Dynamic Prefetching in the Virtual Memory Window of Portable Reconfigurable Coprocessors

Miljan Vuletić, Laura Pozzi, and Paolo Ienne

EPFL I&C LAP
Swiss Federal Institute of Technology Lausanne
1015 Lausanne, Switzerland
{miljan.vuletic, laura.pozzi, paolo.ienne}@epfl.ch
http://lap.epfl.ch

Abstract. In *Reconfigurable Systems-On-Chip (RSoCs)*, operating systems can primarily (1) manage the sharing of limited reconfigurable resources, and (2) support communication between reconfigurable accelerators and user applications. It has been shown in previous work that the operating system can dramatically simplify the interface to reconfigurable coprocessors and isolate the programmer from all details of the hardware. A further potential of the operating system is developed here: the operating system can observe accelerators at runtime and dynamically take actions which improve their execution. The strength of involving the operating system consists in achieving better performance without any information from the end user and without changes either in the coprocessor hardware design or in the software application. Specifically, this paper presents an operating system module that monitors reconfigurable coprocessors, predicts their future memory accesses, and performs memory prefetching accordingly; the goal is to hide completely memory-to-memory communication latency. The module uses a lightweight hardware support to detect coprocessors memory access patterns. The effectiveness of the technique is demonstrated for two applications on an embedded RSoC board running the Linux operating system. Significant speedup is achieved compared to the nonprefetching version, and the improvement is obtained in a manner completely transparent to the application programmer.

1 Introduction

Reconfigurable accelerators, running on behalf of user applications, exploit the potentials of spatial computation in reconfigurable logic. It is a natural task for the *Operating System (OS)* to control the reconfigurable logic and facilitate its use. Reconfigurable resources can be shared, physically and virtually partitioned between applications [3,4,14]. By supporting the virtual memory address space sharing between an application and its coprocessor [12], the OS can enable a transparent way of interfacing: it hides the actual interface and automatically copies data from user memory to the coprocessesor memory and back.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 596–605, 2004.

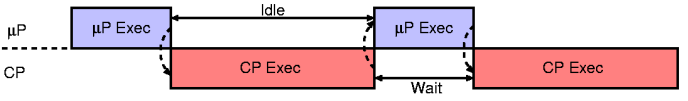

Fig. 1. Simple execution. The processor and coprocessor change in turn.

The OS is not only limited to providing resource sharing and transparent interfacing: it can survey the execution of the coprocessor, optimise communication, and even adapt the interface dynamically. A virtualisation layer makes such improvements possible without any changes in the application and coprocessor code. Although it is intuitively expected that the additional layer brings overheads, it is shown here that it can also lower execution time by taking advantage of run-time information. In this paper, the strength of delegating the interfacing tasks to the OS is presented. As opposed to the simple execution model shown in Figure 1, where the main processor is 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 hardware support, the OS can predict coprocessor memory accesses, schedule prefetches, and thus decrease memory communication latency.

This paper is organised as follows: In Section 2, the basic concepts of *Virtual Memory Window (VMW)* for reconfigurable coprocessors are presented. Section 3 discusses related work. The OS memory prefetching concept, its hardware and software design elements are presented in Section 4. Section 5 shows the measurements that prove the benefits of prefetching. Finally, some conclusions are given in Section 6.

2 Virtual Memory Window

Nonstandard programming paradigms and HW/SW interfacing models have certainly hindered the acceptance of reconfigurable computing. The *Virtual Memory Window (VMW)* addresses these problems [12] by reusing the simple and well-known concept of virtual memory. The VMW enables the coprocessors to share the virtual memory address space with user applications, thus simplifying the programming paradigm and hardware interfacing.

Figure 2 shows how a reconfigurable coprocessor is interfaced with the main processor. The OS provides a uniform and abstract virtual memory image hiding all details about the physical memory. The fast translation from virtual to physical addresses is enabled by hardware accelerators: (1) *Memory Management Unit (MMU)* in the main processor case, and (2) *Window Management Unit (WMU)* in the coprocessor case. The *Virtual Memory Manager (VMM)* and *Virtual Memory Window (VMW) Manager* in the OS ensure that the translation is transparent to the end users. In the same manner as the VMM copies pages between the mass storage and the main memory, the VMW manager copies pages between the main memory and the window memory. Both managers do the tasks *transparently* from the end user.

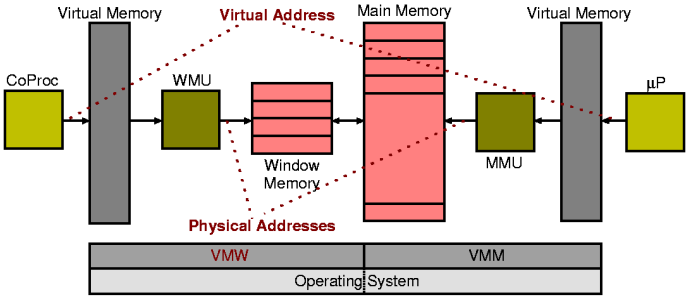

Fig. 2. Virtual Memory Window for reconfigurable coprocessor. The coprocessor and user application share the virtual memory address space. The Window Management Unit supports the address translation, which is managed by the OS.

Benefits of unifying the memory pictures from the main processor and the coprocessor side are: (1) programming software and designing hardware is made simpler—calling a coprocessor from a user application is as simple as a common function call, and designing the coprocessor hardware imposes no memory constraints but only requires complying to the WMU interface; (2) application software and accelerator hardware are made portable—hiding platform-related details behind the VMW manager and the WMU deliberates applications and coprocessor designs of platform dependence.

3 Related Work

Different approaches for virtualisation of reconfigurable resources are proposed [3,4]. The tasks of management and sharing the resources are delegated to the OS [7,14]. An orthogonal approach [12] that involves the OS to support interface virtualisation is used as the basis of the work presented in this paper.

Hardware and software prefetching techniques were originally developed for cache memories to support different memory access patterns. Stream buffers [5] were introduced (and later enhanced [8]) as an extension of tagged-based prefetching to improve sequential memory accesses. Other techniques exist that cover nonsequential memory accesses (e.g., recursive [9], and correlation-based [11] where a user-level thread correlation prefetching is shown). Hardware prefetching techniques have also been used for configurable processors [6]. Besides caching, prefetching techniques have been used for efficient virtual memory management: in hardware (e.g., speculatively preloading the TLB to avoid page faults [10]) and in software (prefetching virtual memory pages for user application [2]).

The technique presented in this paper is in its essence a dynamic software technique with limited hardware support. Its strongest point is the transparency: neither user applications nor hardware accelerators are aware of its presence.

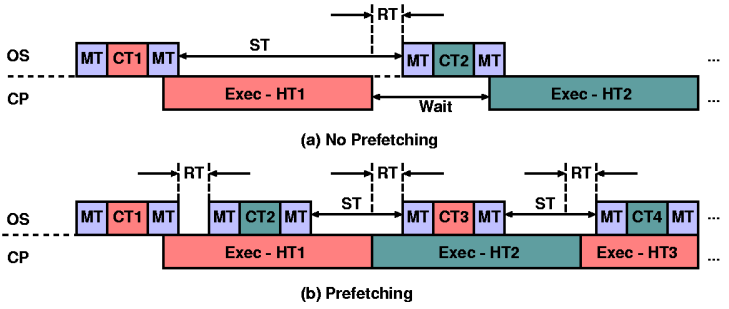

Fig. 3. The OS module (OS) activities related to coprocessor execution (CP). The OS manages VMW data structures (Management Time—MT) and copies pages from/to user memory (Copy Time—CT). When finished, it sleeps (Sleep Time—ST). The coprocessor executes (Hardware Time—HT) until it finishes or misses a page. In both cases, it waits for the OS action. The OS responds after some time (Reponse Time—RT). Instead of sleeping, the VMW can fetch in advance pages that may be used by the coprocessor. This results in uninterrupted coprocessor execution.

Even more importantly, an OS module is developed to optimise execution of coprocessors, and it is not limited to the presented prefetching technique.

4 OS-Based Prefetching

In this section, the basic motivation for applying OS-based prefetch techniques is presented. Afterwards, hardware and software requirements to implement a prefetching system for a VMW are discussed in detail.

4.1 Memory Copy Overhead

The sequence of the OS events during a VMW-based coprocessor execution is shown in Figure 3. Assuming a large spatial locality of coprocessor memory accesses (e.g., stream oriented processing), it can be seen in Figure 3a that the OS sleeps for a significant amount of time. Once the management is finished, the manager goes to sleep waiting for future coprocessor requests.

During idle time, the VMW manager could instead survey the execution of the coprocessor and anticipate its future requests, thus minimising the number of page misses. Figure 3b shows hardware execution time overlapped with the VMW management activities. During coprocessor operation, the WMU informs the manager about the pages accessed by the coprocessor. Based on this information, the manager can predict future activities of the coprocessor and schedule prefetch-based loads of virtual memory pages. If the prediction is correct, the coprocessor can use the prefetched pages without generating miss interrupts. In this way, the involvement of the operating system may completely hide the memory communication latency. *The approach requires no action on the software programmer nor on the hardware designer side.*

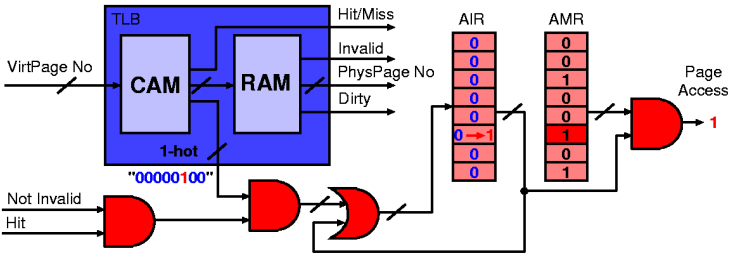

Fig. 4. Page access detection. On a hit in the *Content Addressable Memory (CAM)*, 1-hot bit lines will set the corresponding bit in the *Access Indicator Register (AIR)*. If the mask (*Access Mask Register (AMR)* allows the access, an interrupt is raised.

4.2 Hardware Support

The WMU provides hardware support for the translation of the coprocessor virtual addresses and for accessing the window memory. The window memory is divided into pages that map onto the different regions of the user memory. The optimal number of pages depends on the characteristics of the coprocessor memory access pattern. The WMU supports multiple operation modes—i.e., different page sizes and number of pages. A simple extension—two 32-bit registers and few tens of logic gates—to the WMU is introduced that supports the detection of a page access. Figure 4 contains the internal organisation of the WMU related to address translation. As in typical MMUs, address mapping is performed by a *Translation Lookaside Buffer (TLB)*. 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 OS wants to detect the first access to a particular page, it simply sets the correct mask in the *Access Mask Register (AMR)*. When the access appears, an interrupt is raised requesting OS handling. Nested interrupts are prevented by the OS resetting to 0 the appropriate mask bit. While the interrupt is being handled, there is no need to stop the coprocessor: interrupt handling and coprocessor run in parallel—space is left for speculative work. 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.

4.3 VMW Module

The three main design components of the VMW module are: (1) initialisation and interrupt handling, (2) prediction of future accesses, and (3) fetching of pages from main memory.

Interrupt Handling. Once invoked, the OS service first initialises the internal data structures and the WMU hardware. It then goes to sleep awaiting for interrupts. There are three possible interrupt types coming from the WMU: (1)

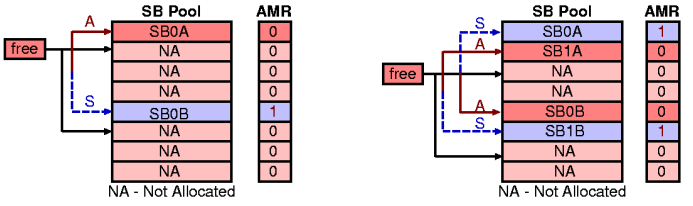

Fig. 5. Stream Buffer (SB) allocation and the AMR. On a miss, a pair of pages (SB) is allocated. The miss page is active (A); it is being accessed by the coprocessor; the prefetched page is speculative (S). The AMR helps to detect accesses to (S) pages.

finish, (2) miss, and (3) access. When *finished*, the module returns control back to the user application. If a *miss* appears, the load of the miss page is scheduled into the fetch queue. Afterward, the predictor is called to attempt predicting future page accesses and speculative pages are also scheduled for loading. The coprocessor is resumed by the fetcher, once all miss-related requests are satisfied. If an *access* appears, it indicates that the coprocessor accessed a page for which this information had been requested. The predictor is called to validate or confute its past predictions and schedule future page loads into the fetch queue. During access handling, the coprocessor is active.

The Predictor. It attempts to guess future memory accesses and to schedule page loading. The only input parameters to the predictor are miss addresses and access page numbers—i.e., there is no information about the state of the coprocessor. The approach is similar to classic prefetching techniques where no information is available about the instructions issued by the main processor [8] but only the addresses on the bus. The current predictor assumes that for each miss a new stream is detected; thus, it requires a stream buffer allocation (i.e., a pair of window memory pages, refer to Figure 5) and it schedules a speculative prefetch for the page following the missing one. By setting appropriately the AMR, it ensures that the WMU hardware will report the first access to the speculatively-loaded page. When the access is reported, the predictor is invoked again and, with this information confirming the correct speculation, further prefetches are scheduled. Each speculative prefetch is designated to its corresponding stream buffer. Ideally, for a correctly-guessed memory access stream and good timing of the prefetching, only one miss per stream should appear: all others misses should be avoided through prefetching.

Since the number of stream buffers is limited, the coprocessor may require more streams than it can be provided. In this case, a stream buffer should be selected for deallocation, to satisfy a new allocation request. For the moment, a simple eviction policy is implemented; yet, since the predictor is a software-only component, more sophisticated eviction policies can be easily added. Furthermore, potential trashing and deadlocks (due to the capacity problems of the window memory) can be resolved *dynamically and transparently for the end-user* simply by changing the operation mode of the WMU.

The Fetcher. The fetcher is responsible for loading pages from/to user space memory. The memory requests are scheduled by the miss handler and by the predictor, with miss-generated requests being always scheduled before speculative ones. The fetcher executes the fetch queue, until all the requests are serviced. It determines the type of fetch (mandatory or speculative), its destination in the window memory, and whether it requires a stream buffer allocation. If the destination is occupied by a dirty page, it is copied back to the user space. The page is then fetched from user memory and the request is deleted from the queue. The coprocessor can be resumed if needed—if the fulfilled request is miss-based and there are no outstanding miss-based requests.

5 Experiments

The system described is implemented on an Altera Excalibur based board with the EPXA1 device [1]. The device consists of an ARM processor with basic peripherals and a reconfigurable part. The ARM processor is running on 133MHz and executes user applications under the GNU/Linux OS. The WMU with access indication support, is synthesised from VHDL code (less than thousand lines) to reconfigurable logic and interfaced to the ARM processor using the Avalon bus [1]. A dual-ported 16KB on-chip memory is used for the VMW window memory. The VMW manager with prefetching support is implemented as a loadable kernel module (in a couple of thousands lines of C-code).

5.1 Results and Comparisons

Two applications are ported to the system: (1) IDEA cryptography application and (2) ADPCM decoder from MediaBench. For both applications, coprocessors have been designed (IDEA running at 6MHz and ADPCM running at 40MHz) complying to the WMU interface and implementing critical parts of the algorithms in hardware. Even without OS controlled prefetching, both applications achieve significant speed up compared to their software-only versions [12]. Notice that no change whatsoever has been made to the user C and VHDL code to take advantage of prefetching—the code is *exactly the same* that was developed in previous work [13,12], and only the WMU and the VMW manager differ.

Figure 6 compares total execution times of ADPCM decoder with and without prefetching in the VMW module. Although running at the same speed, in the prefetching case the coprocessor finishes its task almost twice as fast compared to the nonprefetching case. As indicated in Figure 3, the sleep time reduces: the module handles access requests in parallel with the execution of the coprocessor. Counterintuitively, the management time slightly decreases, because the number of miss-originated interrupts is dramatically lower (e.g., in the case of 32KB input data size it goes down from 48 to only 2). Meanwhile, multiple access-originated interrupts may appear within a relatively short time interval (e.g., two streams usually cross the page boundary at about the same time) and

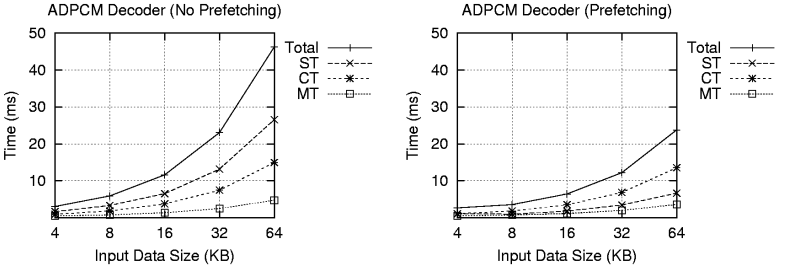

Fig. 6. ADPCM decoder: Total execution times with/without prefetching. The execution time consists of sleep time (ST), copy time (CT), and manage time (MT).

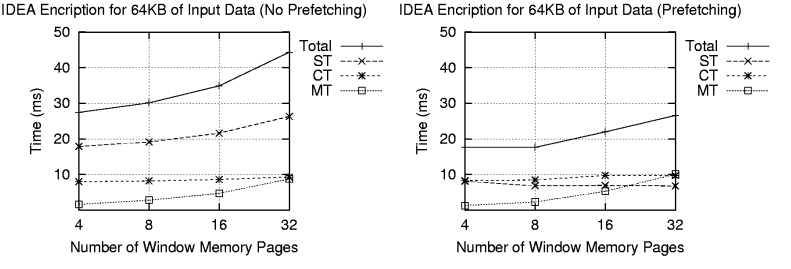

Fig. 7. IDEA encryption: total execution times with/without prefetching.

the VMW manger services them at the same cost. This is not the case for the misses: for a miss to appear, the previous miss needs to be already serviced.

The ADPCM decoder has a specific access pattern: the decoder is producing four times more data than it consumes. Due to the simple FIFO policy used for page eviction in the non-prefetching case, it may happen that a page still being used gets evicted: the page will need to be reread before the coprocessor continues execution. On the other hand, the prefetching approach with stream-buffer allocation is less sensitive to the applied page eviction policy because distinct stream-buffers are allocated for input and output streams.

Figure 7 shows the total execution times of IDEA encryption for different number of window memory pages. A significant improvement in the IDEA execution time is achieved with prefetching. Management time increases with the increasing number of window memory page, since larger data structures are managed. In the prefetching case, the management time is slightly larger than without prefetching. With smaller page sizes, manage and copy time intervals become comparable to the hardware execution intervals: increasingly often, the coprocessor generates a miss while the missing page is already being prefetched. This miss is called a late miss, and it is less costly than a regular one. Still, the VMW manager needs to acknowledge it once its corresponding prefetch is finished—hence the slight increase in the management time. Table 1 shows

Table 1. Interrupts by the IDEA coprocessor. In the prefetching case, beside misses, there are accesses (used to trigger prefetching) and late misses (when the missing page is already being transferred).

Number of pages	Page size	Nonprefetching Misses	Prefetching Misses	Late Misses	Accesses
4	4KB	32	2	0	15
8	2KB	64	2	2	31
16	1KB	128	2	24	63
32	0.5KB	256	3	53	131

how the number of miss-originated and access-originated interrupts grows with smaller page sizes. It also shows how late misses start to appear.

Although it seems costly to manage larger number of window memory pages, in some cases the flexibility of the WMU and the VMW manager may be required, since the WMU operation mode can affect the performance. For example, supposing only two window memory pages, and prefetching, the coprocessor experiences memory trashing problems and performs dramatically slower then the non-prefetching one (e.g., for the IDEA encryption, on 16KB input data and two 8KB pages in the window memory, 757ms vs. 7ms, and 1366 vs. 6 misses!). It is the task of the VMW module to detect this misbehaviour and change to the operation mode that corresponds better to the coprocessor needs.

6 Conclusion

This paper presented an OS module supporting the execution of reconfigurable coprocessors running within the VMW framework. Not only it allows the coprocessors to share transparently the same address space with user applications, but it also makes possible advanced and yet simple runtime optimisations, *without any intervention by the end user*.

In order to demonstrate the presented concept, a stream-based memory prefetch technique was implemented within the OS module (with a simple hardware support in the WMU). A significant execution time improvement is demonstrated for two application-specific reconfigurable coprocessors, *without any change in either application software or coprocessor hardware*.

Future extensions of this work are not limited to implementing other prefetch techniques (e.g., recursive and correlation-based prefetching): the involvement of the OS enables novel runtime optimisations (e.g., changing the number and size of window memory pages in order to fit better application needs).

Acknowledgments. The authors would like to acknowledge and appreciate help and support provided by Nicolas Blanc and Cédric Gaudin.

References

1. Altera Corporation. *Altera Excalibur Devices*, 2003. `http://www.altera.com/`.
2. K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *In the Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI, 1994)*, pages 243–53, Monterey, Calif., Nov. 1994. USENIX Assoc.
3. E. Caspi, M. Chu, R. Huang, J. Yeh, A. DeHon, and J. Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *Proceedings of the 10th International on Field-Programmable Logic and Applications*, pages 605–14, Villach, Austria, Aug. 2000.
4. M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10980–85, Munich, Mar. 2003.
5. N. 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.
6. H. Lange and A. Koch. Memory access schemes for configurable processors. In *Proceedings of the 10th International on Field-Programmable Logic and Applications*, pages 615–25, Villach, Austria, Aug. 2000.
7. V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, Apr. 2003.
8. S. Palacharla and R. 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., Apr. 1994.
9. A. Roth, A. Moshovos, and G. 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., Oct. 1998.
10. A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB preloading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 117–27, Vancouver, British Columbia, Canada, June 2000.
11. Y. Solihin, J. Lee, and J. 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.
12. M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41st Design Automation Conference*, pages 948–53, San Diego, Calif., June 2004.
13. M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for portable reconfigurable cryptography coprocessor. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 2004.
14. H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 290–95, Munich, Mar. 2003.