methods have their own strengths: net-driven method yields better utilization of resources while path-driven method results in better timing results. In addition, the net/path-driven method achieves good tradeoff between resource utilization and timing results.

Table IV studies the impact of various cost factors used in our SA-based refinement. The first three algorithms take only hgsw, mse, or xlp objective into account, whereas the forth algorithm uses a combination of all three objectives and is selected as baseline. First, we obtain 3% more hgswsaving with $\mathrm{hgsw-only}$algorithm compared to the baseline. However, this saving comes with 438% and 39% increase on mse and xlp cost. Second, the mse saving with $\mathrm{mse-only}$ algorithm compared to the baseline is almost negligible while the hgsw and xlp cost increase by 16% and 28%, respectively. Finally, the xlp saving with $\mathrm{xlp-only}$ algorithm compared to the baseline is 15% while the hgsw and mse cost increase by 13% and 64%, respectively. These results reveal that there may be a little improvement for a certain metric if SA focuses only on that metric. However, these individual savings come with huge degradation on other metrics that are ignored. Thus, the combined cost function proves to be the best approach.

VII. Conclusion

This paper focused on making our large-scale floating-gate-based FPAA technology more accessible by providing the first physical synthesis tool. Our analog CAD tool automates the placement of analog circuit components on a target large-scale FPAA. Our placement algorithm incorporates a performance metric that takes into account signal degradation and circuit parasitics under various device-related constraints. Our experimental results demonstrated the effectiveness of our new approaches for solving this new problem.

References

[1] T. Hall, C. Twigg, P. Hasler, and D. Anderson, "Developing large-scale field-programmable analog arrays," in *Proc. Parallel Distrib. Process. Symp.*, 2004, pp. 26–30.

[2] P. Hasler, C. Diorio, B. A. Minch, and C. A. Mead, "Single transistor learning synapses," in *Advances in Neural Information Processing Systems 7*. Cambridge, MA: MIT Press, 1995, pp. 817–824.

[3] J. D. Gray, C. M. Twigg, D. N. Abramson, and P. Hasler, "Characteristics and programming of floating-gate pFET switches in an FPAA crossbar network," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2005, pp. 23–26.

[4] H. Wang and S. Vrudhula, "Behavioral synthesis of field programmable analog array circuits," *ACM Trans. Design Autom. Electron. Syst.*, pp. 563–604, 2002.

[5] S. Ganesan and R. Vemuri, "Behavioral partitioning in the synthesis of mixed analog-digital systems," in *Proc. ACM Design Autom. Conf.*, 2001, pp. 133–138.

[6] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1–12, 1994.

[7] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 1997, pp. 213–222.

[8] M. Pedram, B. Nobandegani, and B. Preas, "Design and analysis of segmented routing channels for row-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1266–1274, 1994.

[9] B. Cochrun and A. Grabel, "A method for the determination of the transfer function of the electronic circuits," *IEEE Trans. Circuit Theory*, vol. CT-20, no. 1, pp. 16–20, Jan. 1973.

[10] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. COM-28, no. 1, pp. 84–95, Jan. 1980.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *Proc. ACM Design Autom. Conf.*, 1997, pp. 526–529.

Virtual Memory Window for Application-Specific Reconfigurable Coprocessors

Miljan Vuletić, Laura Pozzi, and Paolo Ienne

***Abstract*— The complexity of hardware/software (HW/SW) interfacing and the lack of portability across different platforms, restrain the widespread use of reconfigurable accelerators and limit the designer productivity. Furthermore, communication between SW and HW parts of codesigned applications are typically exposed to SW programmers and HW designers. In this work, we introduce a virtualization layer that allows reconfigurable application-specific coprocessors to access the user-space virtual memory and share the memory address space with user applications. The layer, consisting of an operating system (OS) extension and a HW component, shifts the burden of moving data between processor and coprocessor from the programmer to the OS, lowers the complexity of interfacing, and hides physical details of the system. Not only does the virtualization layer enhance programming abstraction and portability, but it also performs runtime optimizations: by predicting future memory accesses and speculatively prefetching data, the virtualization layer improves the coprocessor execution—applications achieve better performance without any user intervention. We use two different reconfigurable system-on-chip (SoC) running Linux and codesigned applications to prove the viability of our concept. The applications run faster than their SW versions, and the overhead due to the virtualisation is limited. Dynamic prefetching in the virtualisation layer further reduces the abstraction overhead.**

***Index Terms*—Codesign, coprocessors, dynamic prefetching, operating system (OS), reconfigurable computing.**

I. Introduction

Blending two computational paradigms (temporal computation on standard processors and spatial computation in reconfigurable hardware) supported by reconfigurable system-on-chip (SoC) devices [1], [2] is a well-known way to increase performance: critical code sections or entire software functions are mapped to reconfigurable hardware accelerators. When it comes to interfacing the application-specific coprocessors with the rest of the reconfigurable SoC: 1) programmers must be aware of data partitioning and memory transfers and 2) hardware designers have to account for different architectural details of the host platform. The memory transfers can particularly burden the programmer, if shared memory accessible by processor and field-programmable gate array (FPGA) is smaller than a dataset to process.

We introduce an abstraction layer for virtualization of hardware/software (HW/SW) interfacing. A lightweight platform-specific hardware and an operating system (OS) extension reduce the burden of SW programmers and HW designers: 1) programmers can write software that invokes reconfigurable coprocessors as if they were software functions—there is no need for explicit data transfers, passing memory pointers is just enough and 2) designers can write coprocessors that access the user virtual memory through a virtual memory window—there are neither physical constraints on addressing nor on the interface memory size. Our contribution shifts the burden of moving data between processor and coprocessor from the programmer to the OS. Codesigned applications become fully platform independent with only a limited penalty.

Manuscript received April 23, 2004; revised January 23, 2006.

M. Vuleti and P. Ienne are with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne CH-1015, Switzerland (e-mail: miljan.vuletic@epfl.ch; paolo.ienne@epfl.ch).

L. Pozzi was with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne CH-1015, Switzerland. She is now with the Faculty of Informatics, University of Lugano, Lugano CH-6900, Switzerland (e-mail: laura.pozzi@unisi.ch).

Digital Object Identifier 10.1109/TVLSI.2006.878481

II. Related Work

Some researchers [3] consider using the OS to manage FPGA resources (i.e., reconfigurable chip area used for executing different coprocessors); on our side, we use the OS to manage interfacing resources. The two concepts are orthogonal and complementary—future systems may have to implement solutions for both.

Several approaches exist that introduce OS extensions supporting user-controlled interfacing of reconfigurable hardware. In contrast to our approach, where the memory transfers are done implicitly by the OS, a typical solution [4] exposes the communication to the programmer through a limited-size interface memory mapped to the user space. An advanced OS-based solution [5] introduces a hardware abstraction layer (HAL) responsible for communication between SW and HW. Unlike our virtual memory approach, it assumes a specific communication scheme based on message passing: HW tasks generate no memory addresses but send messages through a message-passing interface; on the SW side, tasks rely on the message-passing application programming interface (API). Our scheme assumes no specific API, but allows both user SW and user HW to share any location of practically unlimited virtual memory.

An industrial solution [6] introduces a SW (C language) and HW (Handel-C language) API supported by an intermediate system layer—data streaming manager (DSM). The DSM provides platform-independent and stream-oriented communication between SW and HW. 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. Our approach has no such limitation: the programmer is screened from data transfers and the HW can generate any memory access pattern.

III. VMW

The virtual memory window (VMW) addresses the problem of nonstandard HW/SW interfacing [7] by reusing a simple and well-known concept of virtual memory. The VMW simplifies the programming paradigm by enabling the coprocessors to share the virtual memory address space with user applications.

Programmers of a computing platform running an OS are abstracted from the characteristics of the physical memory system [8]. The addresses known to the programmer are virtual in that they describe a memory system with no relation to the real one. The virtual memory manager (VMM) of the OS supports the programmer's illusion and is assisted in HW by the memory management unit (MMU). The approach typically results in suboptimal performance but the advantages are overwhelming: 1) programming is made simpler and 2) code becomes more portable.

We aim to extend these advantages to application-specific coprocessors. Our goal is to have an application (in a high-level language—e.g., C or C++) and the corresponding coprocessor (in a HW description language—e.g., VHDL or Verilog) sharing the same virtual memory address space. This would provide transparent communication between the coprocessor and the user-space software, and portability of reconfigurable applications. The programmer of a reconfigurable application should use coprocessors without any knowledge of their memory access scheme. Similarly, the coprocessor designer should follow the same abstraction and generate abstract addresses rather than specifying physical addresses.

A. Basic Concepts

To allow the unified virtual memory between SW and HW, a local memory accessible directly by the FPGA is used as a coprocessor's VMW. Two elements forming the virtualization layer are added to the basic system: 1) window management unit (WMU), a HW device similar to a classic MMU that performs the translation from virtual to physical addresses and 2) VMW manager, an OS part similar to the VMM that steers the translation and maintains data transfers (from/to the main memory) transparently to the user application.

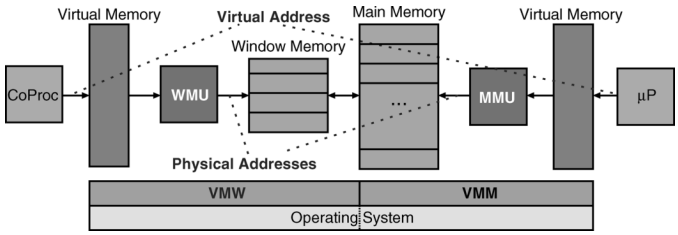

Fig. 1. VMW for reconfigurable coprocessors. Thanks to a hardware translation engine (WMU) and an OS extension (VMW manager), the coprocessor can share the address space with the user application and access the user-space virtual memory.

```
/* (a) Pure software version */
...
idea_cipher_fun(A, B, n64);
...
/* (b) Typical coprocessor version */
...
data_chunk = DP_SIZE / 2; data_pt = 0;
while (data_pt < n64 * 8) {
  memcpy(DP_BASE, A + data_pt, data_chunk);
  *IDEA_CTRL_REG = START;
  while(*IDEA_STATUS_REG != FINISH);
  *IDEA_STATUS_REG = INIT;
  memcpy(B + data_pt, DP_BASE + data_chunk, data_chunk);
  data_pt += data_chunk;
}
...
```

Fig. 2. Motivational example. Different invocations of the IDEA function: (a) pure SW version and (b) typical coprocessor version.

Fig. 1 shows how the VMW integrates to a virtual memory system. A user application running on the main processor and its corresponding coprocessor belong to the same OS process and have the same virtual address space. However, the coprocessor accesses the virtual memory through a different translation path (through the WMU and the window memory managed by the VMW manager). While the coprocessor is accessing the window memory through the WMU, the main processor may run any other runnable process scheduled by the OS. The OS invokes the VMW manager only when the address requested by the coprocessor is not present in the window memory: the WMU reports this event by raising an interrupt.

Benefits of unifying the memory pictures from the main processor and the coprocessor side are as follows. 1) Programming SW is made simpler—calling a coprocessor from a user application is as simple as a function call, no need for explicit data communication. 2) Designing HW is free of interface memory constraints (except complying to the WMU interface)—the coprocessor can access data anywhere in the user space, available virtual memory is practically unlimited. 3) Application SW and accelerator HW are made portable—hiding platform-related details behind the VMW manager and the WMU provides platform independence.

B. Motivating Example

Fig. 2 shows simplified code excerpts of the IDEA cryptography application that invokes either a SW function (a) or a HW coprocessor (b) to encrypt/decrypt an input vector A of 64-bit elements and store the results into an output vector B. In the case of the typical coprocessor version, the programmer has to take care of cumbersome details

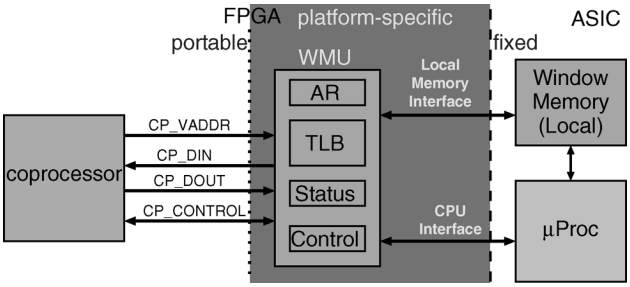

Fig. 3. Coprocessor connected through a WMU. The WMU translates virtual (coprocessor generated) to physical addresses of the on-chip memory.

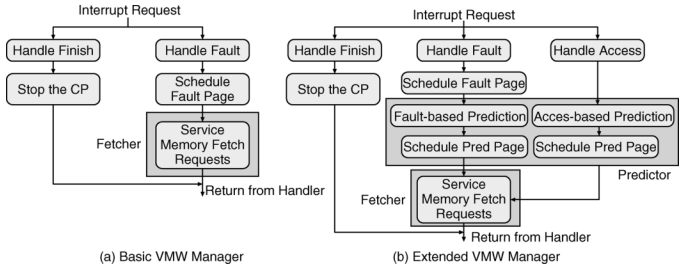

Fig. 4. Basic and extended VMW manager architectures.

(a similar task burdens the HW designer): 1) system-specific address calculation (platform related) and 2) relatively straightforward but burdensome data partitioning and transfers (coprocessor related). On the other side, the VMW-based programming completely resembles the pure SW version (as shown in Section V-B and Fig. 5): a clean and transparent interface to the HW. Not only that it hides memory transfers and data partitioning from the programmer, but it also assumes no particular memory access pattern of the coprocessor. The VMW-based system can process dynamically allocated data (e.g., objects scattered on the heap) without any additional burden on the programmer's side.

IV. WMU

The WMU decouples the design of application-specific coprocessors from the host platform. It is a platform-specific element which is ported once per reconfigurable SoC. As the MMU does on the CPU side, the WMU translates virtual addresses demanded by the coprocessor. The WMU also defines a standardised hardware interface for the VMW-based coprocessors. It can support multiple operation modes, i.e., different page sizes and the number of pages of the window memory.

A. WMU Structure

Fig. 3 shows how a VMW-based coprocessor is interconnected to the WMU. The standard interface consists of virtual address lines (`CP_VADDR`), data lines (`CP_DIN` and `CP_DOUT`), and control lines (`CP_CONTROL`). Platform-specific signals connect the WMU with the rest of the system. The presence of the translation lookaside buffer (TLB) inside the WMU emphasizes its similarity with a conventional MMU [8]. The TLB translates the upper part of the coprocessor address (most significant bits) to a physical page number of the window memory. Due to limitations of the FPGA technology and design methodologies different than in the case of ASICs, our TLB design performs the translation in multiple cycles.

If the coprocessor tries accessing the data which is not present in the local memory, the WMU generates an interrupt and requests the OS handling. While the coprocessor is stalled, the VMW manager: 1) reads the address register (`AR`) to find out the address that generated the fault; 2) transfers the corresponding page from the main memory and updates the TLB state; and 3) resumes the coprocessor. The OS provides transparent dynamic allocation of memory resources (i.e., shared or dual-port memory) between processor and coprocessor: the programmer can avoid explicit data movements.

B. Coprocessors for WMU

The HDL ports of the VMW-based coprocessor module are predefined by the WMU HW interface (shown in Fig. 3). The HW designer writes the coprocessors 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 SW. The HDL code of the VMW-based coprocessor [9]: 1) does not embody any detail related to the memory interfacing; 2) has no limit on the size of the processed data; and 3) is not concerned about physical data location.

V. VMW Manager

The VMW manager provides two functionalities: 1) a system call to access and control the coprocessor and 2) management functions to respond to WMU requests (similarly as the VMM does). The system service is called `FPGA_EXECUTE`. It passes data pointers and parameters to the HW, initializes the WMU, launches the coprocessor, and puts the calling process in sleep mode (avoiding consistency problems of simultaneous accesses to multiple data copies). The SW designer can pass data references to the coprocessor, without any particular preparation; the HW designer implements a coprocessor having in mind no specific data addresses—it fetches all necessary references through a standardized initialization protocol.

A. Manager Architecture

The window memory is logically organized in pages, as in typical virtual memory systems. The data accessed by the coprocessor are mapped to these pages. The OS keeps track of the occupied pages and the corresponding objects. Through memory protection policies, it prevents the coprocessor to access forbidden memory regions.

Fig. 4(a) shows the basic architecture of the VMW interrupt handler. There are two possible requests: 1) *page fault*—the coprocessor attempted an access of an address not currently in the window memory and 2) *end of operation*—the coprocessor signals the end of operation to the main processor and the manager then ensures that the user memory reflects correctly the state of the window memory. For a *page fault*, the OS rearranges the current mapping to the window memory and possibly selects a page for eviction—if dirty, its contents are copied back to the user-space memory; the OS schedules the missing page for transfer (done by the *Fetcher*), updates the WMU state and resumes the coprocessor. For an *end of operation*, the VMW manager copies back to the user space all dirty pages currently residing in the window memory and transfers the execution back to application SW.

B. Example Programming for VMW

For calling the VMW-based IDEA coprocessor, the programmer simply replaces the original SW function in the code with the call to a VMW library function (Fig. 5). In contrast to the typical approach [Fig. 2(b)], our approach is as simple and elegant as the original function call: the library function hides all memory interfacing details. Just before the call to the OS service, an array of the `param` data structure is used to pass parameters and data pointers to the coprocessor. The VMW manager copies the data dynamically to/from the window memory as requested/produced by the coprocessor, and everything is completely hidden from the SW programmer.

```
/* VMW-based coprocessor 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;
  FPGA_EXECUTE(&param);
}
```

Fig. 5. Library function for calling the VMW-based coprocessor version. Calling this function is to all practical purposes identical to the pure SW version and fully system-detail agnostic.

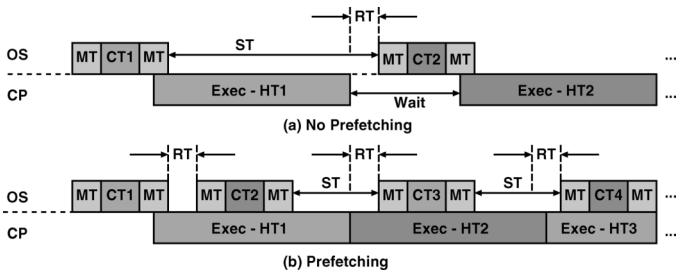

Fig. 6. The OS module (OS) activities related to coprocessor execution (CP). (a) The OS manages VMW data structures (MT) and copies pages from/to user memory (CT). When finished, it sleeps (ST). The coprocessor executes (HT) until it finishes or misses a page. In both cases, it waits for the OS action. The OS responds after some time (RT). (b) Instead of sleeping, the VMW can fetch, in advance, pages that may be used by the coprocessor. This may result in uninterrupted coprocessor execution.

VI. Dynamic Prefetching

Although it is intuitively expected that the additional layer brings overheads, it is shown here that it can also lower the execution time by taking advantage of runtime information, without any changes in the application and coprocessor code. Since the main processor can be idle during the coprocessor busy time, we explore the scenario where the idle time is invested into anticipating and supporting future coprocessor execution: with simple HW support, the OS can predict coprocessor memory accesses, schedule prefetches, and, thus, decrease memory communication latency.

The sequence of the OS events during a VMW-based coprocessor execution is shown in Fig. 6. Assuming a large spatial locality of coprocessor memory accesses (e.g., stream oriented processing), it can be seen in Fig. 6(a) that the OS module sleeps for a significant amount of time. Once the management is finished, the VMW manager goes to sleep waiting for future coprocessor requests. If the VMW manager were active instead of being idle, it could minimize the number of page faults. Fig. 6(b) shows HW execution time overlapped with the VMW manager that predicts future accesses. Based on information provided by the WMU, it schedules prefetch-based loads of virtual memory pages. For correct predictions, the coprocessor generates no faults: the OS activity completely hides the memory communication latency without any action on the user side.

A. HW and SW 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. Fig. 7 contains the internal organization of the WMU related to address translation. If there is a match in the content addressable memory (CAM), the 1-hot bit lines are used to set the appropriate bit in the access indicator register (AIR). If the VMW manager previously set the corresponding bit in the access monitor register (AMR), the WMU will report the page access event by raising an *access* interrupt—the coprocessor accessed a page for which the OS requested the reporting. While the access interrupt is being handled, there is no need to stop the coprocessor: the VMW manager and the coprocessor run in parallel.

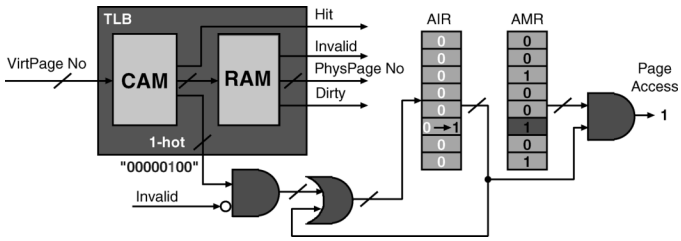

Fig. 7. Page access detection. On a hit in the CAM, 1-hot bit lines will set the corresponding bit in the AIR register. If the mask in the AMR register allows the access, an interrupt is raised.

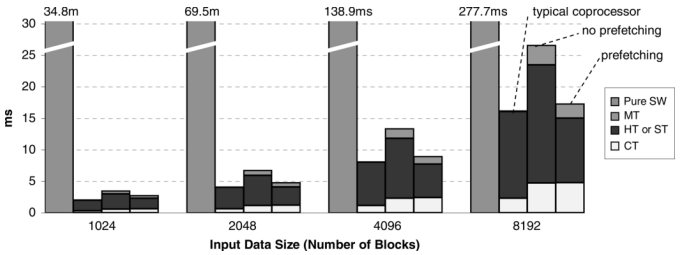

Fig. 8. IDEA execution times.

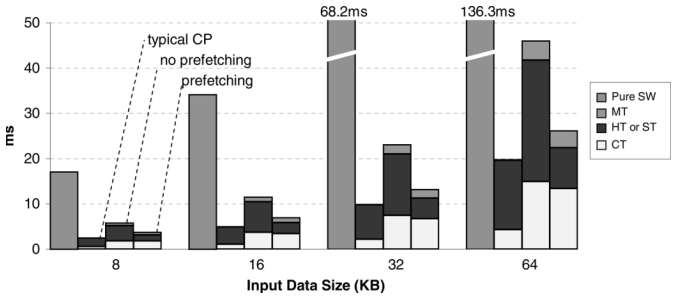

Fig. 9. ADPCM decoder execution times.

We extend the VMW manager [Fig. 4(a)] to support the prediction of future memory accesses and speculative prefetching [Fig. 4(b)]. The three main design components of the extended VMW module [10] are: 1) initialization and interrupt handling; 2) prediction of future accesses (*Predictor*); and 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 prefetching policy [10] motivated by the previous work on stream-buffers for cache memories [11], [12].

VII. Experimental Results

We have implemented two different VMW systems—based on Altera (Excalibur EPXA1 [1], with a 133-MHz ARM processor) and Xilinx (Virtex-II Pro XC2VP30 [2], with a 300-MHz PowerPC processor) reconfigurable devices—with several applications running on them (the IDEA cryptography and the ADPCM voice decoder). 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 coprocessor. Due to the limitations of the FPGA technology, the translation is performed in multiple cycles (4–6 cycles depending on the implementation).

A. Typical Data Sizes

Figs. 8 and 9 show execution times of the IDEA and ADPCM applications (running on the Altera-based board) for typical input data

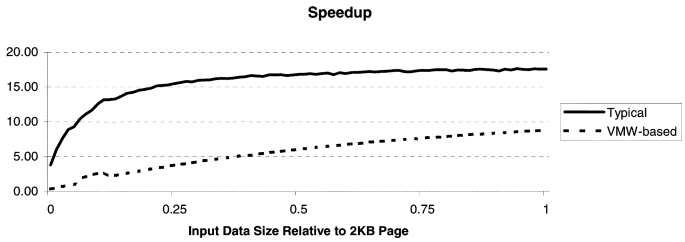

Fig. 10. Speedup (relative to pure SW) for small input data sizes.

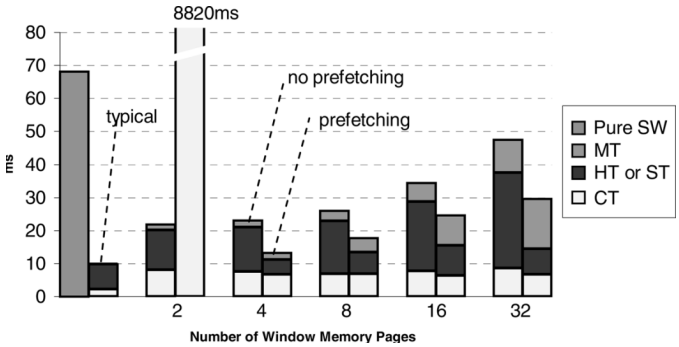

Fig. 11. ADPCM decoder performance for different number of VMW pages.

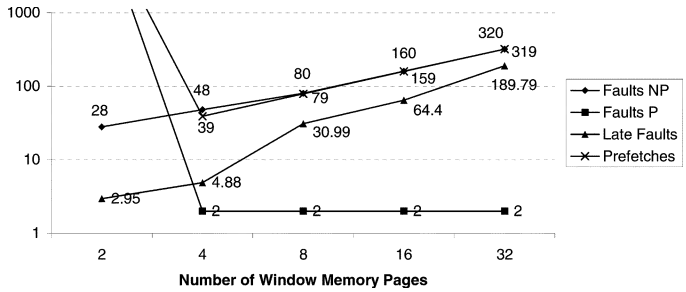

Fig. 12. ADPCM decoder faults with (P) and without (NP) prefetching.

TABLE I
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 (VMW-BASED) IS OCCUPIED BY THE WMU

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

sizes. The VMW-based versions (with and without prefetching) of the applications achieve significant speedup (especially with prefetching) compared to the SW cases, while the overhead is limited compared to the typical coprocessors.

Three components are measured for the typical coprocessor: 1) copy time (CT)—time to copy the exact amount of data for processing; 2) hardware time (HT)—time to perform the computation in hardware; and 3) manage time (MT)—time to update the pointers and iterate until completion. Also, three components are measured for the VMW-based coprocessors (recall Fig. 6). The sleep time consists of the HT (for executing a VMW-based coprocessor) plus the OS response time [not present when programming the typical coprocessor like in Fig. 2(b)].

When compared to the typical coprocessors, sources of the overhead are threefold: 1) costs of managing higher abstraction (represented by MT—goes up to 15% of the total execution time, which is acceptable); 2) current inefficient implementation of the WMU in FPGA (represented by the difference between HW execution times of the typical and VMW-based coprocessors—about 20% and 30% longer execution in the VMW case; 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 MMU); and 3) copy overhead for enforcing memory consistency (represented by the difference between CTs of the typical and VMW-based coprocessors and caused by the page-level memory granularity—discussed in Section VII-B).

Figs. 8 and 9 also show that in the presence of a dynamic optimization like prefetching (described in Section VI), execution times are shortened and the obtained speedup is increased. Prefetching reduces the number of page faults by allowing overlapping of processor and coprocessor execution. As indicated in Fig. 6, the sleep time (ST) decreases: the OS module handles access requests in parallel with the coprocessor execution. Counterintuitively, the management time slightly decreases: the number of fault-originated interrupts is dramatically lower (Fig. 12).

B. 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 window memory pages. Fig. 10 compares speedups (for the Altera-based board) of the typical and VMW-based IDEA coprocessors, in the case of small input data sizes. While the typical coprocessor achieves speedups quite early, the VMW-based solution lags behind: the typical solution has no overhead of copying entire pages into local memory.

C. Different Number of Memory Pages

Having multiple WMU operation modes allows the VMW to fit coprocessors with different memory access patterns. Except for some extreme values, changing the WMU operating modes does not influence performance dramatically (as Fig. 11 shows for the ADPCM application running on the Altera-based board).

Increasing the number of pages (i.e., the fixed-size window memory is divided into smaller pages) for the same input data size increases the number of faults (faults NP in Fig. 12). However, prefetching in the VMW keeps the number of faults (faults P) low and constant (except for only two VMW pages, when memory trashing appears—also visible in Fig. 11). Having smaller page sizes (i.e., window memory contains more pages), MT and CT intervals become comparable to the HW execution intervals: late faults—lest 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).

D. Area Overhead

Table I 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 interface memory (recall Fig. 3): this memory is necessary even without WMU. Although the 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.

E. Results Summary

Fig. 13 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

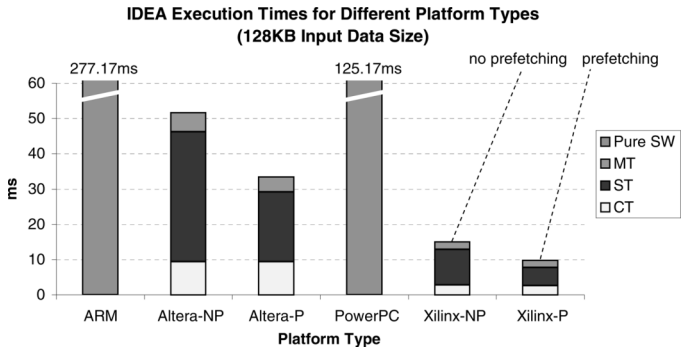

Fig. 13. Two different platforms (Altera Excalibur-based and Xilinx Virtex-II Pro-based) run the same application. Results are shown for pure SW (ARM, PowerPC) and VMW-based coprocessors, with no prefetching (Altera-NP, Xilinx-NP) and with prefetching (Altera-P, Xilinx-P) in the VMW.

intention is not to compare the platforms, but to emphasize that running the experiments on a different platform implies only porting the WMU HW and the VMW SW, and does not require any changes to the coprocessor HDL nor to the application C code. VMW-based applications can achieve significant performance advantage over the pure software, in spite of the introduced virtualization. The overhead can be reduced, especially with the address translation done in VLSI (as it is done for MMUs). Dynamic optimizations can provide additional speedups with no change on the application side.

VIII. Conclusion

We have proposed a unified memory abstraction for SW and HW parts of reconfigurable applications. Our VMW: 1) provides a straightforward programming paradigm (programmers are completely screened from interfacing-related memory transfers); 2) makes reconfigurable applications completely portable (recompiling and resynthesising is sufficient); and 3) enables advanced and yet simple runtime optimizations (without any change in either application SW or coprocessor HW). By testing our approach on real systems, we have shown that the price to pay is affordable. After reducing the overhead of the virtualization layer, future research should address runtime optimizations for exposing the hardware speedup to its maximal extent.

References

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

[2] Xilinx Virtex ProII Devices, Xilinx Inc., 2003. [Online]. Available: http://www.xilinx.com/

[3] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, 2003, pp. 290–295.

[4] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwow, M. Wong, and K. Lee, "Pilchard—A reconfigurable computing platform with memory slot interface," in *Proc. 9th IEEE Symp. Field-Program. Custom Comput. Machines*, 2001, pp. 170–179.

[5] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proc. Int. Parallel Distrib. Process. Symp., Reconfigurable Arch. Workshop (RAW)*, 2003, pp. 22–26.

[6] C. Sullivan and M. Saini, "Software-compiled system design optimizes Xilinx programmable systems," *Xcell J.*, no. 46, pp. 32–37, 2003.

[7] M. Vuletić, L. Pozzi, and P. Ienne, "Seamless hardware-software integration in reconfigurable computing systems," *IEEE Des. Test Comput.*, vol. 22, no. 2, pp. 102–13, Mar.–Apr. 2005.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Mateo, CA: Morgan Kaufmann, 2002.

[9] M. Vuletić, L. Pozzi, and P. Ienne, "Virtual memory window for a portable reconfigurable cryptography coprocessor," in *Proc. 12th IEEE Symp. Field-Program. Custom Comput. Machines*, 2004, pp. 24–33.

[10] ——, "Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors," in *Proc. 14th Int. Conf. Field-Program. Logic Appl.*, 2004, pp. 596–605.

[11] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Comput. Arch.*, 1990, pp. 364–73.

[12] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proc. 21st Annu. Int. Symp. Comput. Arch.*, 1994, pp. 24–33.

New Degree Computationless Modified Euclid Algorithm and Architecture for Reed-Solomon Decoder

Jae H. Baek and Myung H. Sunwoo

***Abstract*—This paper proposes a new degree computationless modified Euclid (DCME) algorithm and its dedicated architecture for Reed–Solomon (RS) decoder. This architecture has low hardware complexity compared with conventional modified Euclid (ME) architectures, since it can completely remove the degree computation and comparison circuits. The architecture employing a systolic array requires only the latency of $2t$ clock cycles to solve the key equation without initial latency. In addition, the DCME architecture using $3t + 2$ basic cells has regularity and scalability since it uses only one processing element. Hence, the proposed DCME architecture provides the short latency and low-cost RS decoding. The DCME architecture has been synthesized using the 0.25-μm Faraday CMOS standard cell library and operates at 200 MHz. The gate count of the DCME architecture is 21 760. Hence, the RS decoder using the proposed DCME architecture can reduce the total gate count by at least 23% and the total latency to at least 10% compared with conventional ME decoders.**

***Index Terms*—Degree computation circuit, forward error control, low hardware complexity, Reed–Solomon (RS) codes, short latency, systolic array, VLSI design.**

I. Introduction

Error control codes are widely used in communication systems to protect the transmitted data from errors. Error control codes are classified into convolutional and block codes. Reed–Solomon (RS) codes are linear block codes and belong to the class of nonbinary Bose–Chaudhuri–Hocquenheim (BCH) codes [1]. RS codes providing the capability to efficiently correct burst errors, as well as random errors, have been extensively used in various communications and digital data storage systems, such as Power Line Communications (PLC) [2], Digital Video Broadcasting Terrestrial (DVB-T) system [3], Vestigial Sideband (VSB) system [4], cable modem [5], satellite and mobile communications [6], magnetic recording [7], etc.

The general (n, k, t) RS code defined in the Galois field (GF) has a code length $n = 2^m - 1$, where k denotes the number of m-bit message symbols. The RS code has the error correction capability

Manuscript received January 31, 2003; revised July 5, 2003, February 17, 2005, and March 8, 2006. This work was supported in part by the National Research Laboratory (NRL) program, by the Post Brain Korea 21 (BK 21) program, and by IC Design Education Center (IDEC).

The authors are with the School of Electrical and Computer Engineering, Ajou University, Suwon 443-794, Korea (e-mail: sunwoo@ajou.ac.kr).

Digital Object Identifier 10.1109/TVLSI.2006.878484