit memory port) and $T_{HW} = 42ns$ (for the accelerator running at 24MHz), we can write the performance equation for the IDEA accelerator with the ideal WMU and without prefetching:

$$\begin{aligned} ET_{improved} &= DC \times (CPD_{virtual} + CPD_{vmemory}) \times T_{HW} \\ &= 16384 \times (18.7 + 14.5) \times 42ns \\ &\approx 22.7ms, \end{aligned} \tag{5.10}$$

or, for the accelerator with prefetching:

$$\begin{aligned} ET_{pref.improved} &= DC \times (CPD_{virtual} + CPD_{vmemory}) \times T_{HW} \\ &= 16384 \times (18.7 + 0.9) \times 42ns \\ &\approx 13.4ms, \end{aligned} \tag{5.11}$$

which would be even faster than the appropriate typical accelerator. Of course, a software programmer and a hardware designer of the typical accelerator could arrange prefetching by hand modifications of the code—thus introducing additional design complexity—to achieve superior performance. The point here is that the virtual-memory-enabled hardware accelerator can achieve the performance improvement without any effort on the side of the programmer and hardware designer.

Similarly to the previous estimation, we can use the CPD metric for estimating execution times of an improved version of the IDEA accelerator (the available space in the Altera device limits the performance of the current implementation; we can improve the accelerator by assuming its implementation on a bigger device, thus exploiting more aggressively the available parallelism) with $CPD_{virtual} = 8$ (one of the examples from Section E.1 shows the CPD calculation for the IDEA accelerator with completely unrolled computation). Since the memory access pattern of the improved accelerator does not change, we can again use Equation 5.7 (having values for $CPD_{virtual} = 8$ and $CPD_{vmemory} = 14.5$) to obtain the execution time $ET_{improved} \approx 15.5ms$ of the improved accelerator without prefetching. Also, we can estimate (with $CPD_{virtual} = 8$ and $CPD_{vmemory} = 0.9$) the execution time $ET_{pref.improved} \approx 6.1ms$ of the improved accelerator with prefetching. The introduced CPD metric proved a convenient mean for estimation and comparison between different implementations of functionally-equivalent hardware accelerators.

5.3.2 Small Input Data

Although not likely in practice, using the accelerated applications with small input data sizes gives us better insight about the overhead of the page-level memory granularity. While the typical solution always copies the exact amount of data to process, the virtualisation layer always copies entire local memory pages. Figure 5.6 compares speedups (for the Altera-based board) of the typical and VMW-based

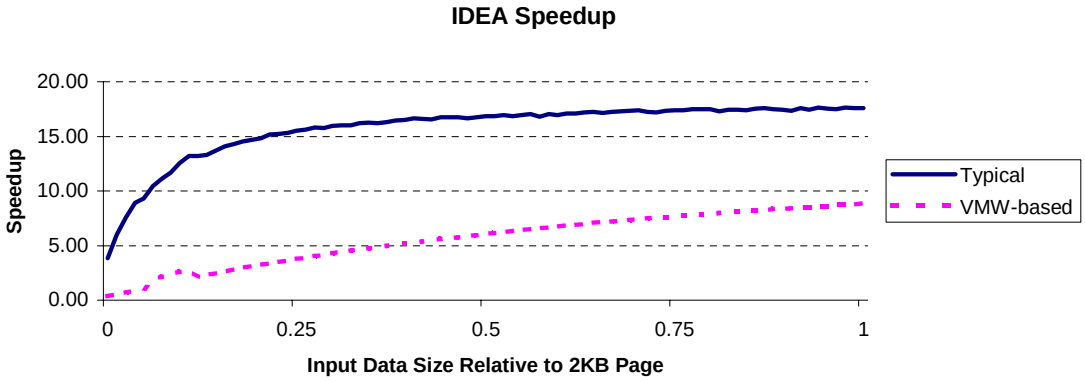


Figure 5.6: Speedup (relative to pure SW) for small input data sizes.

IDEA accelerators, in the case of small input data sizes. While the typical accelerator achieves speedups quite early, the VMW-based solution lags behind. The typical solution has no overhead of copying entire pages into the local memory.

Figure 5.7 and 5.8 explain the sources of the overhead. The execution times of the typical IDEA accelerator with no virtualisation present for input data sizes relative to a 2KB page can be seen in Figure 5.7. Again, the three time-components—their

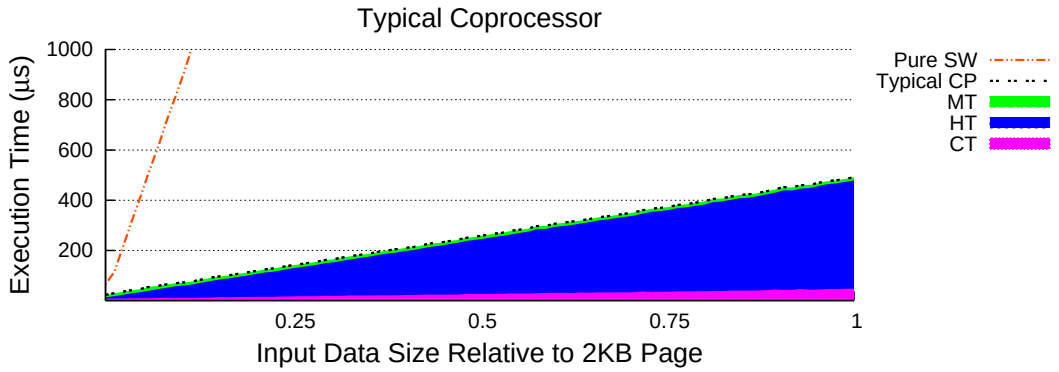


Figure 5.7: Typical accelerator execution times compared to pure software for small input data sizes. The programmer transfers exactly the required amount of data to/from the local memory—*Copy Time (CT)* grows linearly, but rather slowly, with the input data size.

corresponding lines in Figure 5.7 graphically stacked starting from CT—contribute to the overall execution time of the typical hardware accelerator. Copy time and hardware time grow linearly with the input data size, while the management time is negligible.

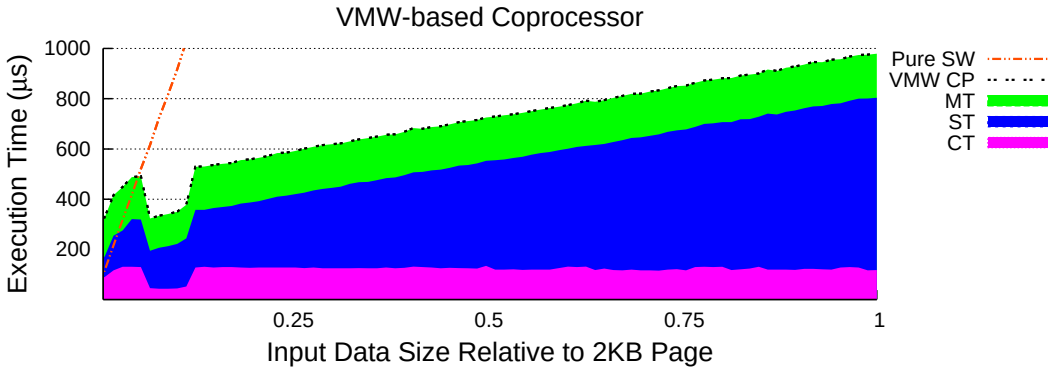


Figure 5.8: VMW-based accelerator execution times compared to pure software for small input data sizes. The VMW manager always transfers pages—*Copy Time (CT)* is constant except for a short-interval drop caused by landing of the input and output data onto the same page.

Figure 5.8 shows the execution times of the VMW-based IDEA accelerator for input data sizes relative to a 2KB page. The three time components—also graphically stacked—contribute to the overall execution time (as Figure 5.1 shows). The copy overhead degrades performance of the VMW-based solution: the size of data to process is much smaller than the size of local memory pages (2KB per page in this experiment). In the particular case, only for input data sizes of 0.05 or larger, the VMW-based accelerator becomes faster than the software implementation. Copy time is constant (except for a short-interval drop), and management time is larger than in the typical accelerator case. The drop of copy time appears because of the fact that the two objects—memory for the input and output data that is allocated by `malloc` on the heap—do not necessarily occupy consecutive memory regions. If they happen to be in the same page (as it is the case for input data sizes between 0.06 and 0.13) the copy time is shorter.

5.3.3 Different Number of Memory Pages

Having multiple WMU operation modes allows the VMW to fit accelerators with different memory access patterns. Except for some extreme values, changing the WMU operating modes does not influence performance dramatically (as Figure 5.9 and Figure 5.10 show for the ADPCM application and the IDEA application, respectively, running on the Altera-based board).

As expected, MT increases with the number of pages (larger data structures are managed), while CT is almost constant except for the thrashing effect (the pages are just swapped in and out) and for the ADPCM decoder (its memory behaviour and the simple allocation policy trigger conflict misses which require some additional transfers).

Increasing the number of pages (i.e., the fixed-size local memory is divided into smaller pages) for the same input data size increases the number of faults (Faults NP in Figure 5.11). However, prefetching in the VMW keeps the number of faults (Faults P) low and constant (except for having only two VMW pages, when memory

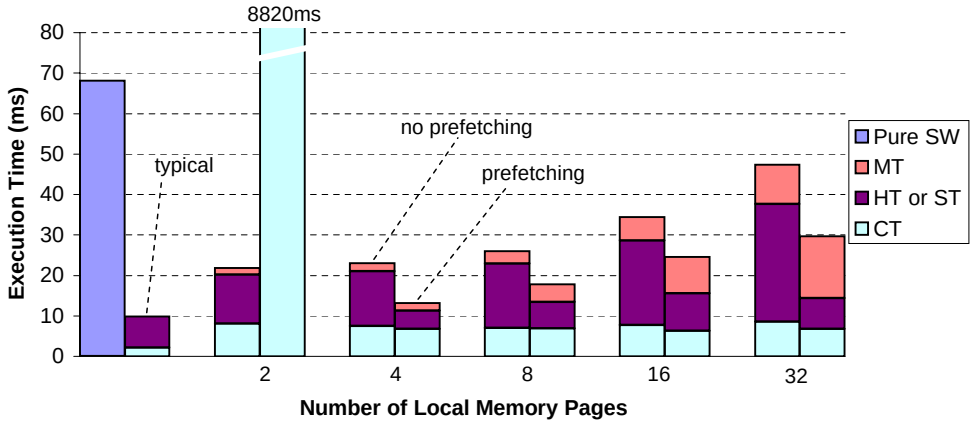


Figure 5.9: ADPCM decoder performance for different number of VMW pages (the input data size is 32KB).

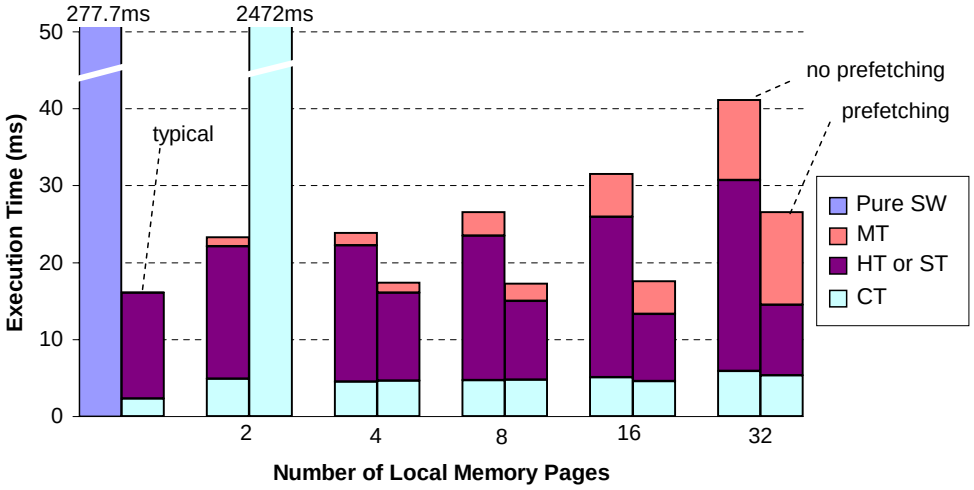


Figure 5.10: IDEA execution times for different number of local memory pages (the input size is 8192 IDEA blocks).

thrashing appears—also visible in Figures 5.9 and 5.10). For smaller page sizes (i.e., local memory contains more pages), manage and copy time intervals become comparable to the hardware execution intervals: late faults—less costly than regular ones—appear (a fault is "late" when the WMU reports a fault while the missing page is already being prefetched by the VMW).

5.3.4 Area Overhead

Table 5.1 shows the complexity of the WMU in terms of occupied FPGA resources (logic cells and memory blocks), for the Altera Excalibur device (EPXA1). The overhead does not include cost of the local memory (recall Figure 4.2 in Section 4.2) as this memory is necessary even without virtualisation with the WMU. Although the Altera device that we use is the smallest in its family, the WMU area overhead is acceptable (not more than one fifth of the EPXA1 resources are used). The area overhead would be practically null, if the WMU were implemented in ASIC, as it is

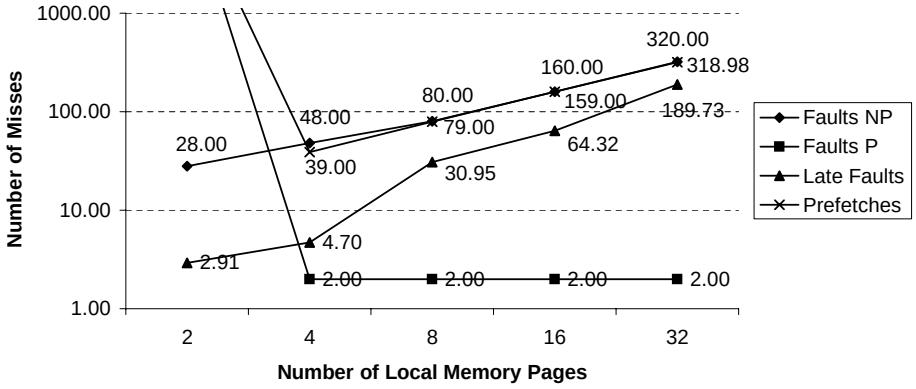


Figure 5.11: ADPCM decoder faults with (P) and without (NP) prefetching.

Block Type	Number of Units	Device Occupancy	WMU Fraction IDEA	WMU Fraction ADPCM
LC	576	14%	16%	48%
MEM	5	19%	45%	83%

Table 5.1: WMU area overhead. Usage of FPGA resources (i.e., logic cells—LC, and memory blocks—MEM) are shown for the Altera EPXA1 device (in number of units and percentage of the device occupancy). The IDEA and ADPCM columns show what fraction of the overall accelerator designs (virtual-memory-enabled) is occupied by the WMU.

the case with MMUs. Furthermore, having the WMU implemented in ASIC would decrease the execution time.

5.3.5 Results Summary

Figure 5.12 shows execution times for the IDEA application running on two different platforms (the Altera-based with Excalibur EPXA1 device, and the Xilinx-based with Virtex-II Pro XC2VP30 device). Our intention is not to compare the platforms, but to emphasize that running the experiments on a different platform implies only porting the WMU hardware and the VMW software, and *does not require any changes to the accelerator HDL nor to the application C code*. VMW-based applications can achieve significant performance advantage over the pure software, in spite of the introduced virtualisation. The overhead can be reduced, especially with the address translation done in VLSI (as it is done for MMUs). Dynamic optimisations can offer additional speedups with no change on the application side.

In the case of the Xilinx platform, both VMW-based accelerators (without and with prefetching) of the IDEA encryption algorithm run faster than the typical accelerator accessing local memory. In spite of highly-optimised C-library functions—and their kernel equivalents—for memory copy operations (on which the typical solution relies), using PowerPC for data transfers over the system bus (refer to Appendix C for platform details) showed to be inefficient. For this reason, the VMW manager employs—transparently to the end user—a controller for *Direct Memory Accesses*

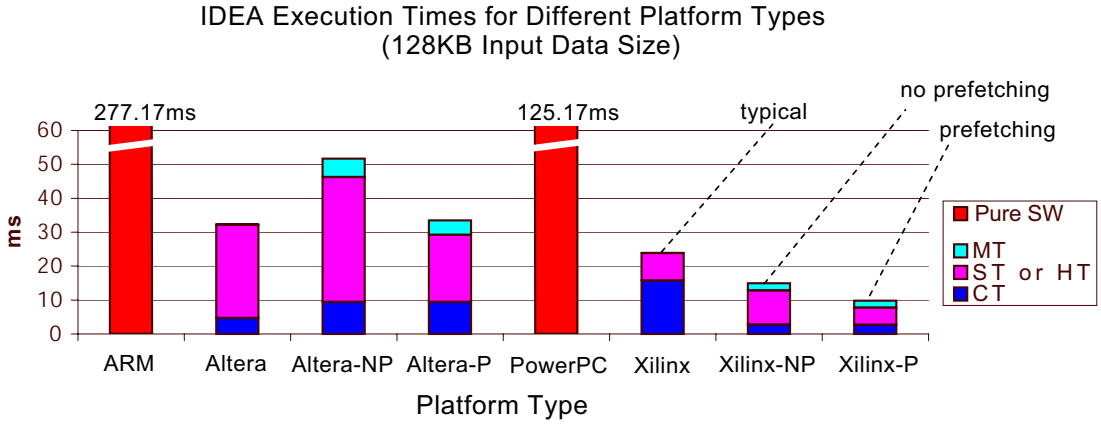


Figure 5.12: Two different platforms (Altera Excalibur-based and Xilinx Virtex-II Pro-based) run the same application. Results are shown for pure software (ARM, PowerPC), typical hardware accelerators accessing local memory (Altera, Xilinx), and VMW-based accelerators, with no prefetching (Altera-NP, Xilinx-NP) and with prefetching (Altera-P, Xilinx-P) in the VMW.

(DMA) (as Figure 4.2 in Section 4.2 shows). On the other side, the typical solution has no such support, thus, it directly suffers from the performance drawback. Of course, the programmer of the typical solution could also use the DMA controller to improve the performance but, this would require descending to the level of system programming (e.g., to add the appropriate support in the kernel) or using some of the existing solutions for user-level DMA (e.g., [80]. In the case of VMW-based applications, on the contrary, the application-level programmer can always stick to the simpler programming interface defined by the VMW manager and, at the same time, benefit from the system-level support for improved performance.

In the following chapters, we present the VMW and WMU extensions (1) supporting transparent execution transfers and hardware callbacks to software (in Chapter 6) and (2) enabling multithreaded execution of user software and user hardware within the unified OS process (in Chapter 7). In both chapters, we perform and show results of additional experiments with cryptography and image processing applications, thus demonstrating unrestricted synthesis of hardware and multithreaded execution of codesigned applications.

Chapter 6

System Support for Execution Transfers

If you wish to make an apple pie from scratch, you must first create the universe.
—Carl Sagan, *Cosmos*

THE common virtual address space for user software and user hardware simplifies the interfacing and supports advanced programming paradigms for codesigned applications. Another essential part of the unified OS process is the support for execution transfers from user software to user hardware and vice-versa. In this chapter, we delegate the execution transfer task to the system. Having system software and system hardware responsible for the execution transfers furthermore simplifies the software and hardware interfacing. All together with the unified memory for user software and user hardware, it fosters unconstrained partitioning and software-to-hardware code migration. We demonstrate how the unified OS process enables unrestricted automated synthesis of high-level programming languages to hardware.

6.1 Callbacks to Software

We have discussed in Section 2.3, how programmers may use software wrappers to call back software and system services on behalf of user hardware. Our unified memory abstraction (introduced in Section 2.5 and implemented in Chapter 4) allows user hardware to use return results of system calls and standard-library functions. This would be either burdensome or even impossible without common virtual-memory address space.

Figure 6.1 and Figure 6.2 illustrate how a hardware accelerator can call back software, even for sophisticated functions such as heap memory allocation. When the accelerator calls back the software, the wrapper invokes the appropriate function and returns the virtual memory pointer back to the accelerator. Execution of the hardware accelerator is then resumed. When resumed, the accelerator can use the pointer to the allocated memory without any obstacle: using the pointer to access the memory initiates the transfer of the accessed data to the local memory of the

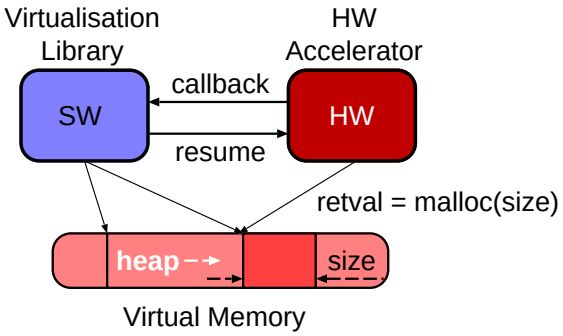


Figure 6.1: Memory allocation callback. Since hardware accelerators have unified memory image with software, using the *malloc* result is straightforward.

accelerator; were there no unified memory abstraction, such callback would not be possible.

```
/* excerpt of a virtualisation library function */
...
struct cp_param param; /* parameter exchange structure */
...
sys_hwacc(HW_ACC, &params); /* accelerator start */
...
while (!params.hwret) { /* wait for callback */
  switch(params.cback) { /* choose function to call */
    case 1:  ...  break;
    case 2:  ...  params.retval = malloc(params.p[0]);
      sys_hwacc(HW_ACC_RESUME,&params); /* accelerator resume */
      ...  break;
        case n:
      }
    }
```

Figure 6.2: Supporting *malloc* callback for hardware accelerators. The library function receives the callback to hardware and based on the callback identifier calls the appropriate software function. Once the function returns, it passes the result to the accelerator and resumes its execution.

Although, in this way, hardware accelerators can practically call any software function (memory side effects are handled through the VMW manager enforcing the memory consistency), if there is no automation present in the design flow, it is still on the programmer to write wrappers and handle the execution transfer to the called function (including parameter passing). We move these activities to system software and release the programmer from writing callback wrappers. We explain the system support within the VMW manager for transparent transfer executions in the following section.

6.2 Kernel Mediation for Callbacks

The invocation of hardware accelerators goes through the *sys_hwacc* system call provided by the VMW manager. We extend the VMW manager to support the

execution transfer in both directions (i.e., we also support user hardware calling back user software). The execution transfer goes through the OS kernel. We implement the *Application Binary Interface* (ABI) for the particular host CPU within the VMW manager. An ABI [9, 125, 134] defines the calling convention (i.e., calling sequence—how function arguments are passed to callees and how return values are retrieved by callers) at the binary machine-instruction level for a CPU family. The ABI also describes execution transfers from applications to the OS or to libraries. It is different from an *Application Programming Interface* (API). While an API guarantees the source code portability by defining the interface between source code and the OS and libraries, an ABI guaranties integrating the object code (compiled source) without changes on systems using a compatible ABI. A high-level language compiler (more precisely its back-end) implements the ABI for a particular platform.

Calling the *sys_hwacc* system call from a C program ensures, after compilation, compliance to the target ABI. To achieve seamless integration of software and hardware, we also delegate such responsibility to hardware; calling back software from a hardware accelerator requires hardware designers (or a synthesiser—a compiler equivalent on the hardware side) to follow the given ABI. The same would be the case if we had a programmer writing parts of the application code in assembler; integrating assembly routines with a high-level language program demands following the ABI rules. Writing the hardware accelerator code in a hardware description language is equivalent to writing in an assembler. If a synthesiser is used, the designer delegates the ABI interfacing tasks to the automated generation of ABI interfaces. Having the unified memory between software and hardware allows the designer or the synthesiser to use the user-space stack for calling sequence and parameter passing. As it is the practice with CPUs [56] to avoid using the stack (for invoking functions with small number of parameters) by using general purpose registers or registers windows, we design the WMU with an exchange register file (as Figure 4.2 shows). We give more details on using the register file for parameter passing in Appendices A and B.

Apart from hardware accelerators complying to the ABI interface when calling back software, we need the VMW manager involvement in the execution transfer: a hardware accelerator cannot just jump in the user software or system call code directly; the VMW manager performs this task instead. We use a mechanism similar to UNIX signal handling [76, 117] to implement the task of execution transfer from hardware to software. Figure 6.3 shows the sequence of the events to call back software from hardware.

At the system-call entry, the CPU switches the execution mode from the user mode to a privileged one. This also means changing the memory address space—user-space memory is no more directly accessible by the kernel, but only through few special memory-access functions [12, 19]. Changing the memory address space imposes using a different stack when running in the kernel mode; each process has a user stack and a kernel stack for user-space and kernel-space executions, respectively. The first frame allocated on the kernel stack contains the state of all CPU registers visible to the programmer. In this way, before returning from the system call, the kernel can recover the original CPU state.

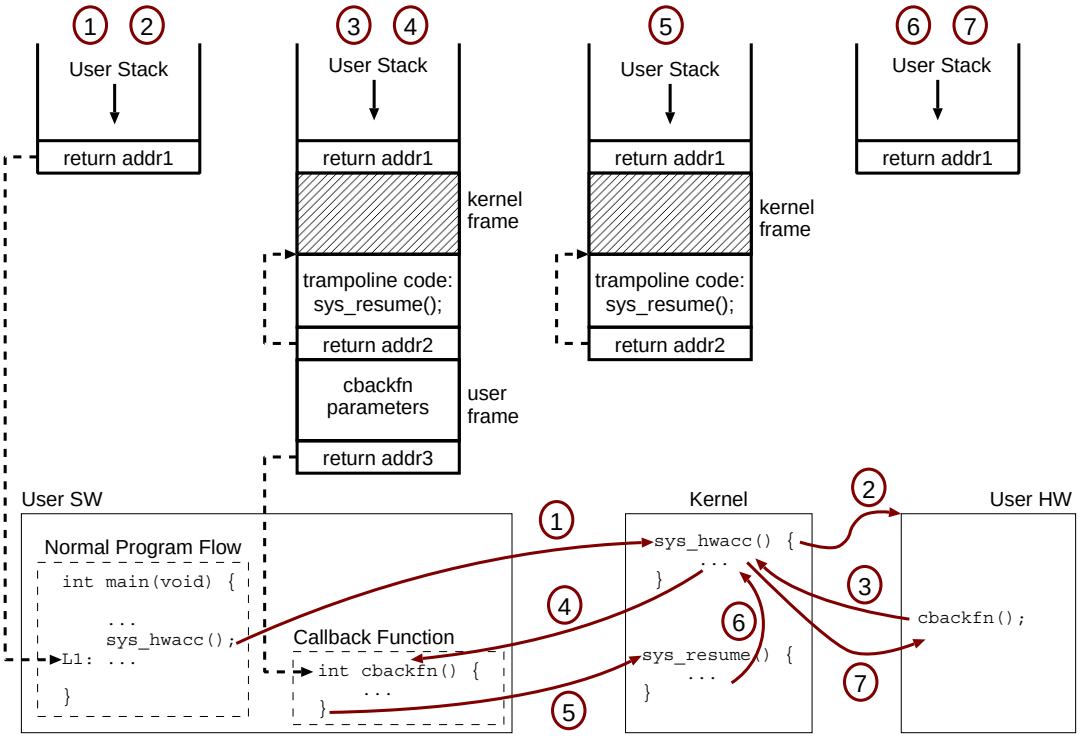


Figure 6.3: Kernel mediation for hardware accelerators calling back software. The kernel uses a mechanism similar to UNIX signal handling to fake the regular control-flow of the program and jump onto a callback function. For this purpose, the kernel manipulates the user space stack. It puts the recovery code (*trampoline*) and the callback address (*return addr3*) on the user stack to provoke invoking the callback function transparently to the end user.

After invoking the *sys_hwacc* system call (action ① in Figure 6.3), the top of the user stack contains the return address pointing to the normal program flow (the *main* function in Figure 6.3). The VMW manager of the kernel transfers the execution to the user hardware (action ② in Figure 6.3). User hardware starts the execution and performs the requested computation. When a user hardware callback to software arrives (action ③ in Figure 6.3), the user software is blocked on the *sys_hwacc* system call. The VMW manager prepares the execution transfer back to software: (a) it copies to the user stack a kernel frame containing the user state of the CPU at the *sys_hwacc* system call entry; (b) it puts trampoline code [104] on the user stack and, on the top, the return address pointing to it; (c) it allocates a user frame with parameters (received from the user hardware) to the callback function (*cbackfn()*) and puts on the top of the stack the pointer—return address—to the function code. After the preparations are over, the VMW manager returns from the system call, changing the execution mode from kernel to user (action ④ in Figure 6.3); the CPU jumps to the return address found on the top of the user stack—the callback function (out of the normal program flow) is launched. When completing its execution, the callback function returns to the caller; it lands on the return address at the top of the stack pointing to the trampoline code. The trampoline code provides a mean

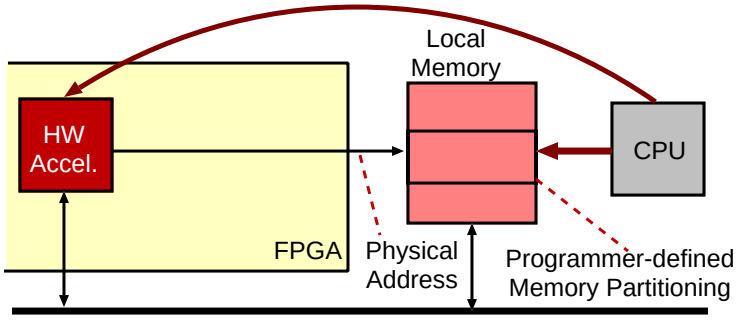
to resume the kernel execution of the VMW manager by calling a special system call *sys_resume*. The callback function returns to the trampoline code which, in turn, resumes the kernel execution (action ⑤ in Figure 6.3). The VMW manager is again in charge; it first copies the kernel frame back to the kernel stack (action ⑥ in Figure 6.3); then, the VMW manager passes the return value of the callback function and resumes the execution of the user hardware (action ⑦ in Figure 6.3). In this way, the transparent callback sequence is over and the user hardware can continue its computation.

The accelerator execution continues toward the next callback to software or the final completion and return from the *sys_hwacc* call (to the regular return address—labeled with *L1* in Figure 6.3—in the program flow). The programmer does not need any more to write wrapper code supporting the callbacks (in contrast to Figure 2.8 and Figure 6.2): it is now the responsibility of the VMW manager. The unified OS process for codesigned applications now becomes complete having the support for virtual memory abstraction and arbitrary execution transfers between heterogeneous application parts. Section 6.3 showcases how the unified OS process enables the unrestricted automated synthesis of hardware accelerators from high-level programming languages.

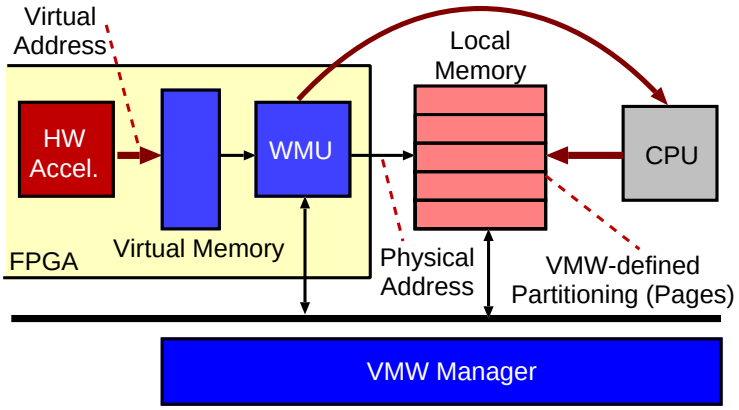
Some Implications. Having virtual-memory-enabled hardware accelerators and the execution transfers supported by the system, we can imagine hardware-centric applications (i.e., applications with its main execution thread in user hardware going back to software just for the system and library calls). Such applications largely use the hardwired logic and sparsely the CPU——at least at the user level. There could be many possible applications. One could imagine a video monitoring system with a video encoding engine and a detector, implemented completely in hardware but connected to a standard computer: a particular event triggers recording; in turn, the accelerator opens a regular file through the appropriate system call to save the recording. Remembering Figure 1.4 from the introductory chapter, our extensions have made such a system true: user hardware and user software are now peers, any of them can dominate in codesigned applications.

6.3 Enabling Unrestricted Automated Synthesis

Abstract language constructs and concepts bring major difficulties for synthesis of program parts written in high-level programming languages to hardware. Our goal is to attain unconstrained application partitioning and to be able to synthesise hardware without any restriction on the input program (unrestricted synthesis). The major problems are the following: (1) in a general case, memory access behaviour of a program is not known at synthesis time, thus, it is impossible to schedule memory transfers—servicing the accelerator accesses—in advance; (2) not all parts of the program can be efficiently implemented in hardware. In case a program part to be synthesised to hardware uses an aliased pointer [11], allocates memory on the heap, calls another function, or happens to call itself recursively, its mapping becomes more complex. Either additional analyses are required (e.g., either aliasing analysis [100] or memory partitioning [106]), or the synthesis is to be abandoned.



(a) Programmer-initiated Memory Transfers



(b) Accelerator-initiated Memory Transfers

Figure 6.4: Handling memory transfers for hardware accelerators: explicit programmer-scheduled (a) and transparent accelerator-initiated memory transfers (b). Programmer-initiated memory transfers require explicit knowledge on the memory access pattern of the accelerator. In the case where the accelerator initiates memory transfers, there is no such requirement.

The unified OS process for codesigned applications enables unrestricted automated synthesis from high-level programming languages. The system layer responsible for providing seamless interfacing and execution transfers fosters mobility of user code, be it from software to hardware or vice-versa. We apply our approach [119] within a virtual execution environment, such as *Java Virtual Machine* (JVM), to show the synthesis-related advantages of the unified OS process.

The abstraction provided by the unified OS process offers a solution to the previously-stated synthesis problems. Our primary concern is to support the unrestricted synthesis and not to explore the challenges of doing it efficiently—it is a well-known research area [35]. As Chapter 4 shows, the VMW manager allows dynamic handling of hardware-accelerator memory accesses. Whatever is the memory access pattern of the user hardware, the manager will ensure that the correct nonstale data is available to the accelerator. In contrast to the typical architectures where the programmer explicitly transfers the data to be processed, the VMW performs transparent accelerator-initiated handling of memory accesses (as Figure 6.4 shows) at runtime. Whatever is the memory access pattern, neither programmers

nor hardware designers need to know anything about. Beside arbitrary memory accesses, in Section 6.2 we have shown how the VMW manager handles execution transfers and hardware callbacks to arbitrary software functions. Synthesised accelerators running in the unified OS process immediately benefit from dynamic optimisations in the system software and system hardware (as Section 4.4 and Section 5.3 demonstrate).

6.3.1 Synthesis Flow

The unified OS process facilitates unrestricted automated synthesis [39, 122]. Sharing the same virtual memory, dynamically handling accelerator memory accesses, and having a standardised possibility to callback software enables synthesis of any Java bytecode method to reconfigurable hardware. Even sophisticated high-level language concepts such as object creation (recall the `malloc` example in Section 6.1), method invocation, and recursion, can be initiated by hardware and handled by the JVM. The presence of the VMW manager makes software to hardware migration easier. The migration could even happen dynamically, at runtime. Then, however, logic synthesis and place and route runtime become a major challenge [79]—these issues are beyond the scope of this thesis.

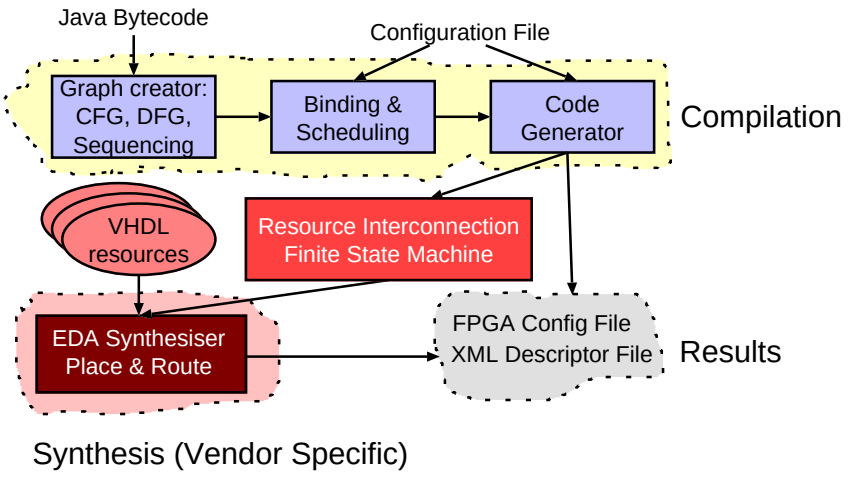


Figure 6.5: Automated unrestricted synthesis flow. The flow consists of typical high-level synthesis passes and the FPGA back end. An intermediate output of the compilation part is the VHDL description of the input Java method. The final outputs are the FPGA configuration file and the XML file containing the information on loading the modified Java class.

Figure 6.5 shows our basic synthesis flow (named Compilation in the figure) [39]. Its inputs are Java bytecode of critical methods and a compiler configuration file. The flow consists of typical high-level synthesis phases [35] such as: (1) sequencing graph construction; (2) resource binding and operation scheduling; (3) code construction. The generated VHDL code describes the interconnections between the resources and the finite state machines resulting from the scheduling step. An EDA

synthesis tool and the FPGA-vendor back end produce the bitstream file. An additional output of the flow is the XML descriptor file—specifying which methods are mapped to the FPGA.

6.3.2 Virtual Machine Integration

We run a JVM within our unified OS process to support transparent interfacing of software and hardware and to enable invoking the accelerators generated by our synthesis flow. Using the JNI interface [62] is a typical and JVM-independent way to extend the JVM. The JNI defines, for a class, a native library (e.g., written in C) that implements some methods of the class. We use an extended class loader and a JNI-based native library to call hardware accelerators from the Java bytecode.

Figure 6.6 shows the whole path from Java to hardware execution. An XML

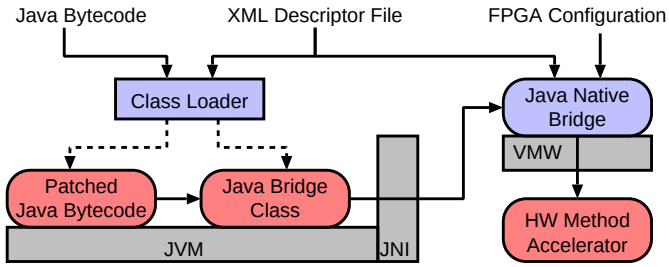


Figure 6.6: Steps involved in the execution of an accelerator from Java program. The class loader patches the original Java bytecode and inserts calls to the Java bridge class. The bridge class invokes a native function that relies on the VMW manager to launch the accelerator.

descriptor file and FPGA bitstream configuration accompany the Java bytecode of the application. The XML file defines which methods are to be executed in hardware and where the corresponding FPGA configurations reside. At class-invocation time, the class loader changes the Java bytecode of the application by replacing accelerated methods with calls to the corresponding native methods in the Java bridge class. Calling a native method, at execution time, invokes its native implementation through the JNI interface. The native implementation launches the hardware accelerator through the VMW manager. The described approach (1) is completely invisible for the programmer, (2) does not require changes in the original Java bytecode, and (3) does not depend on the used JVM.

Java Bridge Class. Our class loader constructs the Java bridge class dynamically at class-invocation time. It adds to the bridge class a native method prototype for each method to be accelerated in hardware. We use a predefined name (*VMWrun* in Figure 6.7) for all accelerated methods and we rely on the JVM capability to handle overloaded methods (same name but different number and type of arguments).

Java Native Bridge. The JVM links through JNI any native method of the bridge class to the same function in the native library. The application programmer is unaware of the existence of native methods (most Java system classes implement

```
/* IDEAEngine Java class */
void encrypt(
   byte[] data_in,
   byte[] data_out,
   int[] key
)
```

(a)

```
/* Bridge Java class */
   void VMWrun(
      int methodId,
      byte[] data_in,
      byte[] data_out,
      int[] key
   )
```

(b)

```
/* Bridge native library */
void VMWrun(
   int methodId,
   ...)  {
   sys_hwacc(HW_ACC, &param);
}
```

(c)

Figure 6.7: Invoking hardware accelerators from a Java application: the original method (a) is replaced with the call to a bridge class (b) which calls the native library (c) starting the accelerator.

their methods as native) and hardware accelerators. The native bridge code is responsible for (1) configuring the FPGA for the first use of an accelerator, (2) invoking the accelerator through *sys_hwacc* call, and (3) defining possible hardware-callback functions.

Parameter passing. The native implementation must handle different accelerators, neither their parameter type nor their number is known at compile time. After using the method identifier to read the number of parameters and their types from the XML descriptor file, it retrieves the parameters thanks to the *variable arguments* feature of C.

Accessing JVM objects. The native implementation has to interpret correctly the type of parameters received from the JVM. If necessary, conversions are done using an appropriate JNI function. For example, obtaining a native pointer for an object reference—there are no pointers in Java—requires calling the JNI function. This discipline also maintains memory consistency, since the garbage collector does not move the acquired objects until their release.

Accelerator Callback. To perform a callback, an accelerator uses the execution transfer sequence of the VMW (described in Section 6.2) to call the appropriate function of the JNI interface. The callback feature allows performing high level operations (e.g., object allocation, synchronisation, virtual method invocation) that require support of the JVM. This is essential when it comes to unrestricted automated synthesis from Java bytecode to hardware. Even if calling back software may be costly, the feature supports constructing hyperblocks [25] and allows mapping larger code sections from software to hardware. The expectation is that the callbacks are not so frequent in computation-intensive methods. When applicable, our flow also uses method inlining, to avoid unnecessary callbacks.

6.3.3 Experiments

The measurement results refer to the development board based on the Altera Excalibur device (refer to Appendix C). Kaffe [64], an open source virtual machine, runs in the JiT (*Just-in-Time* compilation) mode on top of a unified GNU/Linux process for codesigned applications. We do not measure the time for FPGA configuration; although this time may be significant when compared to a single program execution, we assume the accelerated program is run multiple times for a longer period of time; in this case the configuration time becomes irrelevant.

Synthesis Results. Figure 6.8 shows the execution time for two accelerators synthesised (using the flow shown in Figure 6.5 and described in detail in [39]) directly from the critical Java bytecode methods of two Java applications (the IDEA cryptography and the ADPCM voice decoding application). The execution times are compared to pure Java bytecode executed in the JiT-enabled JVM, pure SW compiled to the machine binary code, and a handwritten virtual-memory-based accelerator. The current synthesis flow lacks some common optimisation passes to improve performance and exploit hardware parallelism. On the other side, relying on the unified OS process, it can map arbitrary Java bytecode to hardware. Although not as fast as handwritten, the synthesised accelerators provide performance improvements, in comparison to the JVM with JiT. Furthermore, the performance improvement comes for free since there is neither programmer nor designer intervention: the synthesis flow produces the FPGA configuration and the XML descriptor file automatically.

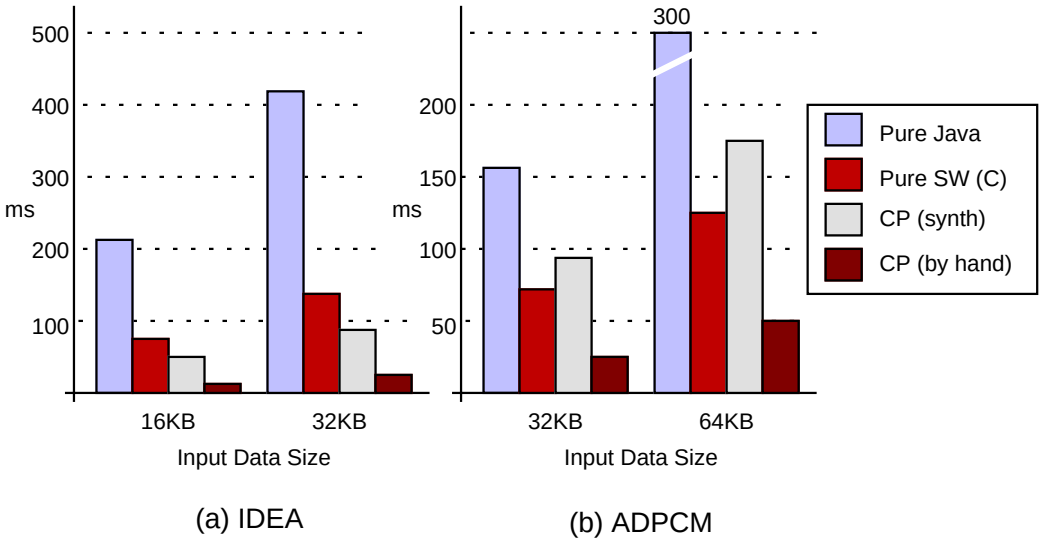


Figure 6.8: Execution times of synthesised IDEA and ADPCM accelerators compared to Java, C, and handwritten accelerators. Although the handwritten accelerators show superior performance, the synthesised accelerators offer automated improvements of pure Java execution.

The complexity of the synthesised accelerators is comparable to the complexity of the handwritten ones. For example, the synthesised IDEA accelerator occupies slightly more reconfigurable logic than the handwritten IDEA (4100 logic cells compared to 3600). Nevertheless, the logic in the handwritten accelerator is used more efficiently, as the performance figures indicate. Additional compiler passes should address improving the logic efficiency and the overall performance.

Overhead Measurement. In Figure 6.9, we show the overhead of invoking the IDEA accelerator for small input data sizes (one IDEA block equals 8 bytes)—execution time of the pure Java version against the execution time of the accelerated version. We measure several time components of the accelerator execution time. The first is the overhead introduced by the Java bridge and its native library (parsing

the XML descriptor file and creating bridge class dynamically). Two thirds of the overhead are coming from the XML configuration file reading; by optimising this operation, the overhead could potentially go below 200μs. The other observable time components relate to the VMW interface (recall *Manage time*, *Copy time*, *Hardware time* from Section 5.2). The sum of the manage time and the copy time is constant, since for any data input size in the graph, only one page of memory is used (the page size is 2KB). Only the hardware time, which is real computation time, is affected by input size.

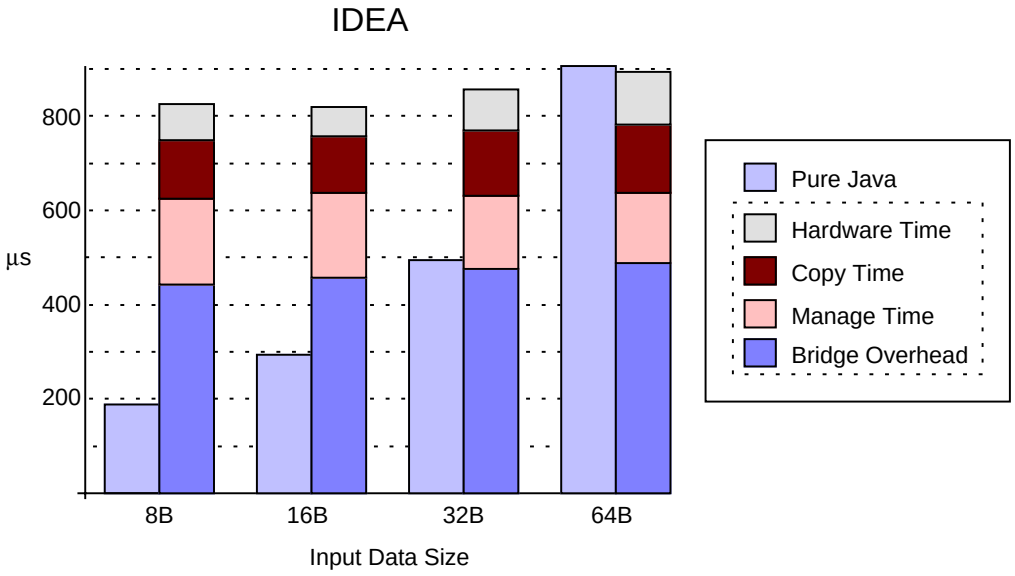


Figure 6.9: Invocation overhead for small data sizes. IDEA encryption of 8 bytes to 64 bytes (one to eight 64-bit blocks) of input data. The Java bridge overhead is considerable.

For the two first data input sizes (8 bytes and 16 bytes), the time is almost the same. In fact, this is due to the way that the IDEA accelerator works (as the graphs from Appendix E clarify): it has a three-stage pipeline that encrypts 3 blocks at a time; thus, the time required to compute 1, 2, or 3 blocks (8, 16, or 24 bytes) is more or less the same. Since the core computation is performed three times, one can see a difference for encryption of 8 blocks.

For 64 bytes (8 blocks), the accelerator version becomes faster than pure Java software, despite the introduced overhead. The same behaviour, with a different break-even point, is expected for other applications having similar memory access patterns. If we recall that the overhead could be reduced (since the current implementation is not optimal), the break-even point can move toward even smaller datasets. What is more important, the total overhead (750μs) becomes negligible for typical input data sizes, especially when one recalls the benefits it brings: transparent and platform-independent acceleration of Java programs.

6.3.4 Summary

The WMU system hardware and the VMW system software enable mapping any virtual machine code to hardware and support portability of codesigned applications. Running a JVM and Java applications within the unified OS process allows transparent use of hardware accelerators to programmers—no change is required in the application software. Our unrestricted synthesis flow for JVM platforms [39, 61] allows handling high-level language concepts such as object creation and method invocation. Further improvements of the flow shall employ typical synthesis optimisations and exploit better the available hardware parallelism.

Chapter 7

Transparent Software and Hardware Multithreading

> As soon as you perceive an object, you draw a line between it and the rest of the world; you divide the world, artificially, into parts, and you thereby miss the Way.
>
> —Douglas Hofstadter, *Gödel, Escher, Bach*

IN this chapter we discuss the extension of our unified OS process for codesigned applications to support multithreaded execution of user software and user hardware. Programmers use threads to exploit application parallelism without the cost of having multiple processes communicating to each other. Creating threads costs less than creating processes; multiple threads within a user process share the same memory address space and some per-process OS data structures. Using threads can be fruitful even in uniprocessor systems; before blocking in a system call, a thread can create another thread that continues the execution; thus, the process does not yield the CPU to another process.

Having unified memory abstraction between user software and user hardware is essential for supporting multithreaded programming paradigms. In the following sections, we present the extensions to our baseline system shown in Chapter 4 to support multithreaded execution. Figure 7.1 shows the architecture we are targeting for multithreading. Our implementation provides an initial infrastructure for simultaneous execution of multiple virtual-memory-enabled hardware accelerators but, it is not necessarily the most efficient one. Our goal—in the first place we are targeting reconfigurable SoCs—is to show feasibility of the unified OS process extensions to support multithreading; for addressing efficiency, scalability, and future improvements, the extensive existing literature [95] in this field should be used.

7.1 Extending a Multithreaded Programming Paradigm

In this section, we build a multithreaded programming paradigm for codesigned applications based on the standard multithreaded programming model—POSIX threads [88]. Relying on the POSIX thread library and the OS support for inter-

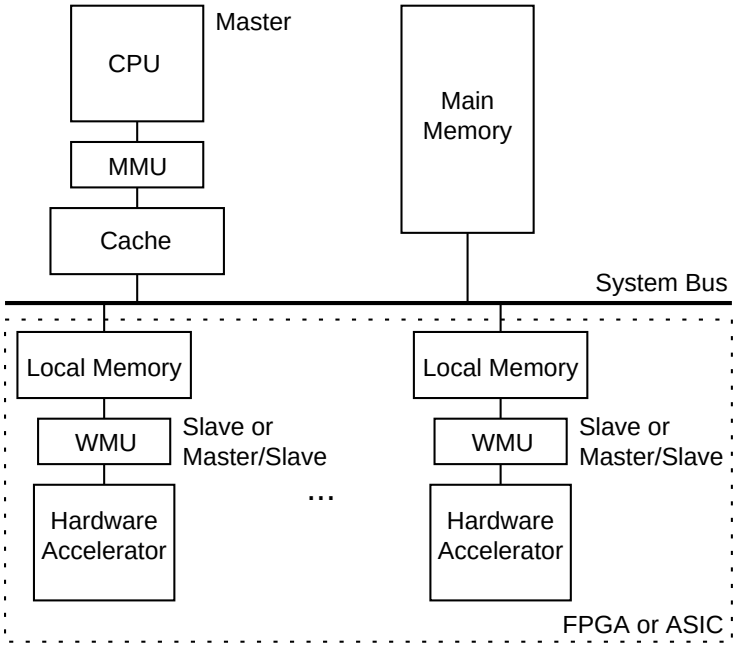
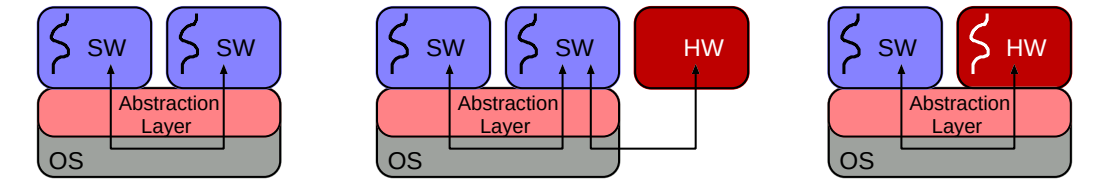


Figure 7.1: Multithreaded virtual-memory-enabled hardware accelerators. Software and hardware threads run in the context of the unified OS process. Each accelerator executes a hardware thread. Software threads execute on the CPU.

thread communication and synchronisation, software-only POSIX threads execute within the context of a common OS process.

With virtual-memory-enabled hardware accelerators, there is no need for wrapper threads (as we needed them in Section 2.4). Similarly to what we do in the case of having a single hardware accelerator, we delegate the accelerator control and data transfers to the operating system. Once again, we keep the programmer interface to hardware clean and elegant. The software and hardware threads share the same virtual memory address space, and the operating system manages virtual to physical memory address translation for both software and hardware threads.



(a) Standard Multithreading (b) Including Hardware Accelerators (c) Extended Multithreading

Figure 7.2: Standard (a), wrapper-based (b), and extended multithreading of user software and user hardware. A multithreading library and OS services support standard multithreading (a). Such support does not cover including hardware accelerators in multithreaded applications (b). Our extensions provide a unified OS process for codesigned applications capable of mixed software-and-hardware multithreading (c).

Figure 7.2 summarises our motivation and illustrates the multithreading of user software and user hardware within an OS process. In standard multithreading (Fig-

ure 7.2a), simultaneous execution of software threads is provided by an abstraction layer usually consisting of thread libraries (e.g., POSIX threads) supported by OS services. Nevertheless, the existing abstraction layers (Figure 7.2b) do not support integrating hardware accelerators with software, and wrapper threads (as Figure 2.9b in Section 2.4 shows) are typically used to perform accelerator control and data transfers. If the layer is extended (Figure 7.2c) to support hardware threads (i.e., to enable communication, synchronisation and sharing of virtual memory address space), programming is improved and brought to the level of software-only applications: programmers and hardware designers can seamlessly integrate software and hardware application parts. In the following section we explain how the system layer can provide the multithreaded programming paradigm for codesigned applications.

7.2 Support for Extended Multithreading

To achieve the transparent multithreading of user software and user hardware within an OS process (as presented in Figure 7.1), we need to extend (1) the existing WMU for supporting the coherence of the WMU local memories, and (2) the VMW manager for steering multiple WMUs and ensuring memory consistency.

7.2.1 WMU Extensions

Having multiple virtual-memory-enabled hardware accelerators that run in parallel accessing their local memories may violate data integrity. To enable the hardware accelerators to share the coherent memory and to enforce strict memory consistency, we extend the basic WMUs with a simple, invalidation-based, mixed hardware-and-software coherency protocol [95].

Figure 7.3 shows multiple WMUs connected to an internal, custom coherence bus, separated from the system one. An arbiter accessible and controllable by the OS through the system bus is also connected to the coherence bus. Each WMU keeps track of states for the pages residing in the local memory and snoops on the coherence bus (the protocol diagram is shown in Figure 7.4). Bus-incoming stimuli, OS events, or hardware accelerator actions (written in italics in Figure 7.4) initiate state transitions. Either the OS or the WMUs are responsible for the transitions. A state transition for a particular WMU is sometimes followed by a bus action—this is the outcome (written in boldface) of the transition. The **Copy In** and **Write Back** actions happen on the system bus: (1) **Copy In** means that a missing page is brought to the local memory either from the main memory or from the local memory of another hardware accelerator; (2) **Write Back** means that a dirty page is copied back to the main memory. The **Invalidate** action happens on the consistency bus. Prior to writing to a shared page, the originating WMU sends *Invalidate* stimulus to other WMUs sharing the page.

The protocol divides the responsibility for the actions between software (OS) and hardware (WMU). Prior to a write access to a shared page, the originating WMU has first to obtain access to the bus from the arbiter. After informing other WMUs

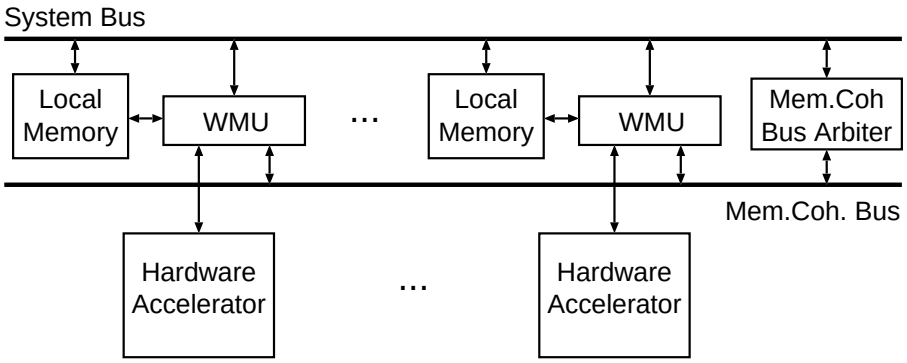


Figure 7.3: Memory coherence support for multiple hardware accelerators. Beside their connections to the system bus, the local memories, and hardware accelerators, the WMUs are connected to a memory coherence bus. The WMUs take special care of shared pages. A hardware accelerator writing to a shared page stimulates its WMU to perform an invalidation transaction on the coherence bus. On the other side, snooping activities on the coherence bus initiate invalidation of locally-stored shared pages written by another hardware accelerator. The arbiter provides to WMUs the serialised access to the bus.

sharing the page to invalidate it, it changes the internal state of the page TLB entry to exclusive. Read accesses to a shared page need no arbitration, but require bus snooping before the data is sent to the accelerator. Read and write accesses to an exclusive or modified page need no activity on the coherency snooping bus.

The only transitions managed by the hardware are: (1) from `Shared` to `Invalid`; (2) from `Shared` to `Modified`; and (3) from `Exclusive` to `Modified` (the transition does not incur any snoopy-bus activity). The OS manages the rest of the transitions from Figure 7.4. The OS-prevailing management for page-level coherency can definitely have negative performance impacts, but in the absence of heavy data sharing may be sufficient [74]. The bus arbiter is also managed by the OS. It provides a way to block the snooping bus, during a critical OS activity regarding the memory consistency. In addition to memory coherence, the WMU interface provides atomic test-and-set operations—hardware accelerators can use such accesses for synchronisation purposes.

7.2.2 VMW Manager Extensions

In the case of multiple hardware accelerators executing in parallel, the manager keeps track of all their translation data structures, manages the translation, and ensures the memory consistency. Figures 7.5 and 7.6 show all transitions from Figure 7.4 that the OS performs. As it services multiple WMUs, the OS can be a potential bottleneck. A distributed VMW manager could be a possible solution, if large scalability is needed. For the moment, we have used only applications with few hardware accelerators and the centralised VMW is sufficient.

Recalling Figure 4.4 from Chapter 4, Figure 7.5 extends the control branch of the interrupt handler designated to handle memory page faults (the branch named

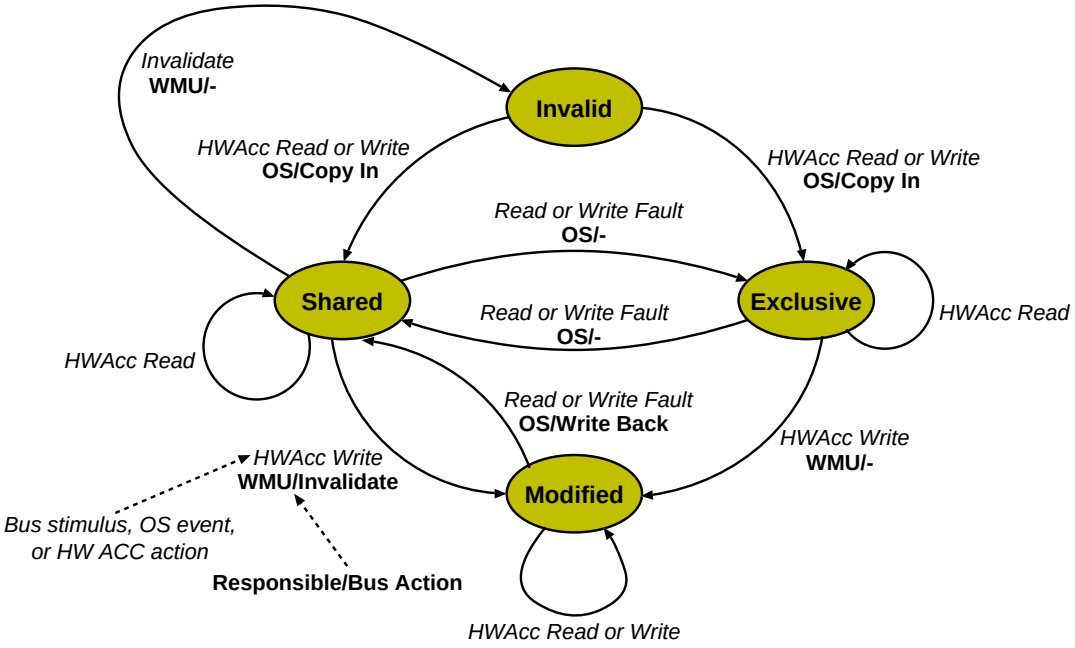


Figure 7.4: State diagram for page-level memory coherence protocol. Based on stimuli coming from the coherence bus, OS events, or hardware accelerator actions, the WMU and the VMW manager (mixed software-and-hardware approach) perform state transitions. Bus actions (Invalidate on the coherence bus done by the WMU; Copy In and Write Back on the system bus done by the OS) follow some of the transitions.

Handle Fault in Figure 4.4). On a page fault, the manager checks if the page is not already present in any local memory (as Figure 7.5 shows). If the page is not found in any other WMU, the manager copies it from the main memory and sets the state of the TLB entry to `Exclusive`. If, on the contrary, some other WMU has the page (which the manager can determine from its internal data structures), the manager chooses a TLB entry containing the page and checks in its corresponding WMU if the page is shared. Before copying in the shared page, the manager blocks the coherency bus to disable undesirable transitions from `Shared` to `Modified`. After the copying is over, it sets the TLB entry of the copied page to `Shared` and unblocks the bus. If another WMU has the page in the `Exclusive` state, before copying the shared page, the manager has to put on hold the WMU currently containing the page. After the copying is over, it sets the TLB entry to the `Shared` state, transitions the TLB entry state of the source WMU to `Shared`, and resumes the source WMU immediately afterward. The VMW manager takes similar actions if the page is `Modified`, except that it also copies it back to the main memory.

When evicting a page, the VMW manager performs the actions shown in Figure 7.6—evicting a page is more complex in the VMW manager for multithreaded accelerators than in the basic one (shown in Figure 4.4). At the beginning, it saves the current state of the page TLB entry. Then it invalidates the entry thus disabling further references to the page. If the saved state shows the page was modified, it writes it back to the main memory. There is no need to change states of any other WMU, since only one WMU can have the page in the `Modified` state. On the

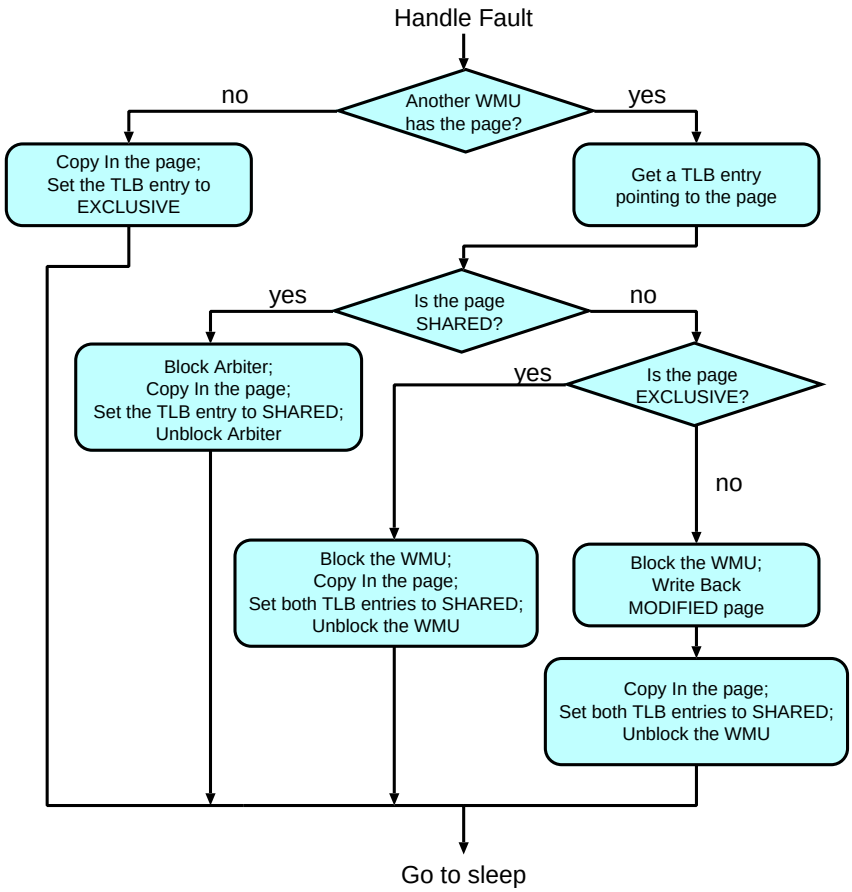


Figure 7.5: VMW manager actions on a memory fault of a multithreaded accelerator. In contrast to the basic VMW manager, there are multiple checks to be done.

contrary, if the page is in some other state than `Modified`, some additional checks are required (as Figure 7.6 shows) for performing possible state transitions in other WMUs.

Another feature of the VMW manager enables consistent memory between user hardware and user software (since they can execute in parallel). If user software tries accessing a page currently present in one of the local memories, the OS blocks the process and puts it on a wait queue. Writing back the modified page and invalidating the appropriate TLB entry initiates the wake up of all sleeping software threads waiting on the page. Although costly (especially because false sharing at the page-level granularity can easily degrade performance), the approach ensures basic memory consistency for applications with no heavy memory sharing. The solution is in favour of user hardware. To overcome this, the VMW also provides relaxed models that avoid blocking but expose the synchronisation tasks to the programmer [52].

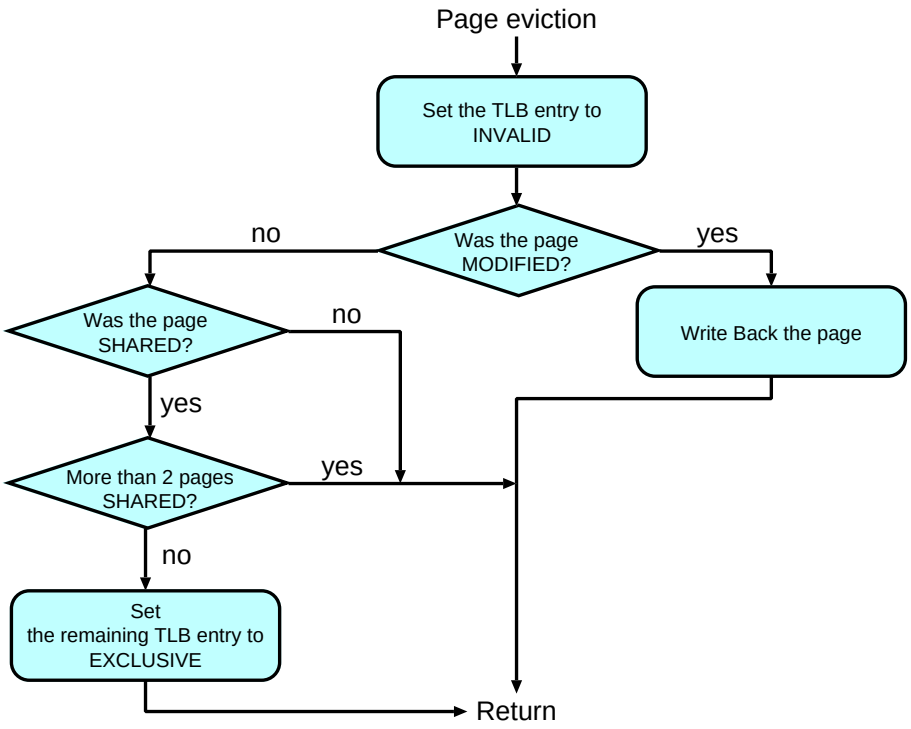


Figure 7.6: VMW manager actions on evicting a page from the local memory. The fact that the page may be shared complicates the procedure. Evicting a shared pages may require changing the state of other WMUs containing the page.

7.3 Multithreading Case Study

Having a large amount of reconfigurable logic allows unloading the main processor and executing multiple hardware accelerators in parallel. However, in our approach, beside running software threads, the CPU is responsible for the execution support of hardware threads (through the VMW activities). Thus, having too many hardware accelerators would make the CPU become a bottleneck: our approach is limited but it trades off scalability for applicability to wide range of reconfigurable SoCs—there is no need for a CPU and system bus memory coherence support in the original system. Nevertheless, in our case studies we use applications with just few WMUs. The CPU involvement is moderate while the performance improvement achieved is significant. The scalability could be improved by applying some of existing decentralised schemes [95] or by delegating more responsibilities to system hardware and, in this way, discharging the CPU.

We use a multithreaded image processing application to demonstrate our virtual memory manager supporting simultaneous execution of multiple hardware accelerators. The application threads are organised in a producer/consumer chain, as it is shown in Figure 7.7. This allows the contrast and edge detection accelerators to work in parallel, on different image windows slid in time. Their synchronisation is achieved by the use of semaphores. We compare the results obtained for the code-signed application with a multithreaded, software-only version of the application.

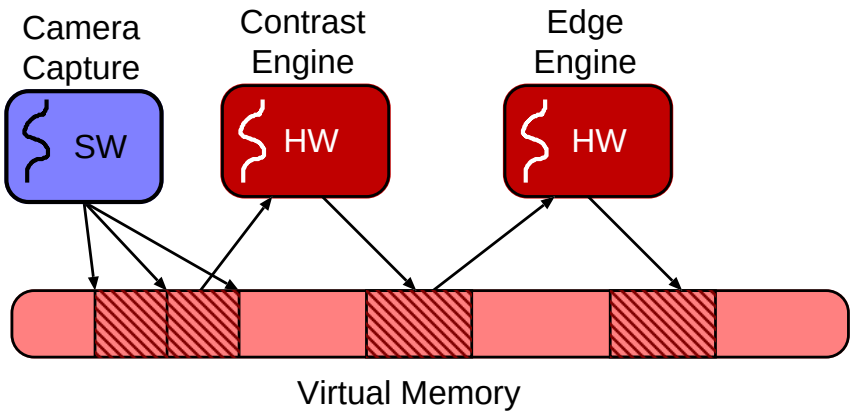


Figure 7.7: Producer/consumer chain of threads for edge detection.

7.3.1 Multithreaded Execution

Figure 7.8 and Figure 7.9 show executions times of two image processing hardware accelerators (contrast enhancement and edge detection [112]) processing a window (region of interest) within an image (as Figure 5.2 shows). Both accelerators are designed and run on the Xilinx ML310 platform (refer to Appendix C for more details about the platform). We show their design and implementation details in Appendix D. To study their characteristics, we first consider the accelerators running separately. Later on, we put them together in the multithreaded application illustrated in Figure 7.7. The typical codesigned applications use hardware accelerators directly accessing the main memory (as explained in Figure 2.5 and in Figure 2.6b). In the case of virtual-memory-based hardware accelerators, we use the local memory of 64KB organised into 8 pages (the bar in the middle for all processing window sizes) or 16 pages (two leftmost bars for all processing window sizes). The OS module uses DMA to transfer the pages to the local memory. In this way, the end user benefits from the improved performance but is completely screened of the enhanced system-level support.

The typical accelerator in Figure 7.8 does not implement burst accesses to the memory which affects its performance. Despite the page-level granularity overhead, the virtual-memory-enabled accelerators outperform the typical solution. More importantly, the benefit comes for free for the hardware designer: there is no need to implement and manage memory transfer bursts.

The typical accelerator in Figure 7.9 implements burst accesses to the memory and local caching of the transferred data. It is significantly faster than the virtual-memory-enabled accelerator for the smallest window size but, for larger window sizes, the two approaches get close performance figures; yet, the WMU memory interface is much simpler (no burst and no pipelining support). It is a simpler way for a designer (thanks to shifting the interfacing and data transfer burden to the system level) to get results comparable with typical approaches.

We put the contrast enhancement and edge detection engines together in the multithreaded application from Figure 7.7, consisting of one software and two hardware threads. We use the multithreaded image processing application to demonstrate our

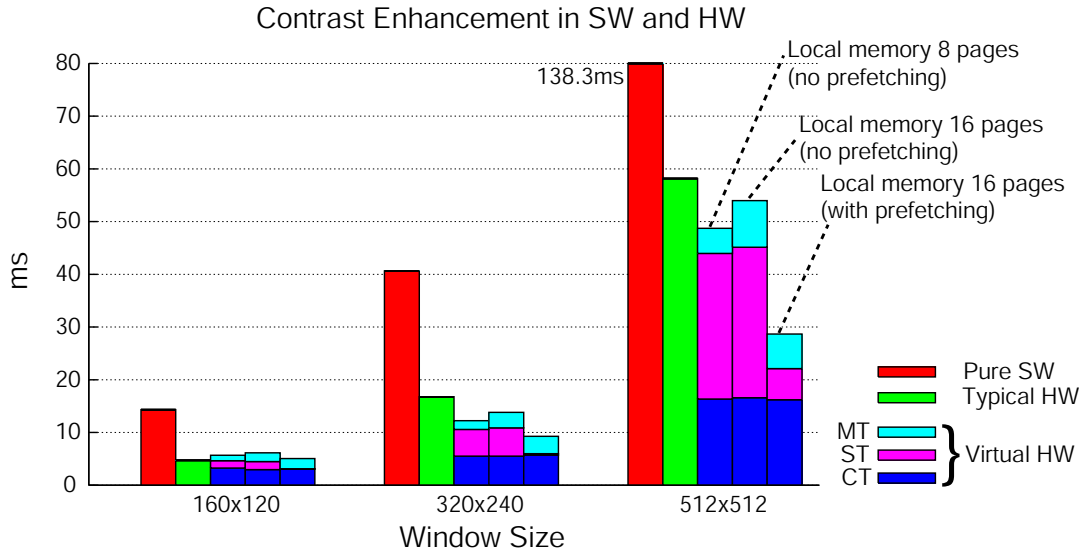


Figure 7.8: Execution times of contrast-enhancement application implemented in pure software, typical hardware, and virtual memory-enabled hardware.

virtual memory manager supporting simultaneous execution of multiple hardware accelerators. We set up the application threads (shown in Figure 7.7) in a producer/consumer chain that allows the contrast and edge detection accelerators to work in parallel, on different image windows slid in time. We compare the results obtained for software-only version of the application and the codesigned applications with typical or virtual-memory-enabled accelerators.

Apart from the significant performance improvement obtained by the multithreaded codesigned version (visible in Figure 7.10), we stress the programming simplicity of our approach: our threads operate on images stored on the heap; we pass to our hardware memory pointers obtained by the *malloc()* function call; there is no need for software wrappers responsible for memory transfers. One can notice that the codesigned application with multithreading support can process more than 50 image windows per second with a large safety margin—there is space left for other applications to run in the system simultaneously.

Our approach outperforms the typical solution and, even more remarkably, this benefit comes along with our simple and transparent design paradigm. On the other side, the typical solution could have achieved better performance, but with much more effort invested on the programmer and designer side. In our case, we delegate interfacing burdens to the system layer and, thus, preserve simplicity of interfacing; yet, our approach achieves better performance with less design efforts. The execution time of the typical solution is dominated by slower contrast-detection accelerator and, the lack of local memories also degrades the performance by increasing the bus contention. Our virtualisation layer hides from designers relatively complex design issues of burst accesses and local memory management (either in software or hardware) and, still, may offer better performance.

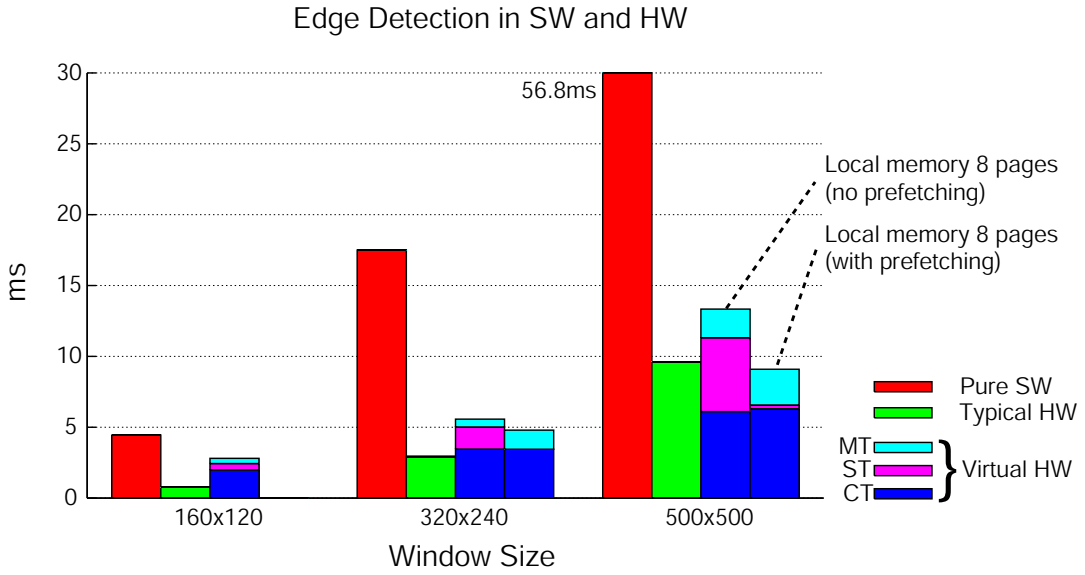


Figure 7.9: Execution times of edge-detection application implemented in pure software, typical hardware, and virtual memory-enabled hardware.

7.3.2 Multiple Memory Ports

The VMW manager capable of handling multiple local memories and WMUs offers another design possibility. In place of having multithreaded hardware accelerators, the hardware designer can use several WMU interfaces to open multiple memory ports for a single hardware accelerator. In this section, we analyse such a possibility for the contrast engine using the CPD metric.

Figure 7.11 shows the contrast enhancement engine with four input and one output ports. For the image size of 512×512 pixels, the engine (Appendix D shows its internal design blocks) operates on four processing windows (contained by the four input images) to produce one output window (contained by the output image). Once started, the engine first samples four input subwindows to set up the internal thresholds. At this phase, it only reads from the inputs without writing anything to the output. In the second phase, the engine enhances the contrast of the input windows and writes the results to the output window. Notably, there are four input and one output data streams in this phase.

We consider the two phases as two separate accelerators and write the following performance equation:

$$\begin{aligned} ET_{HW} &= DC_1 \times (CPD_{virtual1} + CPD_{vmemory1}) \times T_{HW} \\ &\quad + DC_2 \times (CPD_{virtual2} + CPD_{vmemory2}) \times T_{HW}, \end{aligned} \tag{7.1}$$

where $DC_{\{1,2\}}$, $CPD_{virtual\{1,2\}}$, and $CPD_{vmemory\{1,2\}}$ represent the parameters DC, $CPD_{virtual}$, and $CPD_{vmemory}$ of the phases one and two respectively (recall Section 5.2). As Figure 7.8 shows, the execution time for a processing window of 320×240 pixels (inherently to the algorithm implemented by the engine, this deter-

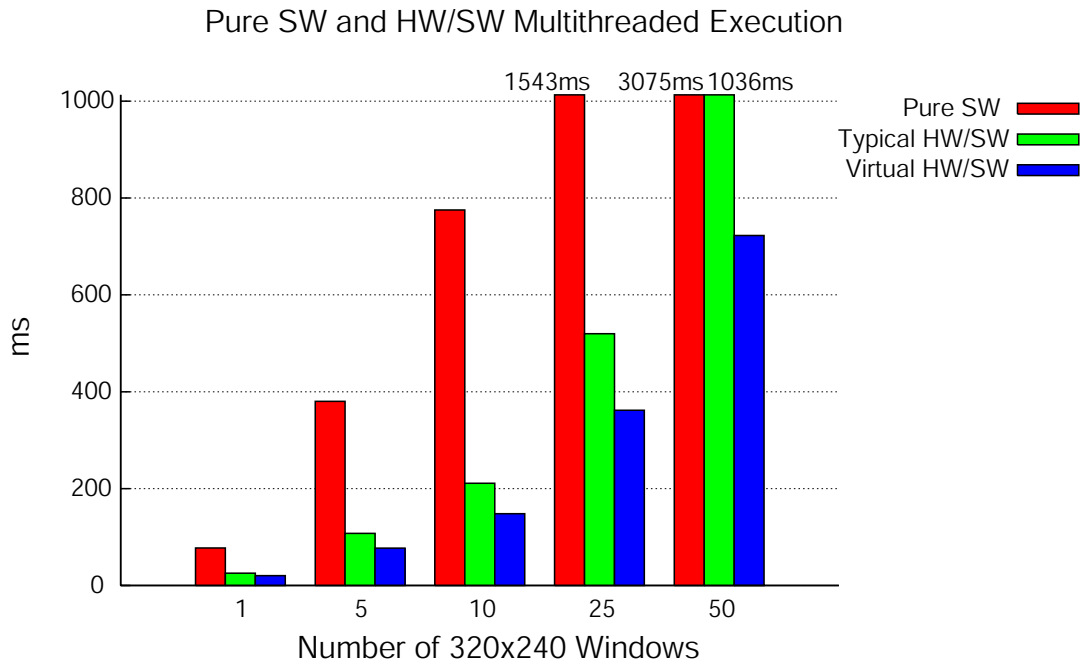


Figure 7.10: Execution times for multithreaded versions of SW-only and virtual memory-based codesigned HW/SW applications. The multithreaded application running within the unified OS process outperforms the pure software and typical versions of the application.

mines the sampling subwindow of 160×80 pixels[1]) is $ET_{HW} = 12.245ms$. The CPD values for the accelerator phases are (from Appendix E) $CPD_{virtual1} = 4.5$ and $CPD_{virtual2} = 5.625$. Regarding the memory hierarchy overhead ($CPD_{vmemory}$), we take an assumption that $CPD_{vmemory1} = 2 \cdot CPD_{vmemory2}$ (Equation 5.9 can show—based on measurement results from Figure 7.8—that the introduced error is less than 15%). With this assumtion, and with $T_{HW} = 10ns$, we find out the memory overheads $CPD_{virtual1} = 15.61$ and $CPD_{virtual2} = 7.805$.

The hardware designer can use multiple memory ports for the contrast enhancement engine, as Figure 7.11 shows. The CPD metric provides a quick estimation of the possible benefit. With four input and one output ports the corresponding CPDs would be smaller ($CPD_{virtual1} = 2.25$ and $CPD_{virtual2} = 2.625$, refer to Appendix E for details) than the original ones and could offer faster execution. However, the clock period (T_{HW}) might also become slower, because the synchronisation logic between the memory ports becomes more complex. To estimate the final impact of having multiple ports to the memory, the designer can use Equation 7.1 with new $CPD_{virtual1}$ and $CPD_{virtual2}$ values, assuming that $CPD_{vmemory1}$ and $CPD_{vmemory2}$ remain the same (a pessimistic assumption, since multiple local memories eliminate possible contentions between input and output data streams, thus, making the memory overhead smaller). For example, for a 10% longer clock period, Equation 7.1

[1]The sizes of the processing window and the sampling window define $DC_1 = (320 \times 240 \times 4)/4 = 76800$ words and $DC_2 = (160 \times 80 \times 4)/4 = 12800$ words, having in mind that there are four input streams and the memory word contains four pixels.

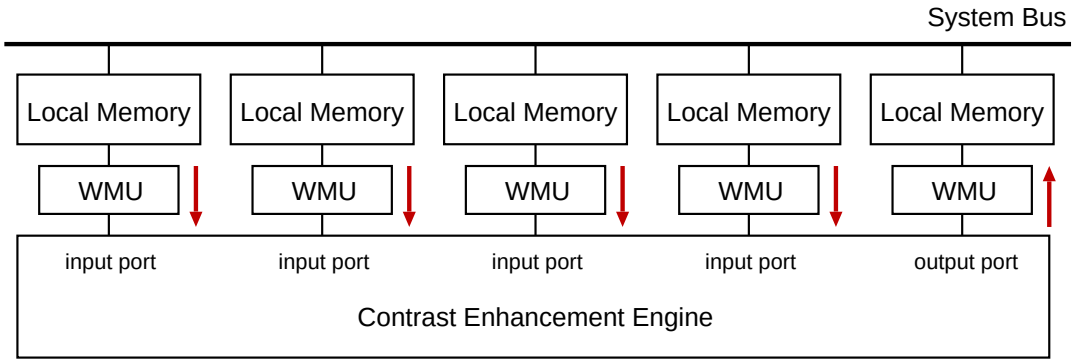


Figure 7.11: Contrast enhancement engine with multiple memory ports. The accelerator uses four WMU interfaces for its input ports and one WMU interface for the output port. The VMW manager handles them as if they were ports of multithreaded hardware accelerators.

gives the estimation for the execution time of the contrast engine (with four input and one output ports) $ET_{HW} = 10.697ms$. The designer can use the estimation (the execution time improvement from $12.245ms$ to $10.697ms$) for deciding whether or not it justifies undertaking more complex design efforts.

For the purpose of another comparison, the designer may use the CPD metric for estimating how much a faster WMU translation could improve the performance. Assuming one cycle per accelerator access through the improved WMU, the CPDs would be $CPD_{virtual1} = 2$ and $CPD_{virtual2} = 2.375$. Preserving the original memory hierarchy (with only one memory port) but having the faster translation would result in the overall execution time $ET_{HW} = 9.51ms$. The CPD metric can be a convenient way for the hardware designer to estimate performance before taking design decisions.

Chapter 8

Conclusions

A Elbereth Gilthoniel,
Silivren penna míriel
O menel aglar elenath!
Na-chaered palan-díriel
O galadhremmin ennorath,
Fanuilos, le linnathon
Nef aear, sí nef aearon!

O! Elbereth who lit the stars
From glittering crystal slanting falls with light like jewels
From heaven on high the glory of the starry host.
To lands remote I have looked afar,
And now to thee, Fanuilos, bright spirit clothed in ever-white,
I here will sing Beyond the Sea, beyond the wide and sundering Sea.

—Tolkien, *The Hymn to Elbereth*

THE message of this thesis regarding application-specific hardware accelerators (i.e., user hardware, semantically belonging to a particular user application) and their integration within codesigned applications is: To simplify the interfacing between software and hardware and to make any programming paradigm feasible, one shall provide user hardware with system-level abstractions (e.g., virtual memory, system calls) and execution support (e.g., memory-hierarchy management, execution transfers) already available to user software; in comparison to existing approaches, our solution is a general one—there is no limitation on the memory access patterns of hardware accelerators.

Delegating the execution steering of hardware accelerators to the system level exposes to the user level only their pure functionality. The separation of user-level from system-level tasks fosters system-level runtime optimisations performed transparently to the end users. It also eases software-to-hardware migrations and enables unrestricted automated hardware synthesis from high-level programming languages. In addition, codesigned applications become portable across different platforms.

Approaching to the principal concepts of general-purpose computing may seem to be compromising the performance and efficiency potential of spatial computation. Notwithstanding such expectations, most of the concepts—when applied to

application-specific hardware—rather become simpler and less costly than in the general-purpose case. For example, having a virtual memory hierarchy per accelerator avoids unwanted effects of cache pollution and frequent context-switches. Furthermore, memory access patterns of hardware accelerators are less general and more predictable in comparison to the general-purpose case: the size and the management policy of the memory hierarchy can be adapted to a particular application for lower cost and better performance.

What follows in this passage is a recommendation for future architects and designers of codesigned applications. When combining two different models of computation (such as temporal computation of a CPU and spatial computation of an application-specific hardware accelerator, both being addressed in this thesis), one should find a common background for the two different worlds: we have chosen the virtual memory abstraction and the execution context of an OS process for this purpose. The next step would be to implement the appropriate support for the common abstraction in the system software and system hardware—not directly visible to the application programmer. In our case, we have shown an implementation (especially applicable to reconfigurable SoCs) of the system software and system hardware that provides the unified process context for codesigned applications. When designing hardware accelerators or when automating this task (by developing a synthesiser), one should apply existing (or establish, if nonexistent) conventions regarding the chosen programming paradigm. In such a way, as presented in this thesis, we follow the API and ABI rules of the host OS and the host CPU to have host-compliant hardware accelerators. Finally, following these guidelines (as the thesis demonstrates) gets software programmers and hardware designers to the seamless interfacing of software and hardware components of codesigned applications.

8.1 Obtained Results

For a chosen, process-based, multithreading computation model, we have proposed extensions to system software and system hardware to support unified OS processes for heterogeneous-code applications. In experiments performed on two different reconfigurable SoCs running GNU/Linux, we have demonstrated the viability of our approach. Our architecture (1) provides a straightforward programming paradigm—*programmers are completely screened from interfacing-related memory transfers*, (2) makes codesigned applications completely portable—*recompiling and resynthesising is sufficient*, and (3) enables advanced and yet simple runtime optimisations—*without any change in either application software or coprocessor hardware*—and unrestricted automated synthesis.

To quantify the overhead incurred by the system-level support for unified memory abstraction, we have introduced a metric for application-specific hardware accelerators and applied it to our particular implementation. To measure overall benefits and check the overhead, we have tested the approach by porting and running several applications with application-specific coprocessors of different complexity. The results show the overhead for unified memory abstraction is generally limited. The hardware accelerators achieve significant speedups compared to software-only ex-

ecution. While our approach provides simpler interfacing, software programming, and hardware design, compared to typical existing solutions, the speedups achieved by the two approaches remain comparable.

System-level runtime optimisations can improve performance transparently to the end user. To demonstrate a possible runtime optimisation, we have implemented a stream-based memory prefetch technique within the VMW manager (with a simple hardware support in the WMU). A significant execution time improvement is demonstrated, *without any change in either application software or coprocessor hardware.*

We have also shown a basic implementation of an unrestricted synthesis flow. Although at an early stage and without typical optimisation passes, our flow produces accelerators that can speedup high-level language code without any restrictions on the language constructions.

8.2 Implications

Designing the execution support and memory hierarchy for standard CPUs is a complex task—the challenge is to achieve performance and cost goals for all applications. On the contrary, designing the execution support and memory hierarchy for application-specific hardware accelerators (especially having the reconfigurable hardware available) is simpler—to support the unified OS process, we keep the high abstractions from general-purpose computing but simplify the system hardware and system software. For example, opening more memory ports with smaller local memories can be more performance-efficient than investing in a single and large local memory with complex management policies.

Having virtual-memory-enabled hardware accelerators and the execution transfers supported by the system, we can imagine hardware-centric applications (i.e., applications with its main execution thread in user hardware going back to software just for the system and library calls). In such a scenario, the CPU becomes responsible for execution support and memory hierarchy management, running mostly on behalf of the user hardware (e.g., executing the VMW manager actions and system or library calls invoked by the accelerator).

Within a unified OS process, user hardware gains the access to static and dynamic data (residing either on the stack or on the heap) of its software counterpart. The unified memory simplifies preserving constructions of high-level programming languages when mapping the application code from software to hardware. Any scope policy (e.g., either static or dynamic scope) can be implemented through the shared stack, any memory-allocation policy can be implemented through the shared heap. Supporting the code mobility and enabling unrestricted software-to-hardware migration opens the way toward future dynamic adaptive systems, capable of implementing runtime (Just-in-Time) synthesis and reconfiguration.

8.3 Future Research Directions

The thesis opens new research directions (1) in configurable memory hierarchies for application-specific hardware accelerators and (2) in automatic high-level synthesis from high-level programming languages.

Here we mention the differences between application-specific and general-purpose virtual memory systems that future research may exploit. In the software-only case, system architects design MMUs and memory hierarchies for the most general application mix—multiple and diverse software tasks share the MMU and the memory hierarchy in the time-multiplexed manner. In the case of virtual-memory-enabled hardware accelerators, one can assume that a hardware accelerator owns the WMU and the memory hierarchy during the application lifetime—considerably longer period than between two context switches on a standard CPU. This fact allows (1) designing the WMU which is less-costly than standard MMUs, and (2) employing the WMU reconfigurability to tailor the translation and the memory hierarchy closer to the application needs. The second point is especially challenging and implies the appropriate support from novel OS features to adapt the WMU and memory hierarchy dynamically, at runtime. For example, in our experiments, we support different sizes of the local memories accessed by hardware accelerators; also, the WMUs support different sizes of virtual memory pages; however, for the moment, our OS extension does not adapt the memory automatically; it is a subject of future work.

Having a configurable memory hierarchy supported by the OS not only allows runtime customisation of the virtual memory for hardware accelerators, but also supports OS-conducted tailoring of the application-specific coherence hardware. The challenge of sharing efficiently the unified memory by multiple hardware threads offers another research possibility toward application-specific coherence protocols for codesigned reconfigurable applications.

Regarding the high-level synthesis (i.e., mapping to hardware from programs written in common high-level programming languages), unified memory abstraction for software and hardware significantly simplifies the automatic synthesis of high-level programming languages to hardware. Mapping to hardware constructs such as memory allocation, object creation, pointer-based accesses, and function calls, is rather difficult (or sometimes impossible) without the unified process concept. Our initial results allow the synthesis community to investigate unrestricted code migration from software to hardware, even dynamically at runtime (which is the biggest future challenge).

Apart from showing the viability of the unified process concept, the research work presented in this thesis calls forth immediate extensions and improvements. The analyses (qualitative and quantitative) presented in the thesis have shown the overheads of the applied scheme. One of the first tasks of future efforts shall be to tackle the overheads and try minimising their effects.

Although we do not address management of reconfigurable hardware and its reconfiguration in this thesis, we recommend the integration of the presented interface management with the resource management (addressed in the related work) to build a full-fledged OS extensions for codesigned reconfigurable applications. With appropriate extensions in binary file formats (such as *Executable and Linkable Format*—

ELF) and the OS support, an application loader could automatically reconfigure an FPGA resource preparing the user hardware at the application start-time. Further on, an FPGA-aware dynamic linker could reconfigure, if needed, the FPGA fabric at runtime. The challenge is to find efficient solutions for task partitioning and configuration scheduling, and to hide performance drawbacks incurred by FPGA configuration times.

The dynamic prefetching in the system layer responsible for the unified memory abstraction between user software and user hardware has demonstrated significant performance improvements. As the currently employed technique handles stream-based memory accesses, future runtime optimisations may address other memory access patterns (e.g., pointer chasing) with appropriate extensions in the system software (the VMW manager) and system hardware (the WMU). The dynamic nature of these techniques may allow the VMW manager to follow the performance and select at runtime the best prefetching policy for a given hardware accelerator. Dynamic optimisation techniques are not limited to prefetching: other techniques may include adapting the memory hierarchy for a particular application, in-advance hardware reconfiguration, and optimising the execution transfers from software to hardware and vice versa.

To enable and foster the widespread presence and use of reconfigurable computing, future work shall address integrating compilers and synthesisers into a single compilation tool. Starting from a program written in a high-level programming language, the tool shall try to exploit the available parallelism and generate HDL descriptions of hardware accelerators complying to the host processor ABIs. With the unified OS process supporting stack frames and calling sequences, treating any high-level language and generating its runtime support code (in both software and hardware) shall be feasible. Finally, having widely-accepted programming languages compiled by tools with no synthesis and partitioning restrictions would offer benefits of spatial computation in application-specific hardware to a large number of general-purpose computer users.

Appendix A

WMU Interface

THE WMU defines the hardware interface of virtual-memory-enabled hardware accelerators. It decouples the accelerator design from the host platform. The WMU is a platform-specific element ported once per reconfigurable SoC; different applications on the same platform reuse the same WMU. In this appendix, we explain the WMU interface toward hardware accelerators.

A.1 Hardware Interface

The WMU provides hardware support for translation of the virtual addresses generated by the accelerator. If possible, the WMU translates virtual addresses demanded by the coprocessor (similarly as the MMU does for the user application running on the main processor) to real addresses of the local memory region. Otherwise, the WMU generates an interrupt to request the OS handling, while the hardware accelerator is stalled.

To foster the accelerator reuse and portability across different platforms, we accept a common interface between the WMU and the accelerator. We standardise the interface for both platforms that we use. Should we use another platform in the future, the basic WMU interface would not change. Figure A.1 shows how a virtual-memory-enabled hardware accelerator is interconnected to the WMU. The standard interface consists of virtual address lines (`hwacc_vaddr`), data lines (`hwacc_din` and `hwacc_dout`), byte enable lines (`hwacc_be`) and control lines. The control signals between the accelerator and the WMU are the following: (1) `hwacc_start`, the accelerator start signal, issued by the WMU once a user initiates the execution transfer; (2) `hwacc_access`, the accelerator access signal, indicates that there is an access currently performed by the accelerator; (3) `hwacc_wr`, the accelerator write signal, indicates that the access is a write; (4) `hwacc_tlbhit`, the translation hit signal, indicates that an address translation is successful—in order to proceed with a memory access, the accelerator should first wait for this signal to appear; (5) `hwacc_fin`, the accelerator completion signal, indicates to the WMU that the accelerator has finished its operation. The WMU is connected to the host platform through platform-specific signals (e.g., the physical address lines of the local memory).

```
--
-- hardware accelerator ports complying to the WMU interface
--
entity hwacc_edge is
  generic (
    HWACC_DATA_WIDTH : positive := 64;
    HWACC_ADDR_WIDTH : positive := 32
  );
  port(
    hwacc_din :  in std_logic_vector (HWACC_DATA_WIDTH-1 downto 0);
    hwacc_dout :  out std_logic_vector (HWACC_DATA_WIDTH-1 downto 0);
    hwacc_addr :  out std_logic_vector (HWACC_ADDR_WIDTH-1 downto 0);
    hwacc_be :  out std_logic_vector (HWACC_DATA_WIDTH/8-1 downto 0);
    hwacc_access :  out std_logic;
    hwacc_xchange :  out std_logic;
    hwacc_wr :  out std_logic;
    hwacc_inv :  out std_logic;
    hwacc_fin :  out std_logic;
    hwacc_start :  in std_logic;
    hwacc_tlbhit :  in std_logic;
    hwacc_clk :  in std_logic;
    hwacc_reset :  in std_logic
  );
end entity hwacc_edge;
```

Figure A.1: The VHDL code of the WMU interface for hardware accelerators. Connecting a virtual-memory-enabled hardware accelerator to the WMU interface requires using the ports shown in the figure.

A.2 Parameter Exchange

The WMU contains a register file that the VMW manager uses for parameter exchange between user software and user hardware. The register file is of the limited size (current WMU implementations have eight 32-bit registers used for this purpose); if invoking the hardware accelerator needs more parameters, either the user stack or any other region in the main memory can be used for this purpose—this poses no problem since the hardware accelerator is capable of accessing the virtual memory address space. Having the register file in the WMU makes the parameter exchange faster—in most cases there is no need to access the memory.

Figure A.2 shows the timing diagram for the parameter exchange. When the user program launches the `sys_hwacc` system call, the VMW writes a start bit in the WMU control register, which, in turn, starts the execution of the accelerator by asserting the `hwacc_start` signal. This indicates that the accelerator shall start reading the input parameters by generating the exchange register file addresses. During this phase, the accelerator asserts the `hwacc_xchange` signal, meaning that the addresses on the address lines are intended for the register file. The usage of the registers is the following: (1) the register at the address `0x0000` holds the number of accelerator parameters (`nparam`); (2) the register at the address `0x0001` contains control flags (`flags`) which are optional; (3) the rest of the registers (the addresses from `0x0002` to `0x0007`) store the parameters used to invoke the accelerator (`param[0-2]` in this particular case). In the case of more than six parameters, the register at the address `0x0007` is a pointer to the memory region (or the stack) where the accelerator can find the remaining parameters. Appendix B shows the usage of the parameter exchange structure in software.

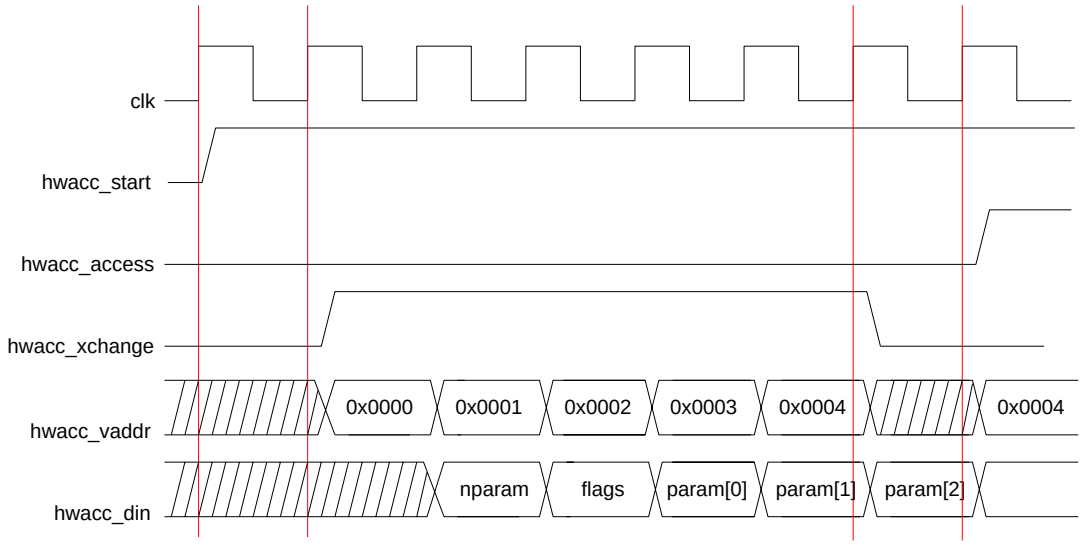


Figure A.2: Timing diagram showing the accelerator start up. The WMU sets the `hwacc_start` signal which starts the accelerator. The accelerator generates accesses in the exchange mode (the `hwacc_xchange` is asserted), reading parameters from the exchange register file. Afterward, the accelerator starts normal operation, accessing the virtual memory (the `hwacc_access` signal is one).

Once the initialisation is over, the accelerator starts normal operation and accesses the virtual memory. During the memory accesses, the accelerator sets the `hwacc_access` signal to one, puts the valid address, and waits for the reception of the `hwacc_tlbhit` signal. Receiving the signal means that the translations is over and the requested data is available at the data input lines. Figure C.3 shows an example of such access.

Figure A.3 shows the timing diagram for the accelerator completion. After finishing the computation, the accelerator returns back to software. If the accelerator produced a return value (we use the C language semantics), it returns it back to software through the exchange register file of the WMU. Before signalising the completion (by asserting the signals `hwacc_fin` and `hwacc_access`), the accelerator writes the return value and the return code to the WMU. The return code is necessary for the VMW manager to distinguish the situation when a normal returns appears from the hardware callback to software. If the value stored by the accelerator at the address `0x0001` of the register file differs from the return code, the VMW manager uses it for invoking the appropriate software function. The rest of the registers (from address `0x0002` to address `0x0007`) are at the accelerator disposal for passing parameters to software.

The completion of the accelerator operation raises an interrupt handled by the VMW manager. Acknowledging the interrupt resets the interrupt bit of the WMU status registers. Then, the VMW manager clears the start bit in the WMU control register, which, in turn, puts the `hwacc_start` signal to zero. The accelerator uses the change of this signal (if there is no callback to software pending) to go to its initial state. In this way, it is ready for future invocations.

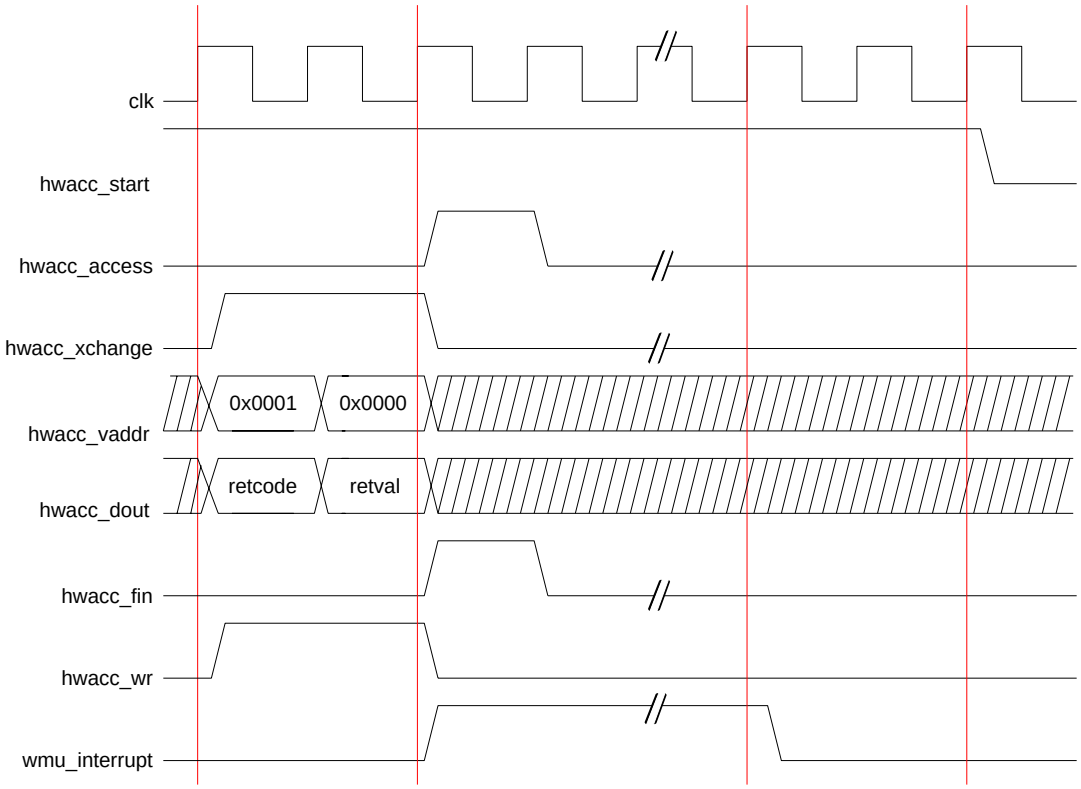


Figure A.3: Timing diagram showing the accelerator completion. The accelerator uses the `hwacc_xchange` and `hwacc_wr` signal to signalise writing to the exchange register. It writes the return indication (`retcode` and the return value `retval`, and signalises the completion to the WMU (through the `hwacc_access` and `hwacc_fin` signals). This raises an interrupt handled by the VMW manager.

A.3 Internal Structure

The VMW manager manages the local memory (similarly as the VMM does with the main memory), which is divided into pages to allow multiple virtual mappings to the user memory.

The WMU matches the upper part of the coprocessor address (most significant bits) to the patterns representing virtual page numbers stored in the translation table. It can support multiple operation modes, i.e., different page sizes and the number of pages. Possible operation modes for a WMU using the local memory of 16KB (12 bits needed for 32-bit word physical addressing) are shown in Table A.1. The optimal number of pages depends on the characteristics of coprocessor's memory access pattern.

How a 32-bit virtual address is translated to the corresponding physical address is shown in Figure A.4. The operation mode determines how many bits of the virtual address represent the virtual page number and how many of them represent the page offset. A simple shifter to the right selects how many bits are given to the WMU as a match pattern. The number of TLB entries in the WMU and the bitwidth of its contents correspond to the number of pages for the current operation

Number of pages	Page size	VirtPage bits	PhyPage bits	Offset bits
2	8KB	21	1	11
4	4KB	22	2	10
8	2KB	23	3	9
16	1KB	24	4	8
32	0.5KB	25	5	7

Table A.1: WMU operation modes. Different number of local memory pages are supported by the WMU.

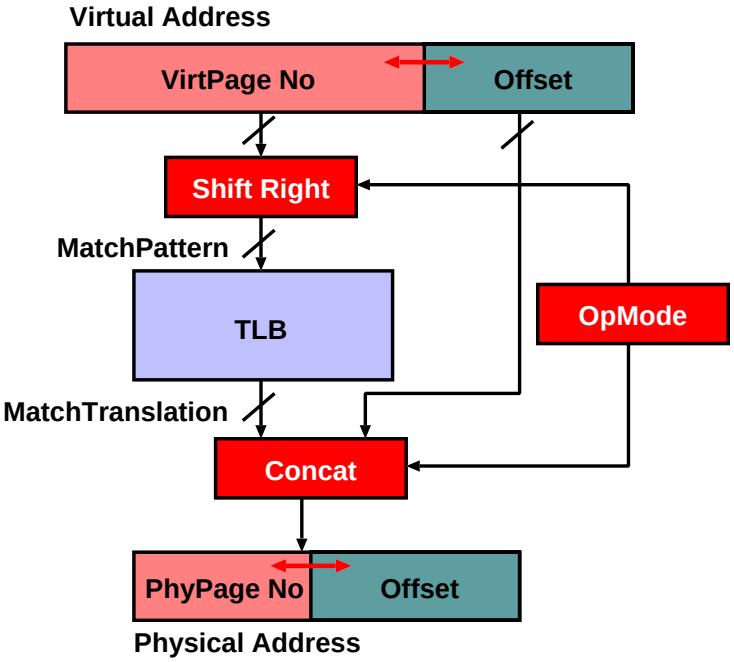


Figure A.4: WMU address translation and multiple operation modes. The WMU forms the appropriate physical address depending on the current operation mode.

mode. At the WMU exit, a concatenator forms the correct physical address. The current operation mode determines how many of the match translation and offset bits are needed to form an address for the 4K words local memory. The OS controls the WMU and could adapt its operation mode to application requirements, even during application runtime.

Appendix B

VMW System Call

THE VMW manager is responsible for the translation process and memory transfers. Besides, it also defines the programming interface for using virtual-memory-enabled hardware accelerators. The interface consists of a system call that performs the execution transfer from user software to user hardware and passes the necessary parameters. It also provides some auxiliary functions that we use for the measurement and control purposes. In this appendix, we show the programming interface and give the details of its use, and we explain the invocation of hardware accelerators from software.

B.1 VMW Programming

We extend the host OS with a system call `sys_hwacc` that is responsible for the interface between use software and user hardware. In our particular implementation, the VMW manager is an OS module loadable at runtime. Once the it is loaded and linked to the kernel, the module hooks the `sys_hwacc` call into the dispatch table for system services. Further on, the programmer can use it for configuring and accessing hardware accelerators.

Before invoking a hardware accelerator, the programmer has to make sure that the FPGA contains the accelerator. In our work we do not address possibilities of dynamic reconfiguration. We just provide means for configuring the FPGA before the particular codesigned application is run. In the case of the Altera device (discussed in Appendix C), we create a special device with the appropriate support in the OS. Copying a bitstream file (the file containing the configuration bits for the programmable logic—it is the outcome of the Altera design tools) to the special device reconfigures the chip and puts the required hardware accelerator in place. This enabled further references to the accelerator through the `sys_hwacc` call. Although the Xilinx device (also explained in Appendix C) should support partial runtime reconfiguration, the design flow could not produce designs for reconfiguration (this flaw in the design tools has been corrected in the latest version of the tools). Thus, in the Xilinx case, we had to configure the system (with all required hardware accelerators, WMUs, and local memories) statically, at start up time, before launching the operating system.

Figure B.1 shows an example of invoking a hardware accelerator through the `sys_hwacc` system call. The function contains preparatory steps such as setting up the parameter passing structure. The programmer sets the number of parameters (`p.u.nparam`), than the parameters received by invoking the library function (`p.param[0-3]`). The flags field of the structure (`p.v.flags`) is set up to define the operation mode of the WMU (`VMW_MODE_P8` macro specifies the operation mode

```
/* include necessary definitions */
#include<vmw.h>

/* callback functions */
int cbackfun1();
int cbackfun2();

/* library function for calling a hardware accelerator */
int hwacc_call(int *A, int *B, int *C, int size) {
  /* the parameter passing structure */
  struct_param p;
  /* the structure for execution reports */
  struct_report r;
  /* the array for function pointers */
  unsigned int funptr[2];

  /* set up the parameters */
  p.u.nparam = 4; /* the number of parameters */
  p.param[0] = A; /* input vector pointer */
  p.param[1] = B; /* input vector pointer */
  p.param[2] = C; /* output vector pointer */
  p.param[3] = size; /* data size */

  /* setting up the operation mode and announcing callbacks */
  p.v.flags = VMW_MODE_P8 | VMW_CBACK_F2;

  /* preparing callbacks */
  funptr[0] = (unsigned int) cbackfun1;
  funptr[1] = (unsigned int) cbackfun2;
  p.param[5] = funptr;

  /* invoking hardware accelerator */
  if (sys_hwacc(HWACC_ID, VMW_START, &p) != 0) {
    printf("Cannot start.  Exiting.");
    exit(0);
  }
  /* collecting execution reports */
  if (sys_hwacc(HWACC_ID, VMW_REPORT, &r) != 0) {
    printf("Warning.  Cannot get the report.");
  }
  /* print execution reports, etc.  */
  ...
}
```

Figure B.1: Invoking a hardware accelerator through the system call interface. The parameter passing structure contains the pointers and sizes of the data to be processed. Function pointers for callbacks to software are also passed to the VMW. The VMW manager collects execution time and other statistics and returns it through the report structure.

with the local memory divided into eight pages). Another parameter to the flags (`VMW_CBACK_F2`) indicates that the hardware accelerator may callback up to two software function. In the presence of callbacks, the sixth parameter (`p.param[5]`) is used to pass the array of callback function pointers. After finishing the preparatory

steps, the `sys_hwacc` call invokes the VMW manager that writes the parameters to the WMU exchange registers (explained in more detail in Section B.2) of the selected hardware accelerator and starts the execution of the accelerator by setting the appropriate bit in the control register of the WMU. When the accelerator finishes its execution, the `sys_hwacc` returns and the programmer can collect the report data through another call to the `sys_hwacc`. All the steps in the library function are rather straightforward and can be easily generated in automated fashion.

B.2 Parameter Exchange

The VMW manager receives the parameter exchange structure from the user space from the `sys_hwacc` system call. Figure B.2 shows the C language definition of the structure. The structure corresponds to the physical layout of the exchange register in the WMU(there are eight 32-bit registers in the exchange register file). The manager reads the `nparam` field of the union `u` and then copies available parameters from the `param[]` array to the appropriate registers in the exchange register file of the WMU. By reading the `flags` field of the union `v`, the manager determines the required operation mode of the WMU (the default one is the local memory divided in eight pages) and writes it to the control register of the WMU. If the `flags` field shows the presence of callbacks, the VMW manager gets the appropriate number of function pointers (jump addresses) from the user space. After passing all necessary parameters, the VMW manager launches the execution of the accelerator, by setting the start bit in the WMU control register.

```
typedef struct {
  union {
    unsigned int nparam; /* number of parameters */
    unsigned int cpuretval; /* CPU return value */
  } u;
  union {
    unsigned int flags; /* flags and hints */
    unsigned int cbackid; /* call back function id */
    unsigned int cpretval; /* CP return value */
  } v;
  unsigned int param[6]; /* parameters array */
} struct_param;
```

Figure B.2: The parameter exchange structure. Apart from the invocation parameters, the structure contains control information exchanged in both direction. The structure corresponds to the registers of the WMU exchange register file.

When the VMW manager receives an interrupt from the WMU indicating the execution transfer back to user software (recall Figure 4.4 in Section 4.3), it first examines the WMU exchange register corresponding to the `cbackid` field of the union `v`. If it finds there the `retcode` code (which we mentioned in Section+A.2), the accelerator has terminated its execution and the VMW manager performs the execution transfer back to the `sys_hwacc` caller. The return value from the accelerator (`cpuretval` from the union `u`) is passed back to the caller. If it finds no return code, the value read from the register is treated as the identifier (`cbackid`) of the

callback request. The manager performs the execution transfer from user hardware to user software (as explained in Section 6.2), and passes possible parameters for invoking the callback software function. It determines the number of parameters by reading the WMU register corresponding to the `nparam` field of the `u` union and reads the parameters from the exchange register file (the registers corresponding to the `param[]` array of the exchange structure). Finally, the `cbackid` determines which function pointer the VMW manager shall use for setting up the jump address on the user stack (recall Figure 6.3). When the callback function finishes, the trampoline on the user stack returns back the execution to the VMW manager. The manager copies the return value received from the callback function to the WMU exchange register file and resumes the execution of the hardware accelerator. The accelerator, in turn, reads the return value through the parameter exchange protocol (discussed in Appendix A) and continues the execution until another callback or the final completion.

Appendix C

Reconfigurable SoC Platforms

WE have implemented the VMW system on two different reconfigurable SoCs coming from major FPGA vendors. We use an in-house-made board (called RokEPXA and shown in Figure C.1) based on an Altera Excalibur device [3], and the ML310 board based on a Xilinx Virtex-II Pro device [130]. We have designed the WMU in VHDL to be synthesised onto FPGA together with a hardware accelerator. We have developed the VMW manager as a Linux kernel module and ported it to both platforms. The VMW manager can support prefetching (as explained in Chapter 4). The following sections explain in more detail the two platforms.

C.1 Altera Excalibur Platform

Figure C.1 shows the RokEPXA board (designed by Gaudin [48]) that we used for our experiments. The main components of the board are the Altera Excalibur EPXA1 device [3], 64MB of SDRAM memory at 266MHz, 8MB of FLASH memory, Ethernet module, serial and USB ports, and various expansion and debugging ports. The development and cross compiling platform (on a standard workstation) is connected to the board over serial and network interfaces.

The Excalibur EPXA1 device we use is the smallest device in its family (e.g., EPXA4 and EPXA10 devices from the same family offer more programmable logic and larger on-chip memories). Figure C.2 shows a simplified scheme of the internal organisation of the EPXA1 device. It consists of a fixed ASIC part, called *Embedded Processor Stripe* in Altera's terminology, and of a programmable part called *PLD*—standing for *Programmable Logic Device*. The stripe contains the ARM922T processor with a cache and memory management unit. The processor runs at 133MHz. Other major components of the stripe are: (1) 32KB of single-port on-chip SRAM memory; (2) 16KB of dual-port on-chip SRAM memory also accessible by the programmable logic; (3) SDRAM controller; (4) JTAG interface for debugging. Internal on-chip peripherals are connected with the CPU over two AMBA buses for high-speed peripherals (AHB1 and AHB2 [8]). The main CPU can configure the PLD at runtime through a PLD configuration controller. In contrast to Xilinx Virtex-II Pro devices, partial configuration of the PLD device is not possible—the whole device has to be configured at once. However, most of the

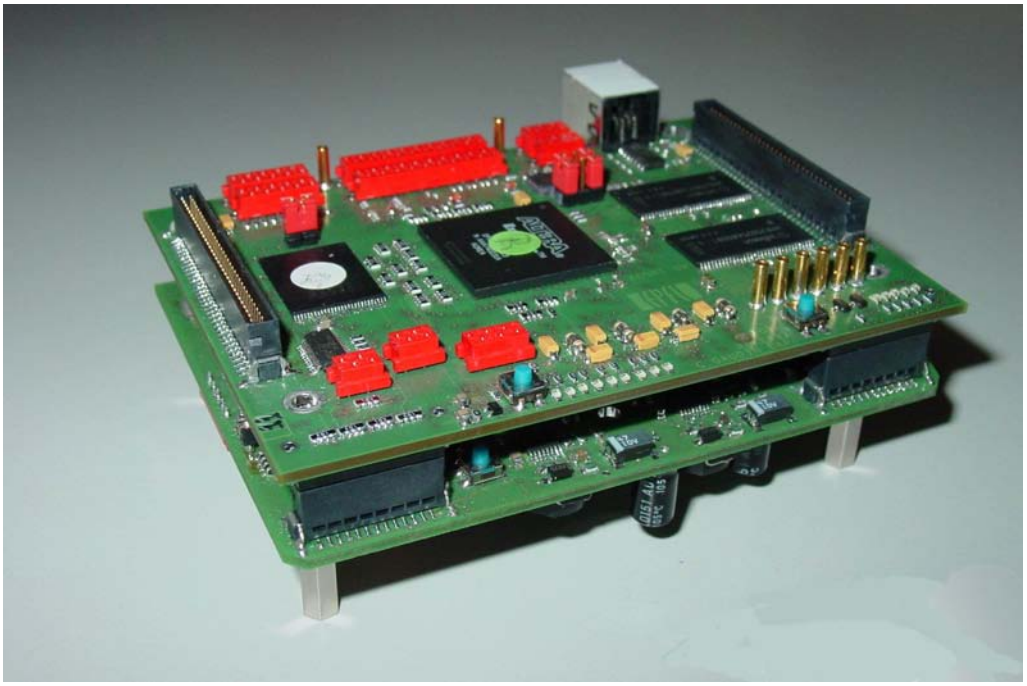


Figure C.1: RokEPXA board. The board is based on Excalibur reconfigurable SoC containing ARM processor and reconfigurable logic.

peripherals necessary for the normal system functioning are already present in the embedded processor stripe (which is not the case in Xilinx devices), thus, albeit the limitation, the reconfiguration process is made simpler.

We have designed and synthesised the WMU within the PLD device of the EPXA1 chip. Figure C.2 shows the WMU integration within the EPXA1. Since the dual-port on-chip SRAM is accessible by both the stripe and the PLD, we have chosen it for the local memory of the virtual-memory-enabled hardware accelerator. The VMW manager running on the ARM processor transfers the pages between the main memory and the dual-port memory using the AMBA bus. Hardware accelerators access the dual-port memory through the WMU using the WMU-interface protocol (shown in Appendix A). Depending on the WMU operation mode, the memory is logically organised in 2–32 pages with respective page sizes 8–0.5KB (the total size is therefore 16KB). The WMU part of the PLD design is fixed, designing a new hardware accelerators assumes just changing the HW ACC block in the design. The WMU is connected to the rest of the platform over the slave AMBA interface. Through this interface the VMW manager accesses the WMU and manages the translation.

Because of the limitations of the FPGA technology, the translation is performed in multiple cycles: if we assume no translation faults, four cycles are needed from the moment when the accelerator generates an access to the moment when the data is read or written (Figure C.3). The performance drop caused by multiple translation cycles can be overcome by pipelining. Although we had to implement the WMU in FPGA for these experiments, WMUs should, in principle, be implemented as ASIC cores, in the same way as MMUs are.

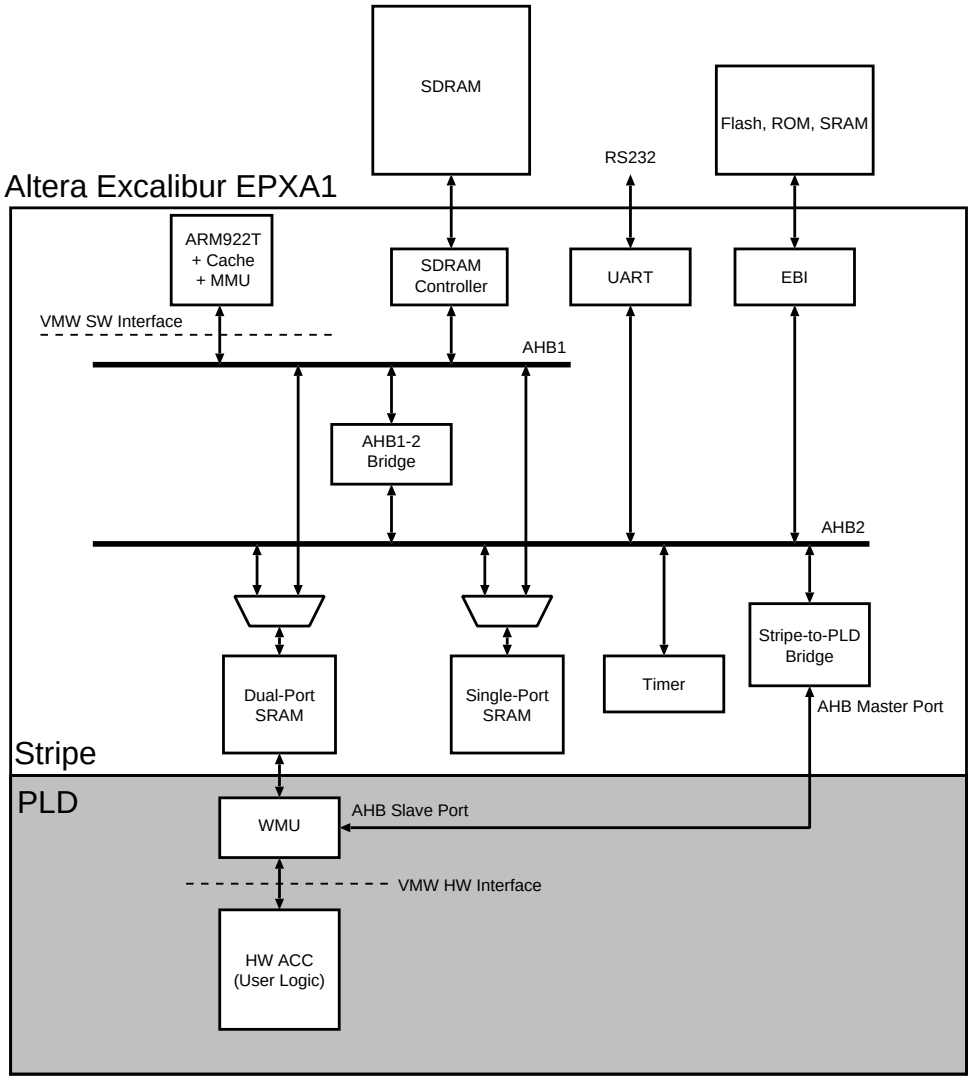


Figure C.2: WMU integration within Altera Excalibur EPXA1 device. The ASIC part of the device (Stripe) contains the ARM processor, on-chip memories, and peripherals. The reconfigurable part of the device (PLD) is used for the WMU and the application-specific hardware accelerator. A 16KB dual-port memory accessible by Stripe and PLD is used for the local memory of the accelerator.

The board is running GNU/Linux OS read from the FLASH memory at boot time. It uses the 64MB of the main memory for the working memory and the root file system. We originally developed the VMW manager for the Altera platform; however, it is easily portable to other platforms by adapting few configuration files. The manager is a loadable kernel module that can be added to the kernel at run time. The Linux kernel version the VMW relies on is 2.6 [75]. For reconfiguring the PLD we use a special character device supported by the module. In this way, one can configure the PLD by just issuing the following shell command: "cp *bitstreamfile* /dev/pld". Similarly, a program can do the same by invoking the appropriate system calls.

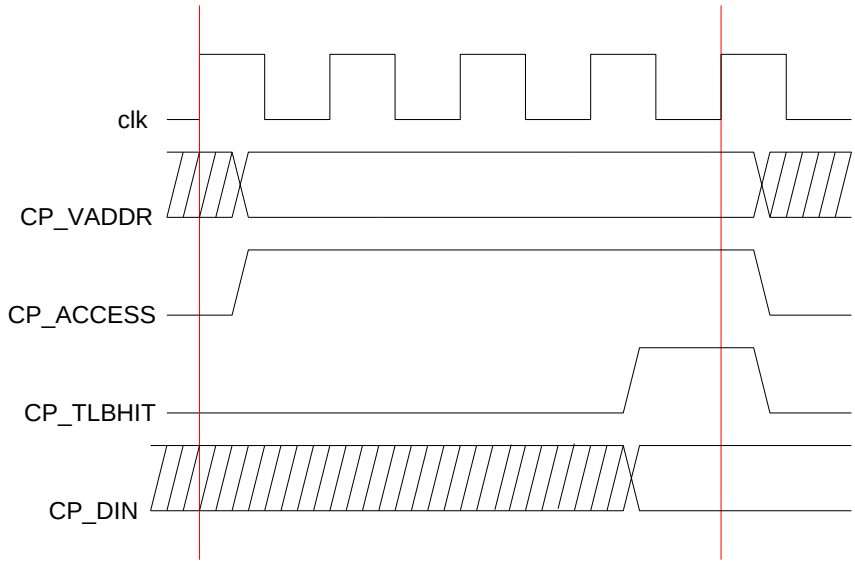


Figure C.3: The accelerator read access. The accelerator initiates the data transfer and waits for the confirmation signal. The Data is ready on the fourth rising edge of the clock.

C.2 Xilinx Virtex-II Pro Platform

The ML310 board [130] from Xilinx that we also used for our experiments is an advanced evaluation board containing the Xilinx Virtex-II Pro device XC2VP30. The board is equipped with 256MB of DDR RAM, 512 MB of CompactFlash memory, four PCI expansion slots, RS232 and USB serial interfaces, two IDE connectors, JTAG debugging interface, etc. Figure C.4 shows the ML310 evaluation board.

The XC2VP30 device contains two PowerPC 405 processors surrounded by FPGA logic (30000 logic cells). Except the two CPUs, the device has no other ASIC cores, meaning that the designer has to implement system peripheral devices (e.g., DDR RAM controller, UART, interrupt controller) in FPGA. The PowerPC processors can run at frequencies up to 300MHz. On-chip processor buses (PLB, OPB, and OCM [59]) and DDR RAM memory can run at the frequency of 100MHz.

Figure C.5 shows the architecture used for our experiments on the ML310 Xilinx evaluation board—in our experiments we use only one of the two PowerPC processors available. The CPU is connected to the *Processor Local Bus* (PLB). The external RAM controller is also connected to the same bus. Slower peripherals, such as UART and interrupt controller, are connected to the *On-chip Peripheral Bus* (OPB). We have connected the WMU to the same bus. The CPU controls the slower peripherals, such as UART, WMU and interrupt controller, through the PLB2OPB bridge. We use multiple on-chip memory blocks (BRAMs) to form the local memory of hardware accelerators (VMW Memory in Figure C.5). A BRAM block is a dual-port memory of configurable size. To simplify the design, we have originally connected the local memory to the OPB bus. This, however, has given unsatisfactory performance results. Our next step was to connect the local memory to the PLB bus, closer to the CPU and the memory. Nevertheless, this has neither

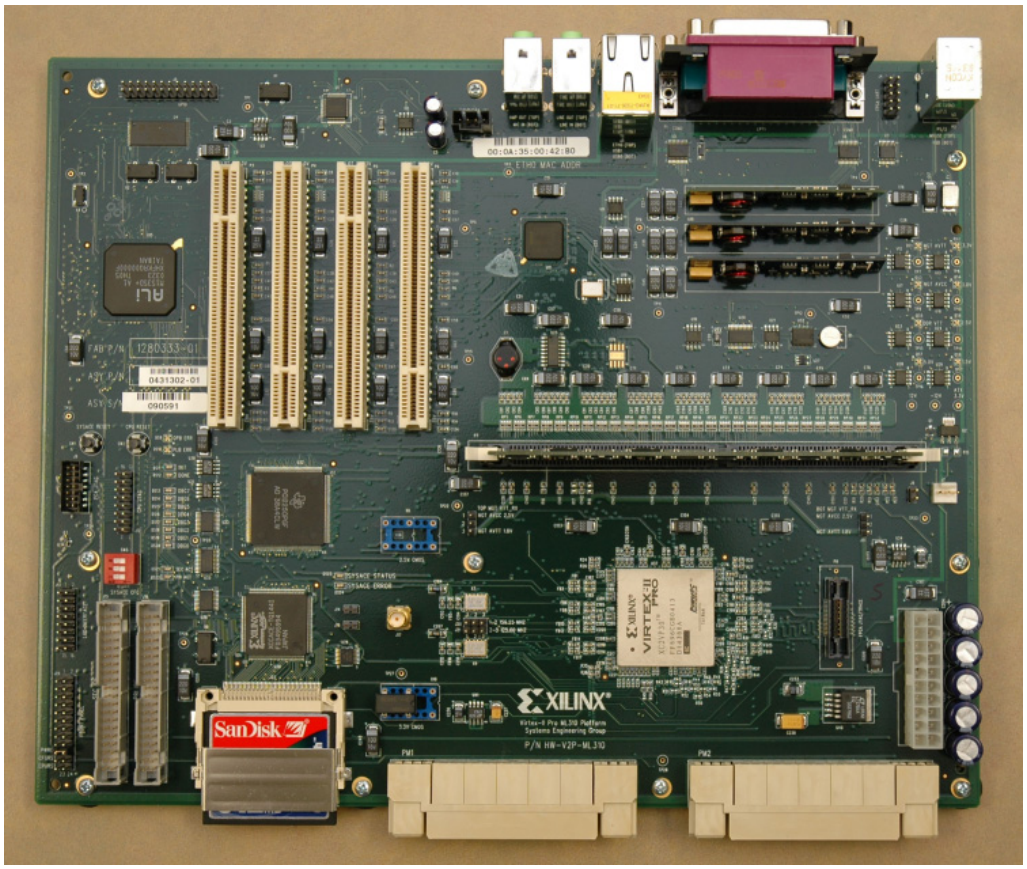


Figure C.4: ML310 board. The board is based on Virtex-II Pro reconfigurable SoC containing two PowerPC processors surrounded by FPGA logic.

improved performance. Unexpectedly, albeit the difference in running frequencies, the transfer times of the Altera Excalibur device were shorter than the corresponding Virtex-II Pro times. We have investigated the corresponding low-level kernel routines used by the VMW manager in both cases (ARM and PowerPC implementations of the Linux kernel). We found both implementations were assembler-based and highly optimised. Only when we excluded the PowerPC from the page transfer path and employed a DMA on the local memory side (as Figure C.5 shows), we have managed to get satisfactory performance results (Section 5.3.5 shows difference in copy times between the Altera and the Xilinx platforms). In the system configuration used for the experiments, the VMW manager just initiates data transfers from the main memory to the VMW memory (and vice versa); then, it is on the DMA to perform the task.

There is another peculiarity regarding our WMU implementation on Xilinx Virtex devices. We use *LookUp Tables* (LUTs) configured as shift registers (as recommended in a Xilinx Application Note—XAPP201 [130]) to implement the *Content Addressable Memory* (CAM) of the TLB. The implementation results in a CAM spending one cycle per pattern match. However, in the management phase, when a pattern is to be written in the CAM, one cycle per pattern bit is necessary. For example, writing a 24 bit pattern requires 24 cycles. As the management access ap-

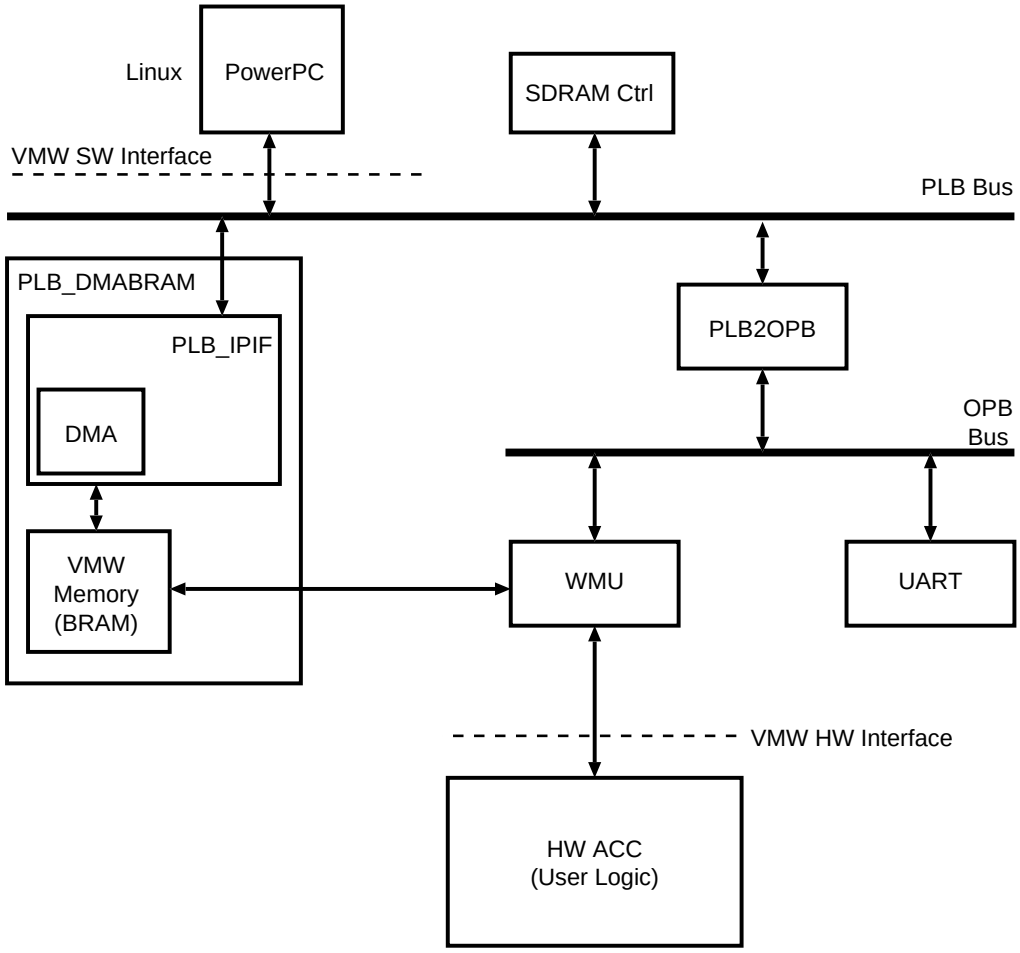


Figure C.5: WMU integration within Xilinx Virtex-II Pro XC2VP30 device. The PowerPC core is connected to the rest of the system through the PLB bus. The WMU is managed by the OS through the OPB bus. The virtual-memory-enabled hardware accelerators access the local memory built out of BRAM blocks. A DMA controller connected to the PLB bus performs memory transfers between the local memory and the main memory. The OS initiates the transfers, after receiving the WMU requests.

pears once per page fault, the performance drawback of did not show to be critical.

Since the reconfigurable SoC device of the ML310 board is significantly larger than in the RokEPXA case, we have extended the original VMW manager to support multiple, multithreaded hardware accelerators. Figure C.6 shows the system architecture for multithreading. One WMU per hardware accelerator is present. Having large amounts of reconfigurable logic available, we could afford sizes of the local memory from 16–64KB. In such a system, we have also envisioned the use of the partial reconfiguration interface (through the HWICAP peripheral connected to the OPB bus). Unfortunately, the immaturity of the design tools did not allow us to accomplish the partial reconfiguration; it is the task of the future efforts, especially now when the design tools have evolved.

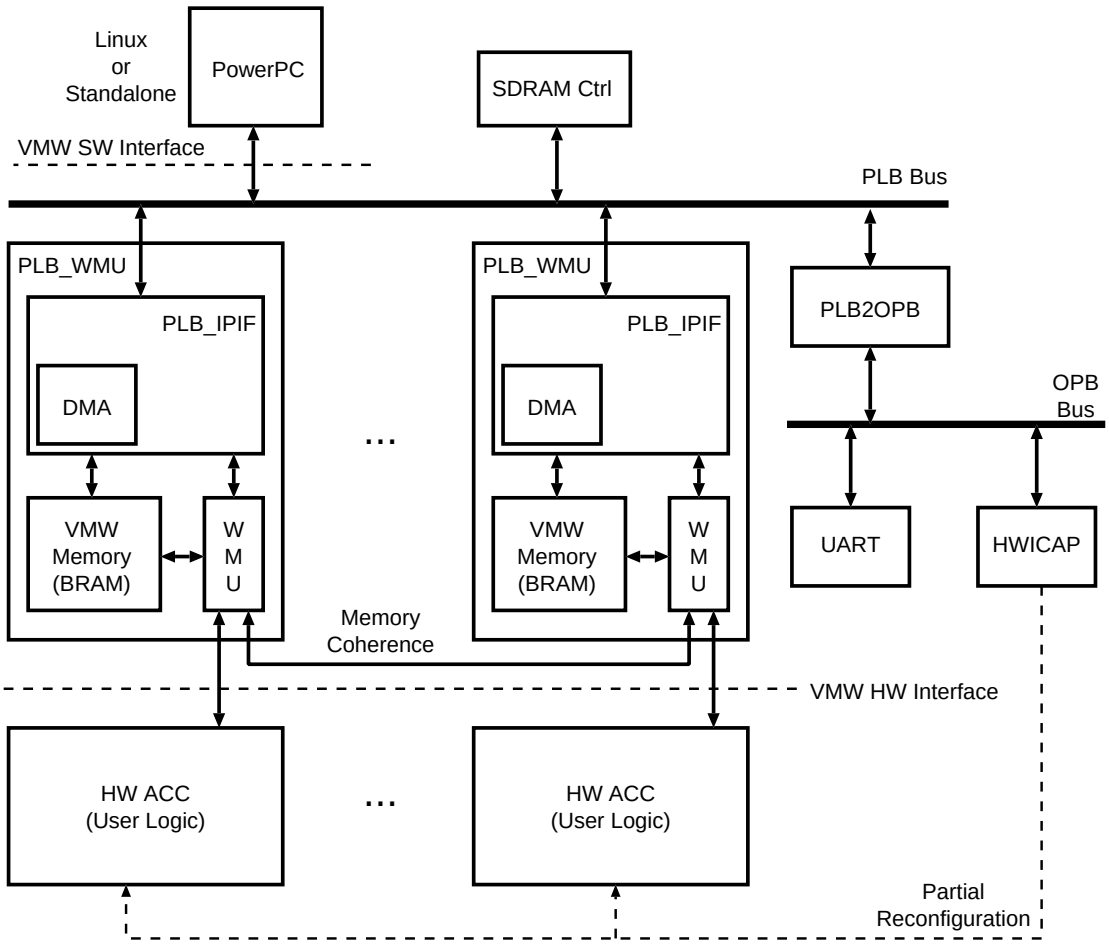


Figure C.6: Xilinx Virtex-II Pro XC2VP30 platform for multithreading. Multiple WMUs connect user hardware accelerators to the rest of the system. The user reconfigures the accelerators through the HWICAP peripheral.

Appendix D

Applications

IN this appendix, we show design details of a couple of hardware accelerators that we used in our experiments. We show the design of the IDEA cryptography accelerator and the design of the contrast enhancement engine for image processing. In both cases, we have taken already existing accelerators designed in synthesisable VHDL [17, 111, 124, 32]) and adapted them for our virtual memory interface. The original IDEA accelerator encrypts the data stored in local memory, while the original contrast engine directly accesses the main memory. In this way, the original accelerators represent two typical architectures (accessing local memory and main memory) described in Section 2.2. The experiments we performed with the virtual-memory-enabled accelerators have shown in practice the advantages of using our virtualisation layer. It simplifies the hardware interfacing and software programming, and allows porting applications from one platform to another without any change in the application code.

D.1 IDEA Accelerator Design

We have used the reference IDEA code [82] design together with a previous design in synthesisable VHDL [17] as the basis for designing the virtual-memory-enabled IDEA accelerator. The IDEA algorithm consists of eight rounds of the core transformation followed by the output transformation. When designing the accelerator, the eight rounds can be "unrolled" a certain number of times, depending on the available reconfigurable logic resources. The computation of a round contains four multiplications, four additions, and several XOR operations. The output transformation consists of two multiplications and two additions. The algorithm exhibits parallelism exploitable by hardware implementation.

Figure D.1 shows the design blocks of the original IDEA accelerator without VMW (a typical accelerator accessing local memory). There are four principal design blocks of the accelerator. 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. IDEA Core and Memory CTRL are synchronised using the stall signal: the core is stalled until Memory CTRL is ready to read/write the data.

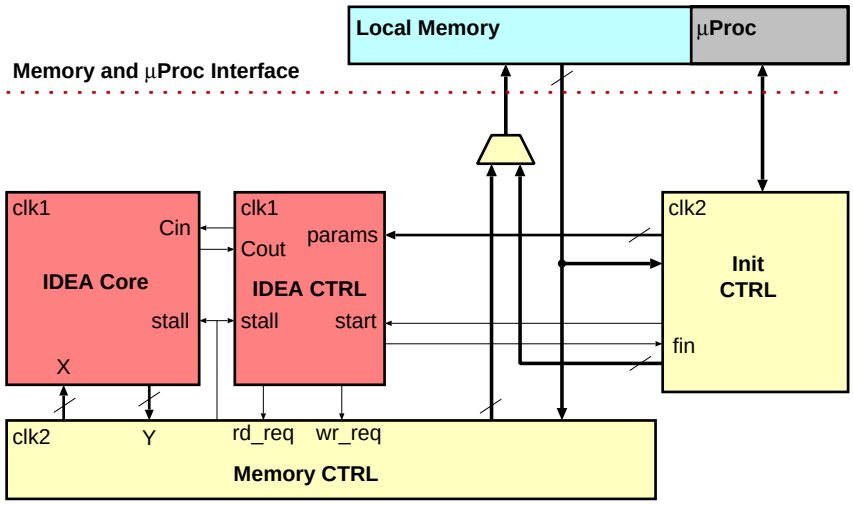


Figure D.1: Typical IDEA accelerator accessing local memory. The accelerator is interfaced directly to local memory. It generates physical memory addresses to access the data.

The IDEA core implements the eight computation rounds of the algorithm and the output transformation. The core performs the computation rounds iteratively—the rounds are not "unrolled" because of the limited FPGA space (we have originally targeted the Altera EPXA1 device, which is rather small in terms of available logic cells). The original pipeline depth of the IDEA round implementation [17] is five stages. The output transformation has only one stage. To obtain a smaller design, we have changed the available design of the IDEA core [121]. 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).

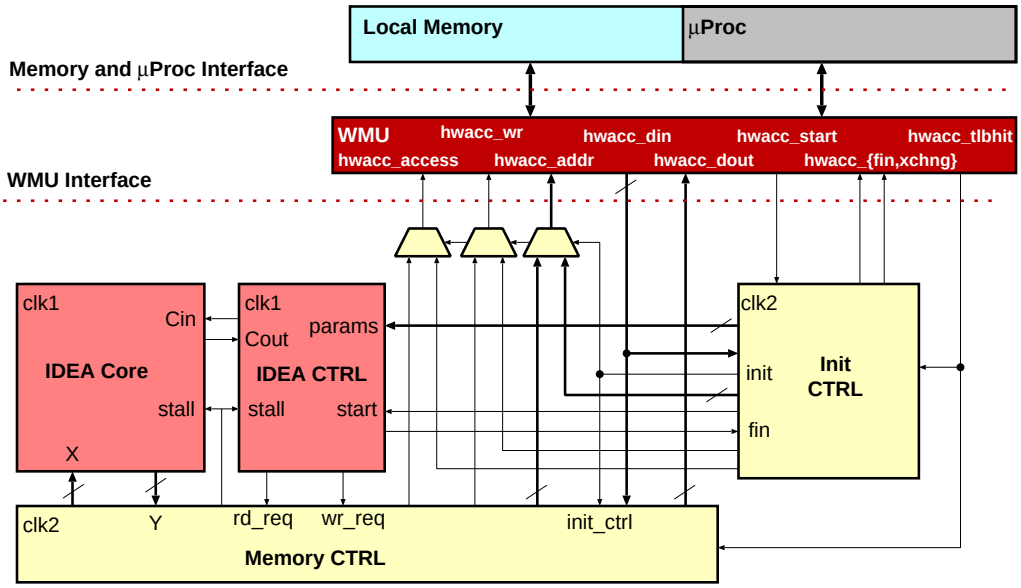


Figure D.2: Virtual-memory-enabled IDEA accelerator. The accelerator is interfaced through a standardised WMU interface and generates no physical addresses.

Figure D.2 shows the virtual-memory-enabled version of the IDEA accelerator. It consists of the same four design blocks as in the typical accelerator case, but with the memory access unit and the initialisation unit changed to comply with the WMU interface. 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 *virtual addresses* and not the physical addresses of the local memory. To avoid performance penalties because of 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). Synchronisation is again achieved using the stall signal (the core blocks until the memory request is fulfilled; afterward, it can continue the execution).

Once the WMU starts the accelerator with `hwacc_START`, Init CTRL takes control. It reads initialisation parameters through the WMU and passes them to the IDEA CTRL block (`params`). After passing all parameters, it gives the interface access to the memory controller (the `init` signal that controls the multiplexers allows Memory CTRL access to the WMU interface) and starts 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 (`hwacc_access`, `hwacc_wr`, `hwacc_vaddr`, `hwacc_dout`), waits for the hit acknowledgment arriving from the TLB. (`hwacc_tlbhit`), and eventually reads the input data lines (`hwacc_din`). When the computation is finished, IDEA CTRL passes the control back to INIT CTRL (`fin`), which informs the WMU about the successful completion, writes the return code and return value (recall Appendix A) to the exchange register file, and finishes the accelerator computation (`hwacc_fin`).

After synthesis for the Altera Excalibur device and the Xilinx Virtex-II Pro device, the complex IDEA accelerator core runs on lower frequency (6MHz—Altera, 25MHz—Xilinx) than the WMU and the IDEA memory subsystem (24MHz—Altera, 100MHz—Xilinx). For improving the processing throughput, the core of the IDEA accelerator is pipelined. However, due to the limited FPGA resources of the EPXA1 device, only three 64-bit blocks can be encrypted at a time. With larger FPGAs (such as EPXA10 from Altera or XC2VP30 from Xilinx) additional parallelism could be exploited. Nevertheless, we decide to use the single IDEA accelerator design that can fit both target device.

D.2 Contrast Engine Design

Starting from image-processing engines (developed by Stechele et al. [111, 124, 32]) that access the system main memory, we have built virtual-memory-enabled accelerators for the contrast enhancement and edge detection applications. The main intention of the original designers was to build a reconfigurable system for driver assistance applications. In this context, the system uses different hardware accelerators for processing camera-captured frames in different driving conditions [111].

In this section, we show the original design of the contrast enhancement engine and of its virtual-memory-enabled version.

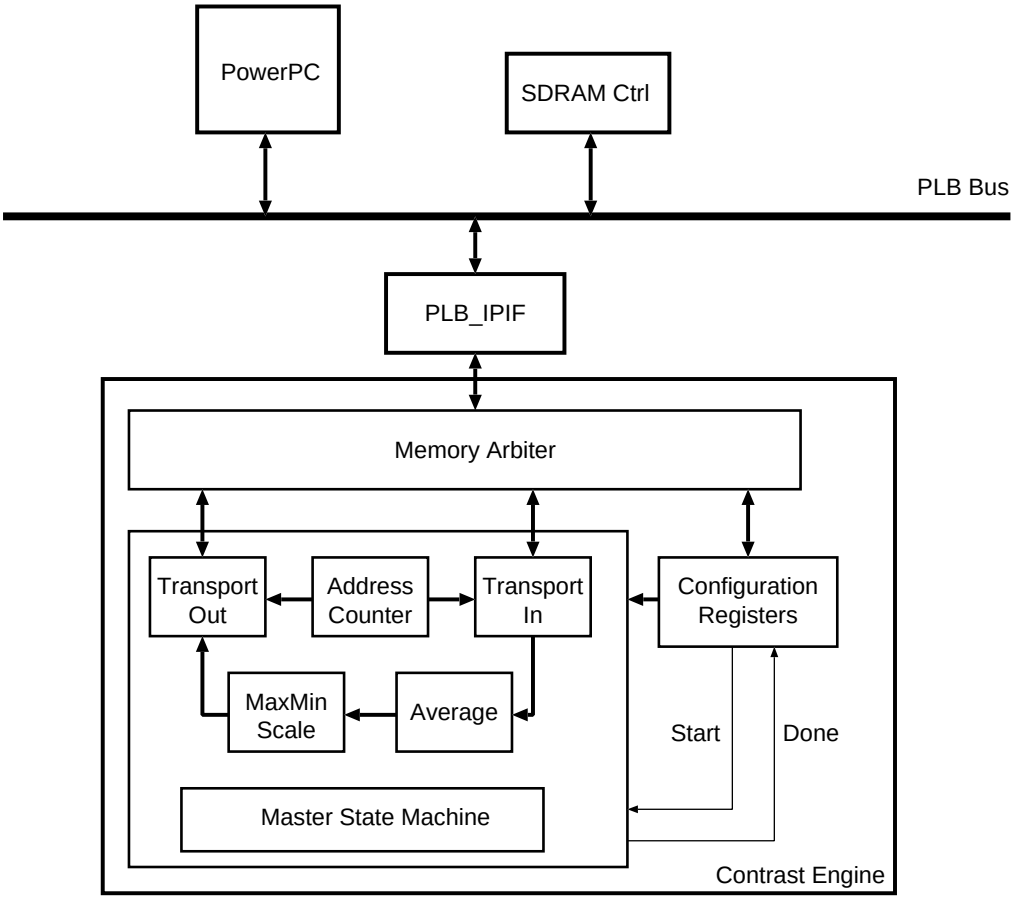


Figure D.3: Typical contrast enhancement accelerator accessing main memory. The accelerator is interfaced to the system bus and generates physical addresses of the main memory.

Figure D.3 shows the design of the contrast engine accessing the main memory. The accelerator is connected directly to the PLB (*Processor Local Bus*) bus through an interface wrapper (PLB_IPIF block generated by Xilinx design tools). The memory arbiter provides means for the host processor to access the configuration registers, and for the accelerator to access the memory via the system bus. The address counter generates addresses of the image processing windows. The task of the transport blocks (Transport In and Transport Out in the figure) is to initiate memory read and write accesses. Since the PLB bus width for data is 64 bits, each bus transaction (the accelerator uses single transfers only and it does not implement bursts) reads or writes 8 pixels at once. The averaging and scaling blocks perform the actual computation (i.e., contrast enhancement).

For starting the engine, the programmer has to write explicitly the engine parameters to the configuration registers. The registers are memory mapped to the user address space, as well as the input and output images (recall Figure 2.6 in Section 2.2). The input parameters define the size of the processing window and its

origin relative to the whole image (as Figure 5.2 illustrates). Once the computation has started, the engine first samples a subwindow (its size is 6 times smaller than the processing window) to determine the internal thresholds for the main processing. Then, it performs the contrast enhancement by reading four input windows and writing the output window in the result image. At the end, the accelerator generates an interrupt and waits for the software to acknowledge it. The programmer restarts the operation for processing another frame. The OS has to reserve the memory regions accessed by the accelerator at the system boot time. Processing another window would require copying to/from the reserved memory regions or reserving more memory from the OS. The programming is much simplified for the virtual-memory-enabled accelerators, where passing pointers to the dynamic memory containing the images is sufficient.

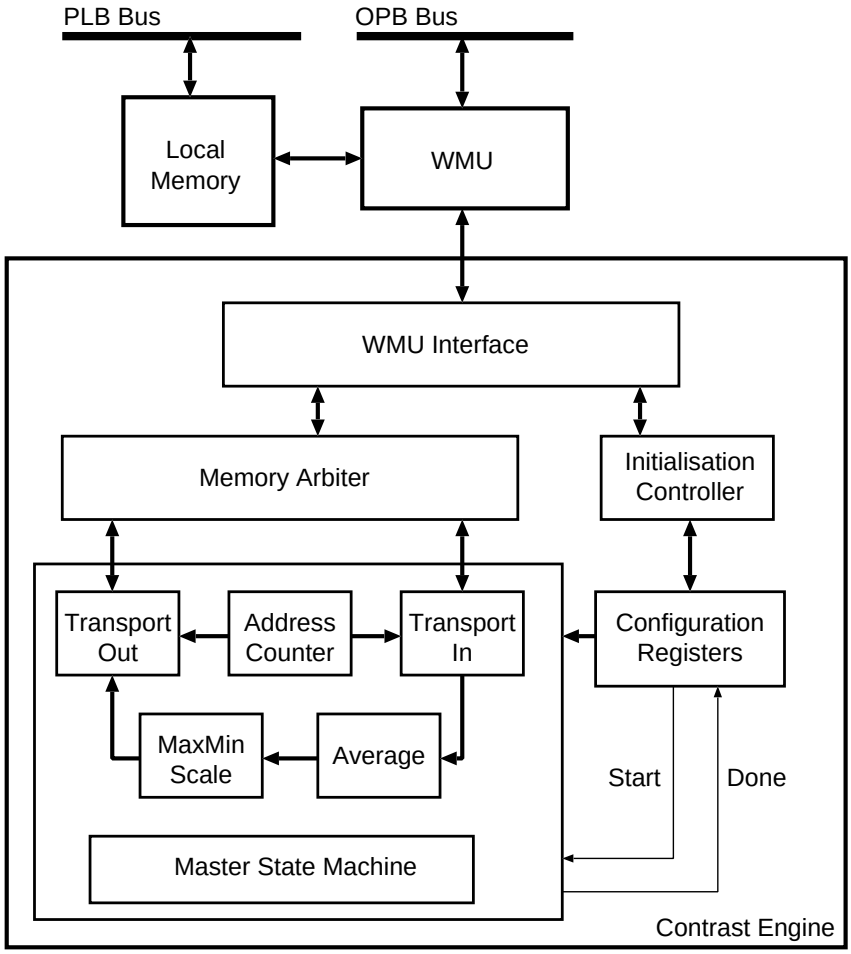


Figure D.4: Virtual-memory-enabled contrast enhancement accelerator. The accelerator is interfaced to the local memory through the WMU. It accesses virtual memory of the user address space.

Figure D.4 shows the virtual-memory-enabled version of the contrast enhancement engine. Instead of using the Xilinx interface, the accelerator connects to the system through the WMU interface toward the local memory. We have changed the original memory arbiter and added the initialisation controller. The rest of the

accelerator is practically unchanged. When the programmer starts the accelerator through the `sys_hwacc` system call, the engine reads the initialisation parameters through the WMU exchange registers and writes them to the configuration registers. Then, it launches the computation. The transport blocks now use virtual memory addresses to access the local memory through the WMU interface. There are no explicit transfers over the system bus initiated by the accelerator. It is completely hidden by the WMU interfaces, thus, platform-independent.

After synthesis of the accelerator (and the surrounding interfacing and system support blocks—discussed in Appendix C), the engine could run at the frequency of 100MHz on the Xilinx Virtex-II Pro device. The virtual-memory-enabled accelerators are connected to the local memory of 64KB. The spare reconfigurable logic of the device allows adding another accelerator to the system (with its own WMU interface and the local memory). We have used such system for running multithreaded experiments, with the contrast enhancement and edge detection accelerators running simultaneously in the reconfigurable hardware.

Appendix E

CPD Calculation

CHAPTER 5 introduces the CPD metric for heterogeneous code computers. In this appendix, we show how one can calculate the CPD value for simple control-flow (or data-flow) graphs representing the computation done by hardware accelerators. We also present an example of using the CPD metric to estimate the speedup achievable by a hardware accelerator in comparison with the software execution of a critical section.

E.1 CPD from Control-Flow Graphs

Figure E.1a shows a graph representing the IDEA hardware accelerator that we use in our experiments (the node weights for this particular example are related to the Altera-based reconfigurable platform). Coloured nodes represent memory operations and the white nodes stand for computation. Node weights define how many cycles it takes to complete the execution of the corresponding operation. The solid-line edges represent data dependencies, while the dashed-line edges represent the control dependencies. The sum of weights across a cycle built of control edges expresses the number of iterations for a control loop. In our discussion we show only simple graphs of the accelerators that we used in practice. We do not show graphs containing memory nodes that have computation nodes as its predecessors and successors at the same time—these graphs would require more complex computation of the CPD.

The memory accesses in this case are 32 bits wide (there is only one memory port available). For reading or writing an IDEA block the accelerator needs two memory accesses (IDEA blocks are 64 bits wide). The graph represents this fact with two read and two write nodes—the accelerator has to perform them in sequence. The computation consists of IDEA rounds (repeated eight times) and the output transformation. Executing the accelerator for a single input block requires $(4 + 4) + (12 \cdot 8 + 4) + (4 + 4) = 116$ cycles. The design of the accelerator permits preforming pipelined computation (in the main computation core—the IDEA round) on three IDEA blocks at the same time. Assuming the ideal memory port (i.e., each memory access is fulfilled in four cycles) and the pipelined execution of three IDEA blocks at a time, we find the CPD for the accelerator to be $CPD_{IDEA} = ((4 + 4) \cdot 3 + (12 \cdot 8 + 4) + (4 + 4) \cdot 3)/(3 \cdot 2) = 24.67$.

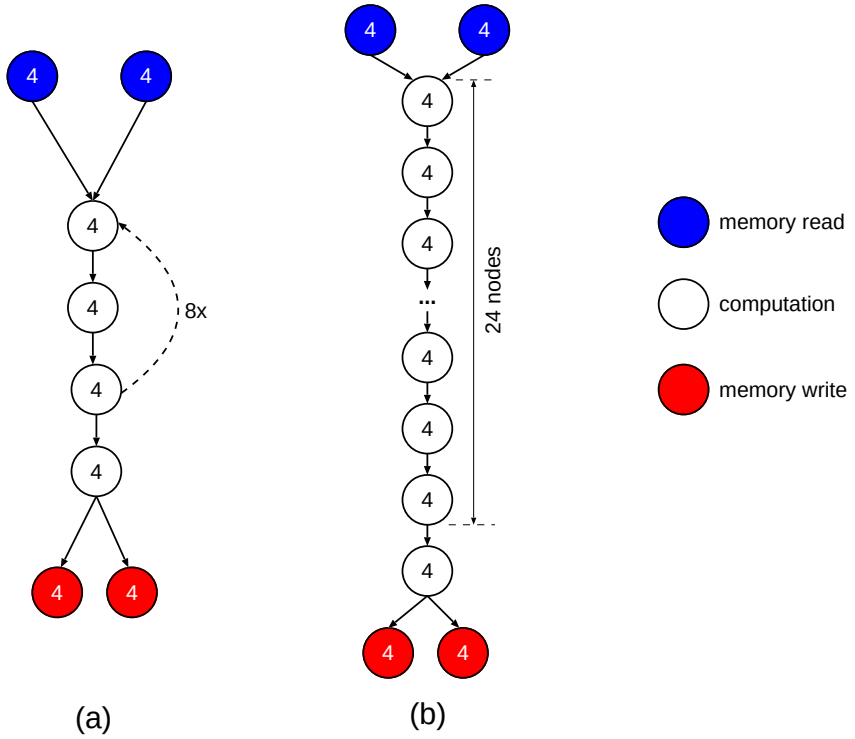


Figure E.1: Control-flow graph of the IDEA hardware accelerator for the original (a) and unrolled (b) version of the accelerator. Weights of the nodes represent the number of cycles. Memory reads and writes take multiple cycles because of the WMU translation overhead. The back-going edge represents a control loop that iterates eight times. Unrolling the computation eliminates the control loop.

If the designer had more FPGA area available, she could unroll the control loop and further exploit the available parallelism. Figure E.1b shows the unrolled version of the original iterative accelerator. Each node represent a pipeline stage of the computation. However, the single memory port is the limiting factor for this accelerator. Again, assuming that there are no memory stalls, the CPD of the modified accelerator is $CPD_{IDEA} = ((4+4) \cdot 25 + (4+4) \cdot 25)/(25 \cdot 2) = 8$. Having separate input and output ports in this case would provide the $CPD_{IDEA} = 4$.

For using the CPD values calculated as in the previous examples as correct (with a limited error), one has to ensure that the number of data to be processed (DC from Section 5.1) is much larger than the CPD value ($DC \gg CPD$). If this assumption does not hold, the effects of an unfilled pipeline influence the correctness of the result. For example, processing four IDEA blocks ($DC = 8$, in 32-bit words) by the accelerator with the pipeline capacity of three IDEA blocks (the one shown in Figure E.1a) takes in reality $((4 + 4) \cdot 3 + (12 \cdot 8 + 4) + (4 + 4) \cdot 3)) + ((4 + 4) + (12 \cdot 8 + 4) + (4 + 4)) = 148 + 116 = 264$ cycles, while using the CPD gives $DC \times CPD_{IDEA} = 8 \cdot 24.67 = 197.36$ cycles, i.e., about 34% error.

Contrast Enhancement Engine. Figure E.2 shows the data-flow graphs for the two phases of the contrast enhancement hardware accelerator (presented in Section D.2). The graphs contain no explicit control-flow dependencies. As dis-

cussed in Appendix D, the computation of contrast enhancement consists of two phases. The phases are not pipelined. Assuming one memory port (64-bits wide in the implementation on the Xilinx device) and no memory stalls, we find $CPD_1 = ((6+6+6+6)+12)/(4 \cdot 2) = 4.5$ and $CPD_2 = ((6+6+6+6)+15+6)/(4 \cdot 2) = 5.625$. In the case with each input and output port having its own memory port we have $CPD_1 = (6+12)/(4 \cdot 2) = 2.25$ and $CPD_2 = (6+15+6)/(4 \cdot 2) = 2.625$

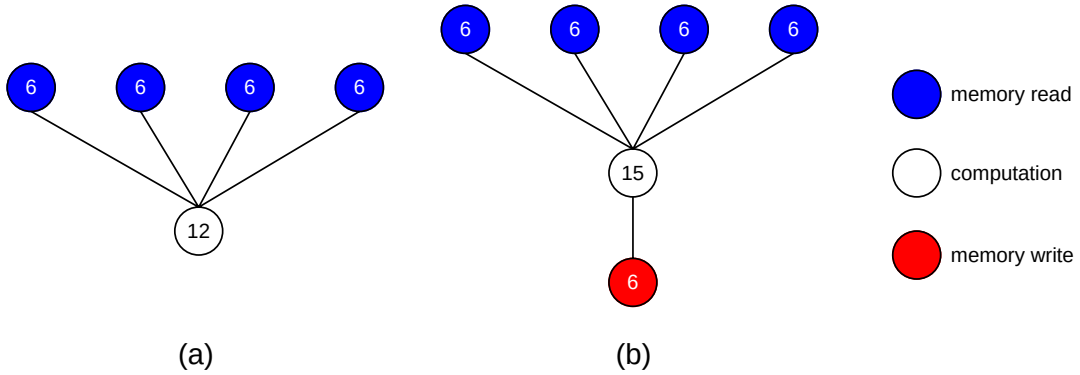


Figure E.2: In the first phase (a), the accelerator samples input subwindows and produces no output. In the second phase (b), the accelerator performs the contrast enhancement and produces the output window. Memory operations go through the WMU and take 6 cycles in this case (the WMU implementation on the Xilinx device).

E.2 Hardware Execution Time

The overall execution time of a hardware accelerator consists of the time spent in software activities to support the interfacing and of the time spent in hardware execution:

$$ET_{HW} = IC' \times CPI' \times T_{CPU} + CC_{HW} \times T_{HW}, \tag{E.1}$$

where CC_{HW} is the total number of cycles (cycle count of hardware clock periods T_{HW}—in the general case, T_{HW} is different from T_{CPU}) needed by the accelerator to perform its computation. We use IC' and CPI' in the above equation to emphasise their difference from the corresponding quantities of Equation 5.1 (Section 5.1).

While Equation 5.1 implicitly accounts (through CPI) for time spent in system hardware activities (such as branch prediction, instruction and data caching, multiple issue [44, 56]) to support the software-only execution, Equation E.1 explicitly shows (through $IC' \times CPI' \times T_{CPU}$) the time spent —recalling the discussion in Section 2.2—in burdensome software activities (such as partitioning data, scheduling memory transfers) to support the interfacing. On the other side, Equation E.1 implicitly accounts (through CC_{HW}) for the time spent in cumbersome hardware activities (such as managing buffers and burst accesses). In contrast to the system activities accounted in ET_{SW}, the execution support activities in ET_{HW} are delegated to the user (i.e., the software programmer and the hardware designer).

Section 5.1 proposes a metric that clearly separates execution-support and pure-execution activities of hardware accelerators.

The CPI parameter gives an insight regarding the CPU architecture and organisation, which is not the case with the CC_{HW} figure. We can notice that there is no instruction count for the hardware part of Equation E.1—there is only one instruction (i.e., the hardware accelerator itself), which does not change over the execution of the application.

We use the CPD metric to derive another way of expressing the overall execution time of the hardware accelerator[1]:

$$\begin{aligned} ET_{HW} &= HT + CT \\ &= DC \times CPD_{HW} \times T_{HW} + CC_{mem.stalls} \times T_{HW}, \end{aligned} \tag{E.2}$$

where CPD_{HW} is the CPD value for the hardware accelerator assuming the ideal memory hierarchy (i.e., the memory accesses introduce no stalls) and, $CC_{mem.stalls}$ is the number of cycles (cycle count) that the accelerator has to wait for the software or hardware action or, in other words[2]:

$$CC_{mem.stalls} = N_{miss} \times P_{miss}, \tag{E.3}$$

with N_{miss} being the number of times the accelerator has to wait for the transfers to the local memory, and P_{miss} being the cost or penalty in cycles of such event. We shall notice that P_{miss} represents, in fact, the cycles spent in software for copying, or in hardware for accessing the main memory and filling local buffers with burst accesses. We further develop Equation E.2 to obtain

$$\begin{aligned} ET_{HW} &= DC \times CPD_{HW} \times T_{HW} + N_{miss} \times P_{miss} \times T_{HW} \\ &= DC \times \left(CPD_{HW} + \frac{N_{miss}}{DC} \times P_{miss} \right) \times T_{HW} \\ &= DC \times (CPD_{HW} + R_{miss} \times P_{miss}) \times T_{HW} \\ &= DC \times (CPD_{HW} + CPD_{memory}) \times T_{HW}, \end{aligned} \tag{E.4}$$

where R_{miss} is the miss rate, and CPD_{memory} represents overhead incurred by the memory transfers from the main memory to the local memory and vice versa or by the direct accesses to the main memory (with and without bursts, as Section 2.2 discusses).

The form of Equation E.4 is convenient for the hardware accelerators where we cannot easily separate the CT and HT components, but we can measure or calculate R_{miss} and P_{miss}.

[1] We assume again that the management time is negligible, as well as execution transfers between software and hardware.

[2] A simplifying assumption here is that all memory transfers have the same cost, which holds for the accelerators with simple memory access patterns.

E.3 Critical Section Speedup

If we extend our general-purpose system with a hardware accelerator implementing the critical code section and running within the codesigned application, the speedup achieved by the hardware accelerator is

$$Speedup_{criticalsection} = \frac{ET_{SW}}{ET_{HW}}, \tag{E.5}$$

where ET_{HW} is the overall execution time of the accelerator.

A designer (or an automated design flow) who decides to move a critical section from software to hardware may want to estimate the potential benefit first. In such situation, the CPD metric provides the means for the comparison. Here we show an example of its use. Recalling discussion from Chapter 5, for the speedup of the critical section (moved from software to hardware execution), we can write:

$$\begin{aligned} Speedup_{criticalsection} &= \frac{IC \times CPI \times T_{CPU}}{DC \times CPD_{HW} \times T_{HW}} = \frac{DC \times \frac{IC \times CPI}{DC} \times T_{CPU}}{DC \times CPD_{HW} \times T_{HW}} \\ &= \frac{CPD_{SW} \times T_{CPU}}{CPD_{HW} \times T_{HW}} \end{aligned} \tag{E.6}$$

if we assume the same word size for software and hardware, when accessing the data from memory. For instance, if we take an example of a code section originally executed to process 2048 words on a single-issue pipelined CPU (running at 300MHz, thus $T_{CPU} = 3.33ns$), with $IC = 20000$ and $CPI = 1.05$, we find $CPD_{SW} = (20000 \times 1.05)/2048 = 10.25$. On the other side, we have the corresponding hardware accelerator (accessing local memory as Figure 2.3 shows and running in FPGA at 100MHz, thus $T_{HW} = 10ns$) operating on the same data, with $CPD_{HW} = 1.2$. The CPD value close to 1 indicates a heavily-pipelined implementation of the accelerator, capable of processing one datum per cycle (assuming ideal accesses to the local memory that always take one cycle per access). In reality, the difference of 0.2 from the ideal CPD value accounts for an average number of cycles spent for hardware accelerator stalls and software activities to transfer the data from the main memory to the local memory—the average access time to the local memory in practice is not equal to one cycle.

In this particular example, moving the critical section to hardware would achieve therefore a speedup of:

$$Speedup = \frac{10.25 \times 3.33ns}{1.2 \times 10ns} = 2.84 \tag{E.7}$$

Appendix F

MPEG4 Hardware Reference

THE *Moving Picture Experts Group* (MPEG) is a working group of the *International Standardisation Organisation* (ISO) responsible for the development of international standards for compression, decompression, processing, and coded representation of moving pictures, audio and their combination [85]. A common way of describing outcoming MPEG standards is to use a reference code written in a high-level programming language (typically C programming language). One of the activities of the MPEG—within an ad-hoc group assigned to this task—is to develop a reference hardware description of the MPEG-4 algorithms written in the VHDL hardware-description language. Historically, designers would develop hardware accelerators for MPEG starting from the reference C code; having the reference description of the standard in VHDL would shortcut the design path from the software reference description to the accelerator hardware.

The reference hardware description group has developed a framework for platform-independent hardware accelerators called Virtual Sockets [97, 96]. One of the ongoing activities of the group is to integrate our concept of unified virtual memory for software and hardware into the existing framework [77]. The integration (1) will allow user IPs to be completely unaware of physical location of the accessed data and (2) there will be no need for the software programmer to do the explicit data transfers between the main memory and the IPs. In this appendix, we briefly show the existing MPEG-4 framework for reference hardware design and present the application of our unified virtual memory to it.

F.1 Virtual Socket Framework

The University of Calgary [97] has developed Virtual Socket framework for the Wildcard II reconfigurable PC card (shown in Figure F.1) from Annapolis Micro Systems [7]. The Wildcard II is an extension card for portable or wearable computers. It contains a Virtex-II device (X2CV3000) from Xilinx [130], external SDRAM (64MB) and SRAM (2MB) memory, analog-to-digital converter, a multi-channel DMA controller, and digital input/output ports. The reconfigurable device intended for accelerating applications is connected to the host computer over the PC Card bus (former PCMCIA interface).

Figure F.1: Wildcard II reconfigurable PC Card from Annapolis Micro Systems. The card contains a Xilinx Virtex-II device, PC interfacing logic, and on-card memories.

Figure F.2 shows Virtual Socket framework. The framework provides platform-independent connections (virtual sockets) to multiple user IPs (hardware modules implementing some MPEG functionality). Each socket allows user IPs to access four different on-chip (within the Xilinx Virtex-II device, registers and BRAM space) and on-card (SDRAM and SRAM off the Virtex-II device) memory spaces. Regarding on-chip memories, Virtual Socket assigns to each IP its own memory slot. It is the task of the programmer to control explicitly the IPs and copy the data from software to hardware. For example, if the data to be processed by the IP is larger than the available memory in the corresponding slot, the programmer has to partition the data and schedule the transfers.

F.2 Virtual Memory Extension

The ongoing extensions of Virtual Socket framework have the goal to adapt the existing framework and provide the virtual memory abstraction and shared address space for user software and hardware IPs. The burdensome task of data movements is delegated to the system layer, thus, releasing the programmer. The virtual memory extension mode is optional, meaning that MPEG hardware designers can decide whether or not to use the extension. Since the unified memory abstraction facilitates hardware and software interfacing and partitioning, an appealing possibility for the designers is to use the virtual memory extension at the early stages of the module design. During this phase, the system-level layer would provide memory-access profiling for a given hardware module. Later on, the obtained profile would be used

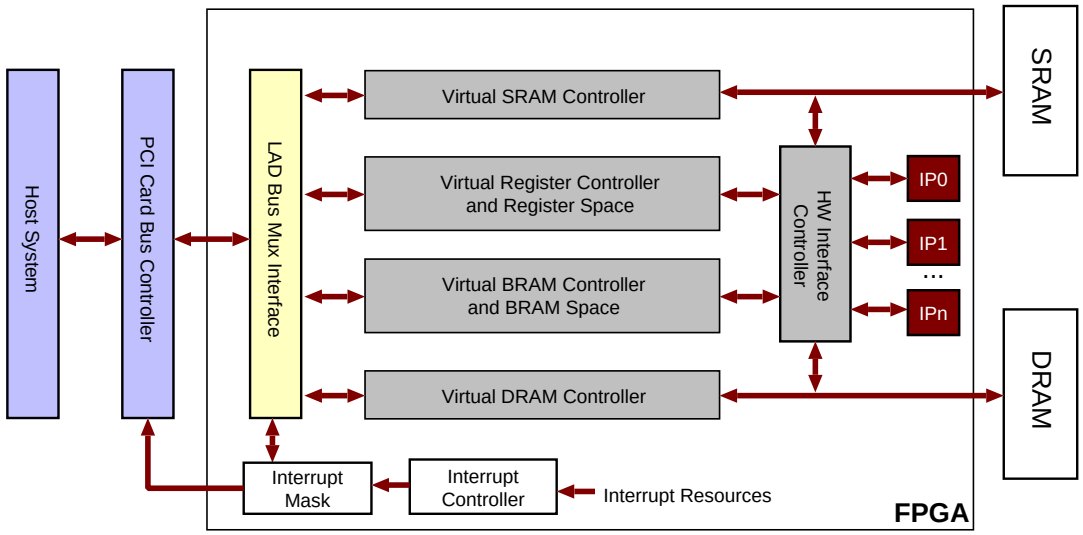


Figure F.2: Virtual Socket framework. The HW Interface Controller allows user IP blocks to access four different memory address spaces. The programmer explicitly transfers data from the host computer to these memories.

to tailor the system layer and optimise the memory transfers, thus minimising the virtual memory overhead.

Figure F.3 shows the integration of the WMU within the existing framework. One of the design tasks was to keep the IP interface as close as possible to the original Virtual Socket. To integrate the WMU, we have added an additional signal to the original interface. The signal called `Virt_mem_access` (asserted by an IP accessing the virtual memory) extends the interface and controls a switch box (shown in Figure F.3). During a virtual-memory access of the IP, the switch detours memory read or write addresses and related control signals to the hardware translation unit—the WMU; in turn, for a successful translation, the WMU generates the physical page number and passes it to the hardware interface controller of the original framework; finally, the controller performs the memory operation and returns the data from/to the IP block. In the case of an unsuccessful translation, the WMU generates an interrupt and the system software resolves its cause. For the moment, the WMU translates virtual memory accesses only to the BRAM address space. There is no slot limitation from the original framework.

The LAD bus connects the WMU directly to the host platform. Through the interface, the VMW manager can access the control register, the status register, and the address registers of the WMU. In the case of an access fault, the address register contains the IP-generated address causing the interrupt. The VMW software is supposed to check this address, copy the requested data in the BRAM space of the Virtual Socket platform, and update the TLB through the LAD bus. The virtualisation layer screens the programmer and the hardware designer of these events.

The WMU generates another signal (`Phys_page_num`, an additional port of the Hardware Interface Controller) acting as an input to the Hardware Interface Controller. The signal indicates the number of the page in the BRAM space where the

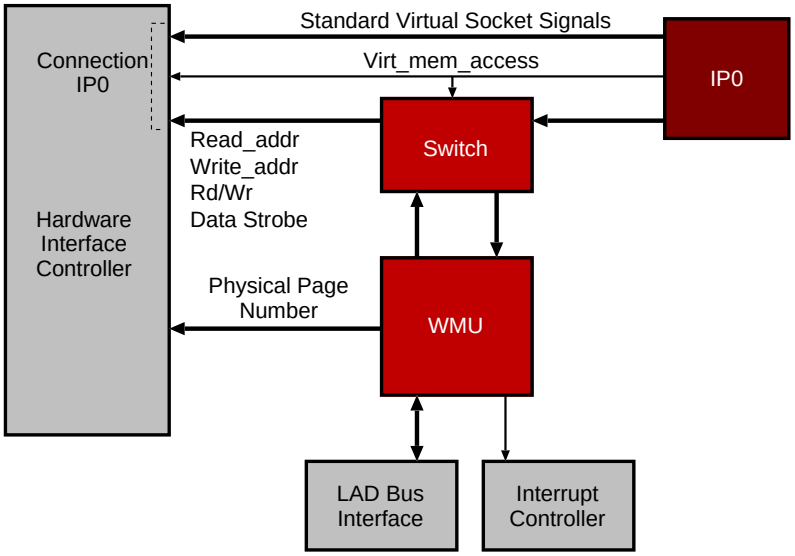


Figure F.3: WMU integration within the Virtual Socket framework. The WMU translates virtual memory addresses to the physical addresses of the BRAM address space. The programmer does not need any more to transfer the IP data explicitly.

requested data are found. The memory access protocol is slightly different than the original one [97] and relies on the few additional signals [77].

Bibliography

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David A. Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley Longman, Inc., Boston, 1986.

[3] Altera Corporation. *Excalibur Devices*, November 2002. [Online; accessed via `http://www.altera.com/literature/manual/mnl_arm_hardware_ref.pdf` on 10th December 2003].

[4] Don Anderson. *Pentium Processor System Architecture*. Mindshare Press, New York, 1994.

[5] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ed Komp, and David Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., April 2006.

[6] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, and Ed Komp. Programming models for hyprid FPGA-CPU computational components: A missing link. *IEEE Micro*, 24(4):42–53, July 2004.

[7] Annapolis Micro Systems, Inc., Annapolis, Md. *Wildcard-II Reference Manual*, 2004. [Online; accessed via `http://www.annapmicro.com/manuals/wcii_manual_v2_6.pdf` on April 30th 2006].

[8] ARM Inc. *AMBA Specification*, 1999. [Online; accessed via `http://www.arm.com/products/solutions/AMBA_Spec.html` on 7th September 2006].

[9] ARM Inc. *Application Binary Interface (ABI) for the ARM Architecture*, 2006. [Online; accessed via `http://www.arm.com/products/DevTools/ABI.html` on 30th April 2006].

[10] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[11] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 70–80, Napa Valley, Calif., April 1999.

[12] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, N.J., 1986.

[13] Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley, Reading, Mass., 2003.

[14] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 243–53, Monterey, Calif., November 1994.

[15] Gregory A. Baxes. *Digital Image Processing: Principles and Applications*. Wiley, New York, 1994.

[16] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–34, Seattle, Wash., June 1990.

[17] Jean-Luc Beuchat. Modular multiplication for FPGA implementation of the IDEA block cipher. In *Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors*, pages 412–22, The Hague, Netherlands, June 2003.

[18] Mattias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–53, Chicago, Ill., April 1994.

[19] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., Sebastopol, Calif., second edition, 2002.

[20] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications*, pages 327–36, Darmstadt, Germany, September 1996.

[21] Gordon Brebner. The swappable logic unit: a paradigm for virtual hardware. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 77–86, Napa Valley, Calif., April 1997.

[22] Gordon Brebner, Phil James-Roxby, and Chidamber Kulkarni. Hyper-programmable architecture for adaptable networked systems. In *Proceedings of the 15th International Conference on Application-specific Systems, Architectures and Processors*, pages 328–38, Galveston, Tex., September 2004.

[23] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, Boston, Mass., October 2004.

[24] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architure News*, 25(3):13–25, June 1997.

[25] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[26] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, André DeHon, and John Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, pages 605–14, Villach, Austria, August 2000.

[27] Eylon Caspi, André Dehon, and John Wawrzynek. A streaming multi-threaded model. In *In Proceedings of the Third Workshop on Media and Stream Processors*, pages 21–28, Austin, Tex., December 2001.

[28] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecapelle. *Custom Memory Management Methodology*. Kluwer Academic, Boston, Mass., 1998.

[29] Celoxica Ltd. *Handel-C Language Reference Manual*, 2004. [Online; accessed via `http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf` on 7th September 2006].

[30] Celoxica Ltd., Oxfordshire, UK. *Data Streaming Manager (Datasheet)*, 2005. [Online; accessed via `http://www.celoxica.com/techlib/files/CEL-W0506201CG6-46.pdf` on 7th September 2006].

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., second edition, 2001.

[32] Ingmar Cramm. Entwicklung und implementierung einer EdgeEngine zur erkennung von kanten in zukünftigen fahrerassistenzsystemen. Master thesis, Technische Universität München, Munich, May 2006.

[33] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Mateo, Calif., 1999.

[34] Michael Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10980–85, Munich, March 2003.

[35] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[36] Giovanni De Micheli. Hardware synthesis from C/C++ models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 382–83, Munich, March 1999.

[37] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. *Readings in Hardware/-Software Co-design*. Kluwer Academic, Boston, Mass., 2002.

[38] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, April 2000.

[39] Christophe Dubach. Java bytecode synthesis for reconfigurable computing platforms. Master thesis, École Polytechnique Fédérale de Lausanne (EPFL), January 2005.

[40] Stephen A. Edwards. The challenges of hardware synthesis from C-like languages. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 66–67, Munich, March 2005.

[41] Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. *Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers Inc., San Francisco, 2004.

[42] B. Flachs, S. Asano, S.H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. In *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 134–35, San Francisco, February 2005.

[43] Josef Fleischmann, Klaus Buchenrieder, and Rainer Kress. Java driven codesign and prototyping of networked embedded systems. In *Proceedings of the 36th Design Automation Conference*, pages 794–97, New Orleans, La., June 1999.

[44] Michael J. Flynn. *Computer Architecture—Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, Mass., 1995.

[45] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–58, Napa Valley, Calif., April 2005.

[46] Daniel Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Springer, Berlin, 2000.

[47] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 136–44, Napa Valley, Calif., April 1995.

[48] Cédric Gaudin. Module SOC/ARM sur FPGA. Master thesis, École Polytechnique Fédérale de Lausanne (EPFL), February 2003.

[49] Ferid Gharsalli, Samy Meftali, Frédéric Rousseau, and Ahmed A. Jerraya. Automatic generation of embedded memory wrapper for multiprocessor SoC. In *Proceedings of the 39th Design Automation Conference*, pages 596–601, New Orleans, La., June 2002.

[50] J. C. Gibson. The Gibson mix. Technical Report TR.00.2043, International Business Machines, Inc., Poughkeepsie, N.Y., June 1970.

[51] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Berlin, 2005.

[52] Sven A. Gowal. Operating system support for multithreaded reconfigurable hardware. Internship project, École Polytechnique Fédérale de Lausanne (EPFL), September 2005.

[53] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, Mass., 2002.

[54] Tom R. Halfhill. New patent reveals Cell secrets. *Microprocessor Report*, 3 January 2005.

[55] John R. Hauser and John 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., April 1997.

[56] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, Calif., third edition, 2002.

[57] Michael Herz, Reiner Hartenstein, Miguel Miranda, Erik Brockmeyer, and Francky Catthoor. Memory addressing organisation for stream-based reconfigurable computing. In *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems*, Dubrovnik, Croatia, September 2002.

[58] Paolo Ienne and Rainer Leupers, editors. *Customizable Embedded Processors: Design Technologies & Applications*. Systems on Silicon Series. Morgan Kaufmann, San Mateo, Calif., 2006.

[59] International Business Machines, Inc. *The CoreConnect Bus Architecture*, 2006. [Online; accessed via `http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture/` on 7th September 2006].

[60] Ayal Itzkovitz and Assaf Schuster. Distributed shared memory: Bridging the granularity gap. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, Rhodes, Greece, June 1999.

[61] Cédric Jeannaret. Synthesising Java bytecode to hardware. Semester project, École Polytechnique Fédérale de Lausanne (EPFL), February 2006.

[62] Java Native Interface specification, 2003. [Online; accessed via `http://java.sun.com/j2se/1.4.2/docs/guide/jni/` on 7th September 2006].

[63] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–73, Seattle, Wash., May 1990.

[64] Kaffe is not Java, 2006. [Online; accessed via `http://www.kaffe.org/` on 30th April 2006].

[65] Kevin Krewell. Cell moves into the limelight. *Microprocessor Report*, 14 February 2005.

[66] Holger Lange and Andreas Koch. Memory access schemes for configurable processors. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, pages 615–25, Villach, Austria, August 2000.

[67] Emanuele Lattanzi, Aman Gayasen, Mahmuth Kandemir, Vijaykrishnan Narayanan, Luca Benini, and Alessandro Bogliolo. Improving Java performance by dynamic method migration on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop*, pages 134–141, Santa Fe, N. Mex., April 2004.

[68] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., December 1997.

[69] Tien-Lung Lee and Neil W. Bergmann. An interface methodology for retargetable FPGA peripherals. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 167–73, Las Vegas, Nev., June 2003.

[70] Marc Leeman, David Atienza, Chantal Ykman, Francky Catthoor, Jose Manuel Mendias, and Geert 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, August 2003.

[71] Christopher K. Lennard, Patrick Schaumont, Gjalt De Jong, Anssi Haverinen, and Pete 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 576–83, Paris, March 2000.

[72] P.H.W. Leong, M.P. Leong, O.Y.H Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H Lee. Pilchard—a reconfigurable computing platform with memory slot interface. In *Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 170–70, Napa Valley, Calif., April 2001.

[73] Markus Levy. Java to go: The finale. *Microprocessor Report*, 4 June 2001.

[74] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, The Pennsylvania State University, University Part, Penn., August 1988.

[75] The Linux kernel archives. `http://www.kernel.org/`, 2006.

[76] Robert Love. *Linux Kernel Development*. Novell Press, Waltham, Mass., second edition, 2005.

[77] Adding virtual memory support to MPEG4 hardware reference platform, April 2006. MPEG4-Part9 ISO/IEC JTC1/SC29/WG11 M13117, Montreux, Switzerland.

[78] Damien Lyonnard, Sungjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the 38th Design Automation Conference*, pages 518–23, Las Vegas, Nev., June 2001.

[79] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proceedings of the 41st Design Automation Conference*, pages 954–59, San Diego, Calif., June 2004.

[80] Evangelos P. Markatos and Manolis G. H. Katevenis. User-level DMA without operating system kernel modification. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 322–31, San Antonio, Tex., February 1997.

[81] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers Inc., San Francisco, 1994.

[82] MediaCrypt. *IDEA Algorithm*, 2003. [Online; accessed via `http://www.mediacrypt.com/_pdf/IDEA_Technical_Description_0105.pdf` on 7th September 2006].

[83] Chad L. Mitchell and Michael J. Flynn. A workbench for computer architects. *IEEE Design and Test of Computers*, 5(1):19–29, January–February 1988.

[84] Overview of MPEG-4 Part-9 reference HW description. `http://www.chiariglione.org/MPEG/technologies/mp04-hrd/`, October 2005. International Organisation for Standardisation.

[85] MPEG: Moving pictures experts group. `http://www.chiariglione.org/mpeg/`, 2006. International Organisation for Standardisation.

[86] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–58, Philadelphia, Penn., 1996.

[87] Shubhendu S. Mukherjee and Mark D. Hill. Making network interfaces less peripheral. *Computer*, 31(10):70–6, October 1998.

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

[89] Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy 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.

[90] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, Ill., April 1994.

[91] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-on-Chip*. Kluwer Academic, Boston, Mass., 1999.

[92] David Pellerin and Scott Thibault. *Practical FPGA programming in C*. Prentice-Hall, Englewood Cliffs, N.J., 2005.

[93] Karin Petersen and Kai Li. Multiprocessor cache coherence based on virtual memory support. *Journal of Parallel and Distributed Computing*, 29(2):158–78, September 1995.

[94] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 184–85,592, San Francisco, February 2005.

[95] Jelica Protić, Milo Tomasević, and Veljko Milutinović, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, Calif., 1997.

[96] Reference for virtual socket API function calls, January 2006. MPEG4-Part9 ISO/IEC JTC1/SC29/WG11 M12788, Bangkok, Thailand.

[97] Tutorial on an integrated virtual socket hardware-accelerated co-design platform, January 2006. MPEG4-Part9 ISO/IEC JTC1/SC29/WG11 M12789, Bangkok, Thailand.

[98] Rahul Razdan and Michael 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., November 1994.

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

[100] Radu Rugina and Martin Rinard. Span: a shape and pointer analysis package. Technical Report TM-581, Massachusetts Institute of Technology, Boston, Mass., June 1998.

[101] John E. Savage. The performance of multilective VLSI algorithms. *Journal of Computer and System Sciences*, 29(2):243–73, October 1984.

[102] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison Wesley Longman, Inc., Reading, Mass., 1998.

[103] Ioannis Schoinas and Mark D. Hill. Address translation mechanisms in network interfaces. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 219–30, Las Vegas, Nev., February 1998.

[104] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, second edition, 2005.

[105] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, Mass., 2000.

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

[107] Richard L. Sites and Richard L. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, Bedford, Mass., second edition, 1995.

[108] Donald Soderman and Yuri Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. In *Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 339–42, Napa Valley, Calif., April 1998.

[109] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–82, Anchorage, Ala., May 2002.

[110] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Upper Saddle River, N.J., 2005.

[111] Walter Stechele. Video processing using reconfigurable hardware acceleration for driver assistance. In *Workshop on Future Trends in Automotive Electronics and Tool Integration at DATE 2006*, Munich, March 2006.

[112] Walter Stechele, Alvado L. Cárcel, Stephan Herrmann, and J.Lidón Simón. A coprocessor for accelerating visual information processing. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 26–31, Munich, March 2005.

[113] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Paris, March 2002.

[114] Chris Sullivan and Milan Saini. Software-compiled system design optimizes Xilinx programmable systems. *Xcell Journal*, (46):32–37, Summer 2003.

[115] Sun Microsystems. *The Java HotSpot Virtual Machine (White Paper)*, 2002. [Online; accessed via `http://java.sun.com/javase/technologies/hotspot.jsp` on 7th September 2006].

[116] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, N.J., second edition, 2001.

[117] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[118] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, C-53(11):1363–75, November 2004.

[119] Miljan Vuletić, Christophe Dubach, Laura Pozzi, and Paolo Ienne. Enabling unrestricted automated synthesis of portable hardware accelerators for virtual machines. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 243–48, Jersey City, N.J., September 2005.

[120] Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors. In *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, pages 596–605, Antwerp, Belgium, August 2004.

[121] Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Virtual memory window for a portable reconfigurable cryptography coprocessor. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 24–33, Napa Valley, Calif., April 2004.

[122] Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers*, 22(2):102–13, March–April 2005.

[123] Herbert Walder and Marco Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10290–95, Munich, March 2003.

[124] Jian Wang. An FPGA based software/hardware codesign for real time video processing. Diploma Thesis LITH-ISY-EX–06/3756–SE, Linköping University, Linköping, Sweden, March 2006.

[125] Wikipedia. Application binary interface — Wikipedia, the free encyclopedia, 2006. [Online; accessed on 30th April 2006].

[126] Wikipedia. System software — Wikipedia, the free encyclopedia, 2006. [Online; accessed on 10th April 2006].

[127] R. Wilson, C. French, C. Wilson, J. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, M. Tseng, Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29:31–37, December 1994.

[128] Wayne Wolf. *Computers as Components: Principles of Embedded Computer Systems Design*. Morgan Kaufmann, San Mateo, Calif., 2001.

[129] Sven Wuytack, Julio L. da Silva Jr., Francky Catthoor, Gjalt de Jong, and Chantal Ykman-Couvreur. Memory management for embedded network applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-18(5):533–44, May 1999.

[130] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X User Guide*, March 2005. [Online; accessed via `http://direct.xilinx.com/bvdocs/userguides/ug012.pdf` on 10th December 2005].

[131] A. Yamada, K. Nishida, R. Sakurai, A. Kay, and T. Nomura, T.and Kambe. Hardware synthesis with the Bach system. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 6, pages 366–69, Orlando, Fla., May–June 1999.

[132] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–35, Vancouver, June 2000.

[133] Albert Y. Zomaya, editor. *Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies*. Springer, Berlin, 2006.

[134] Steve. Zucker and Kari Karhi. System V application binary interface: PowerPC processor supplement. Technical Report 802-3334-10, Sun Microsystems, Inc., Mountain View, Calif., September 1995.

Curriculum Vitae

VULETIĆ MILJAN

PERSONAL DATA

Date of birth: December 10, 1971
Place of birth: Zrenjanin, Vojvodina, Yugoslavia
Citizenship: Serbia

RESEARCH INTERESTS

Reconfigurable computing systems, OS support for heterogeneous and reconfigurable computing, dynamic and partial reconfiguration, distributed shared-memory for reconfigurable computing, platform-independent hardware accelerators for multimedia applications, synthesis of high-level languages to hardware, instruction set specialisation for embedded processors.

EDUCATION

Jan 2001–Aug 2006 **Ph.D. in Computer Engineering**, Processor Architecture Laboratory (LAP), School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland. **Thesis title:** *Unifying Software and Hardware of Multithreaded Reconfigurable Applications within Operating System Processes.* **Advisor:** prof. Paolo Ienne.

Mar 1997–Mar 1999 **M.S. Graduate School**, School of Electrical Engineering, University of Belgrade, Serbia, Master project done and graduate exams passed, but thesis defense never scheduled (because of political turmoils in former Yugoslavia). GPA: 10.00/10.00. **Project title:** *Suboptimal Detection of Telemetry Signals: Functional Simulation and VLSI Implementation.* **Advisor:** prof. Veljko Milutinović.

Oct 1990–Feb 1997 **Dipl.Ing. in Electrical Engineering**, major in Computer Science and Engineering, School of Electrical Engineering, University of Belgrade. GPA: 8.93/10.00.

EMPLOYMENT

Jan 2001–present Research and Teaching Assistant, Processor Architecture Laboratory, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.

Oct 1997–Oct 1998 Teaching Assistant (part-time), Computer Science and Engineering Department, University of Belgrade, Serbia.

Jul 1994–Dec 2000 Chief System Engineer, OpenNet Internet provider (the first Internet provider in Yugoslavia), Radio B92, Serbia.

RESEARCH EXPERIENCE

Graduate Research Assistant (2001–present) to prof. Paolo Ienne, Processor Architecture Laboratory, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland. Research activities:

- Investigated graph-based methods for identification of instruction set extensions.
- Proposed a seamless integration of software and hardware in reconfigurable computing systems based on the extensions in system software and system hardware.
- Proposed OS-based dynamic optimisation techniques to screen users (software programmers and hardware designers) from memory-copy latencies in reconfigurable computing systems.
- Supervised the development of a synthesis flow, from a high-level programming language to reconfigurable hardware.

Research Assistant (1997–2000) to prof. Veljko Milutinović, Department of Computer Science and Engineering, School of Electrical Engineering, University of Belgrade, Serbia. Research activities:

- Proposed a low-quantisation method for detecting telemetry signals and suggested a detector implementation in VLSI.
- Investigated the application of low-quantisation detection methods to power measurement.

TEACHING EXPERIENCE

Teaching Assistant (2001–present) to prof. Paolo Ienne, Processor Architecture Laboratory, School of Computer and Communication Sciences, EPFL, *Computer Architecture I and II*, 2nd and 3rd year undergraduate courses. As a part of teaching activities but within the research scope, led about 20 students during their diploma, semester, and internship projects.

Teaching Assistant (1997–1998) to prof. Veljko Milutinović, Department of Computer Science and Engineering, School of Electrical Engineering, University of Belgrade, Serbia, *Computer VLSI Systems*, 5th year undergraduate course.

PROFESSIONAL EXPERIENCE

Member of a VLSI engineering team (Jun 1996–Dec 1996), implementing R2 signalization (analog telephony switching protocol) on an FPGA-based detection board.

Design engineer (May 1997–Dec 1997), involved in a research project for *eT Communications*, in the field of VLSI and ASIC design for telemetry applications.

Research analyst (May 1998 - September 1998) for *eT Communications* in the field of VLSI for solid state power meter design.

Engineering-team leader (Spring 1998), American Center Internet Classrooms, developed specialized software solution for public Internet classrooms (Belgrade, Podgorica), the project in association with the USA Information center.

Engineering-team leader (Fall 1998), realization of VPN (Virtual Private Network) secure network project, in cooperation with XS4ALL Netherlands Internet provider, employing and customizing VPN technologies.

Principal software engineer (Jul 1999–Dec 1999), design and implementation of an automatic application generator from a database model (Borland Delphi code generator, based on the Oracle Designer 2000 model).

Software architect (May 2000), analysis, modeling and programming for the information system development of the Montenegro national airline company (Montenegro Airlines).

AWARDS, FELLOWSHIPS, HONORS

2001	Research assistantship from Infineon, Munich, Germany
1997–2000	Research fellowship from the Ministry of Education, Serbia
1997	First (out of aprox. 200) on the graduate enrollment list of School of Electrical Engineering, University of Belgrade, Serbia

PROFESSIONAL ACTIVITIES

Reviewer for Design Automation Conference (DAC 2003), International Conference on Computer Aided Design (ICCAD 2004), Asynchronous Circuits and Systems (ASYNC 2006), Design, Automation, and Test in Europe (DATE 2006). Participating in the work of the hardware reference implementation group (ISG) of the MPEG (a working group of ISO/IEC in charge of the development of international audio/video standards).

PROFESSIONAL SOCIETIES

Member of the ACM and IEEE (including IEEE Computer Society membership).

SKILLS

Programming languages: C/C++, Java, PHP, awk, perl, bash, Pascal, FORTRAN, Lisp, Basic, assembler (PDP-11, Z80, X86, MIPS, ARM). **Hardware description languages:** VHDL, Verilog, ISP'. **Operating systems:** FreeBSD, Linux and Digital Unix—advanced administrator, kernel and user programmer; Windows—administrator and user programmer, VAX—advanced user. **EDA tools:** ModelSim, HDL Designer, Synplify, Leonardo, Quartus (Altera), Xilinx ISE and Platform Studio.

LANGUAGES

Serbian (native), English (fluent, spoken and written), French (fluent, spoken and written), Italian and German (rudimentary communication and understanding).

MISCELLANEOUS

Music (classic and jazz), internet activism (B92.NET and XS4ALL.NET projects for freedom of speech in Milošević's Serbia—Time magazine, March 1997, and WIRED magazine, April 1997), sports (soccer, alpinism, hiking, running).

PUBLICATIONS

Journal Articles

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne, "Virtual memory window for application-specific reconfigurable coprocessors" *In IEEE Trans. on VLSI*, 14(8):910–15, August 2006.

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Seamless hardware-software integration in reconfigurable computing systems. *In IEEE Design and Test of Computers*, 22(2):102–13, March–April 2005.

Conference Papers

- Miljan Vuletić, Paolo Ienne, Christopher Claus, and Walter Stechele. Multithreaded Virtual-Memory-Enabled Reconfigurable Hardware Accelerators. *In Proceedings of the IEEE International Conference on Field Programmable Technology (FPT 06)*, Bangkok, Thailand, December 2006. Accepted for publication.

- Miljan Vuletić, Christophe Dubach, Laura Pozzi, and Paolo Ienne. Enabling unrestricted automated synthesis of portable hardware accelerators for virtual machines. *In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS 05)*, Jersey City, N.J., September 2005.

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing. *In Proceedings of the 15th International Conference on Application-specific Systems, Architectures and Processors (ASAP 04)*, Galveston, Tex., September 2004.

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors. *In Proceedings of the 14th International Conference on Field-Programmable Logic and Applications (FPL 04)*, Antwerp, Belgium, August 2004.

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *In Proceedings of the 41st Design Automation Conference (DAC 04)*, San Diego, Calif., June 2004.

- Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Virtual memory window for a portable reconfigurable cryptography coprocessor. *In Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 04)*, Napa Valley, Calif., April 2004.

- Miljan Vuletić, Ludovic Righetti, Laura Pozzi, and Paolo Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 04)*, Paris, February 2004.

- Miljan Vuletić, Ludovic Righetti, Laura Pozzi, and Paolo Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. *In Proceedings of the 2nd Workshop on Application Specific Processors (WASP 03)*, San Diego, Calif., December 2003.

- Ajay Kumar Verma, Kubilay Atasu, Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *In Proceedings of the 1st Workshop on Application Specific Processors (WASP 02)*, Istanbul, November 2002.

- Laura Pozzi, Miljan Vuletić, and Paolo Ienne. Automatic topology-based identification of instruction-set extensions for embedded processors. *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 02)*, page 1138, Paris, March 2002.

- Miljan Vuletić, Goran Davidović, and Veljko Milutinović, "Suboptimal Detection of Telemetry Signals: Functional Simulation and VLSI Implementation," *In Proceedings of Sixth International Symposium on Modeling and Analysis of Computer and Telecommunication Systems (MASCOTS 98)*, Montreal, Canada, July 19-24, 1998.

Patents

- Patent application (EP 04 10 1629, Virtual Memory Window with Dynamic Prefetching Support for Portable Reconfigurable Coprocessors) approved by European patent office (October 2005).

Selected Talks

- Virtual Memory Management for Multithreaded Hardware Accelerators, Institut National Polytechnique de Grenoble, Grenoble, June 2006
- Virtual Memory Management for Reconfigurable Hardware Accelerators, Technical University Munich, Munich, December 2005
- Seamless Integration of Hardware and Software in Reconfigurable Computing Systems, UC Davis, Davis, CA, May 2005
- Seamless Integration of Hardware and Software in Reconfigurable Computing Systems, Xilinx Corporation, San Jose, CA, May 2005
- Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing, University of Alabama, Huntsville, AL, October 2004
- Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing, Altera Corporation, San Jose, CA, September 2004

Others

- Paolo Ienne, Laura Pozzi, and Miljan Vuletić. On the limits of processor specialisation by mapping dataflow sections on ad-hoc functional units. *Technical Report 01/376*, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, December 2001.
- Laura Pozzi, Miljan Vuletić, and Paolo Ienne. Automatic topology-based identification of instruction-set extensions for embedded processors. *Technical Report 01/377*, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, December 2001.
- Miljan Vuletić, Jasna Ristić-Djurović, Milivoj Aleksić, Veljko Milutinović, and Michael Flynn. Per window switching of window characteristics: wave-pipelining vs. classical design. *In IEEE TCCA Newsletter*, September 1997.