2011 21st International Conference on Field Programmable Logic and Applications

Memory Virtualization for Multithreaded Reconfigurable Hardware

Andreas Agne, Marco Platzner
Computer Engineering Group
University of Paderborn
Paderborn, Germany
e-mail: {agne, platzner}@uni-paderborn.de

Enno Lübbers
Innovation Works
EADS Deutschland GmbH
Munich, Germany
e-mail: enno.luebbers@eads.net

***Abstract*—With the introduction of multithreaded programming for reconfigurable hardware, it is possible to map both sequential software and parallel hardware to a single CPU/FPGA platform using threads as a unifying development model. At the same time, platform FPGAs are a natural technology for implementing computationally intensive systems in the aerospace, automotive and industrial domains, as they combine high performance and flexibility with lower non-recurring engineering (NRE) costs when compared to low-volume ASIC solutions. The reusability and portability of hardware components in these safety-critical domains could be significantly improved by using multithreaded programming. However, the unique design considerations for memory virtualization, as required in safety-critical systems, are difficult to transfer directly from software to autonomous hardware threads.**

This paper presents a transparent and efficient way of augmenting current multithreaded and partially reconfigurable hardware runtime environments with dedicated, hardware-thread-aware memory address translation units to provide seamless memory translation for hardware threads. We show an analysis of the overheads, as well as an experimental evaluation of the latencies caused by address translation.

***Keywords*-multithreaded hardware; virtualization; safety-critical systems; FPGAs**

I. Introduction

Recent years have seen the evolution of FPGAs from low-density prototyping platforms to highly complex million-gate-equivalent programmable hardware platforms, which has significantly broadened the application areas of these reconfigurable devices. On one hand, this development has given rise to a number of intriguing devices such as platform FPGAs [1], [2] and Extensible Processing Platforms [3], which allow the efficient integration of control-dominated sequential software and high-performance data-parallel hardware on a single chip. On the other hand, the class of hybrid hardware/software systems opens up new questions as to what programming models would be suitable for a CPU/FPGA platform, as the predominant paradigms for (logic-centric parallel) hardware and (sequential) software differ significantly. A promising approach is that of multithreaded hardware, where both software threads, running on a CPU, and hardware threads, mapped to an FPGA's fabric, use a common set of operating system services [4], [5], [6]. Multithreaded hardware has been successfully adopted in a number of different application areas, such as adaptive networking [7], distributed systems [8], and video tracking [9].

In low-volume safety-critical applications such as space transportation systems, aircraft electronics, or avionic test systems, CPU/FPGA platforms are of particular interest, as they provide high-performance, flexibility, fault-tolerance through reprogrammability (i.e., self-healing or adaptive redundancy), and in-field upgradeability while avoiding the NRE overheads of traditional ASICs. However, safety issues currently prevent the widespread application of CPU/FPGA systems in these areas, as methods for the functional separation of HW and SW components with different levels of criticality (which is a prerequisite for any certifiable embedded system) are still missing.

This paper presents a novel concept for providing transparent memory virtualization support for hardware threads in hybrid CPU/FPGA systems. By providing thread-aware translation-lookaside buffers (TLBs) for the reconfigurable slots in which hardware threads are executed, our approach enables multithreaded hardware to seamlessly operate on virtual addresses. The proposed system provides data integrity of separated processes, access control to memory segments shared between hardware and software, as well as a transparent extension of the multithreaded hardware approach to virtual memory environments, and takes an important step towards the adoption of multithreaded reconfigurable hardware in future safety-critical systems.

II. Related work

Multithreaded programming of reconfigurable hardware is an emerging research field with recent work from a number of different directions. The idea of managing programmable hardware modules using an operating system was first proposed by Brebner [10]. Later, Steiger et al. [11] introduced the notion of hardware tasks as independent units with access to OS resources like FIFOs or memory blocks. These ideas were successively refined and have by now been implemented in a number of development and execution environments such as ReconOS [4] (which will be extended in this work) and hthreads [5].

Although these systems provide transparent access to the main memory, they are still lacking in support for virtual address translation.

An approach to provide a common address space for hard- and software threads is discussed by Vuletic et al. [12], who propose Virtual Memory Windows to

978-0-7695-4529-5/11 $26.00 © 2011 IEEE
DOI 10.1109/FPL.2011.42

allow hardware threads to access a dedicated memory area using virtual addresses. Transaction requests to addresses outside of this window cause the Virtual Memory Window Manager running on the CPU to copy the missing page into a local buffer directly accessible by the hardware thread. While this system enables the hardware to connect to virtual memory, random page accesses across page boundaries are not efficiently implemented, as they involve copying entire pages between system and local memory, even if only a single byte is to be changed.

The idea of augmenting hardware threads with a dedicated address translation unit has been investigated by Garcia and Compton [13]. Similar to the approach presented in this paper, they add an MMU with a 16-entry TLB managed by system software to each hardware thread. A TLB miss interrupts the CPU, which resolves the page fault and adds the missing entry to the hardware thread's TLB. While this approach more transparently integrates the hardware threads into the system, random memory accesses still cause considerable delays due to the software-based resolution of TLB misses.

This deficiency is addressed by the system proposed by Lange und Koch [14], where the implemented TLBs are able to read translation entries from the page tables of the operating system without any software intervention. Page faults (i.e., requests for pages not present in the process's page tables) are again delegated to software. This efficient approach allows hardware modules to access the entire virtual memory subsystem, and is also adopted by our implementation. In extension to their work, we focus on integrating virtual memory management with a unified programming model to combine the transparency of multithreaded reconfigurable hardware development with efficient memory management. In this way, we close a crucial gap between hybrid CPU/FPGA platforms and the domain of safety-critical applications.

III. Multithreaded Programming of Reconfigurable Hardware

In software-centric systems, multithreaded programming is already a popular means to decompose an application into individual threads of execution. The operating system ReconOS [4] extends the multithreaded programming model to the domain of reconfigurable hardware. Instead of regarding hardware modules as passive coprocessors to the system CPU, they are treated as independent *hardware threads* on an equal footing with the software threads running in the system. In particular, ReconOS allows hardware threads to use the same operating system services for communication and synchronization as software threads, providing a transparent programming model across the hardware/software boundary. The structured modularization of components, together with a familiar API as a common abstraction layer for hard- and software, simplifies design space exploration and incremental design, and streamlines modular development for hybrid CPU/FPGA systems.

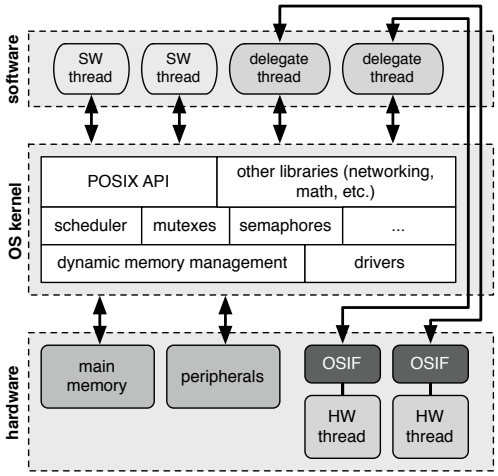

Figure 1. ReconOS hardware architecture.

ReconOS builds on top of and extends existing operating system kernels, such as eCos or Linux, and is targeted specifically at platform FPGAs integrating microprocessors and reconfigurable logic. Figure 1 shows the architecture of a typical ReconOS system. The CPU executes the operating system kernel and manages all interactions between the individual threads of the application. In addition, it can run regular software threads that use the kernel's software API. The reconfigurable area is divided into multiple slots. These slots are connected to the system through a dedicated hardware OS interface (OSIF). The OSIF also manages the low-level synchronization and includes the logic necessary for partial run-time reconfiguration. Hardware threads running in the reconfigurable slots route all their operating system interaction requests through their OSIF, which forwards the requests to a corresponding *delegate thread* running on the CPU. This delegate thread performs all OS interaction on behalf of the hardware thread. Hence, the operating system can transparently handle, control, and supervise both hardware and software threads running on a mix of heterogeneous processing elements.

IV. Memory Virtualization for Hardware Threads

In order to enable address space separation and memory access control for hardware threads, we use the concept of a Memory Management Unit, as it is widely used in the domain of general purpose processors. For software-based systems, the MMU is typically closely integrated into the CPU's caching and bus logic. In ReconOS, hardware threads are connected to the memory bus through the Operating System Interface. By integrating the hardware thread MMU (HWT-MMU) into the OSIF, we are able to make use of the already existing communication link between OSIF and the software operating system so that MMU exceptions can be handled by the delegate threads,

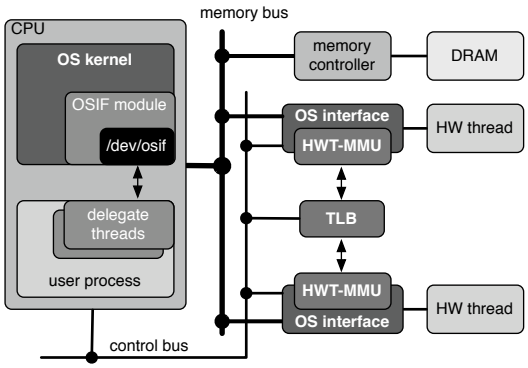

Figure 2. Multiple HWTs connected to a shared TLB.

as it is the case for many other operating system services provided by ReconOS.

While address translation can be delegated to software, where it can be handled transparently by the operating system, this incurs a large overhead, mainly due to interrupt processing. Instead we chose to supply the HWT-MMU with a state machine that is able to access the operating system's page tables directly. This way, page table entries can be accessed autonomously by each HWT-MMU without interfering with the CPU's processing, which significantly reduces the overhead associated with TLB misses. Status registers that contain statistics on TLB hits, misses and page faults are attached to the control bus and can be read by software. An overview of the topology of the system is given in Figure 2.

While the HWT-MMU is designed to provide address translation for the HWTs autonomously, exceptions may occur that require further processing in software. There are two possible causes for HWT-MMU exceptions: (1) An access violation exception is generated when the mode of memory access is not permitted, as for instance a write access to read-only memory. Apart from programming errors, a common cause for this is the operating system keeping track of pages that have been written to, by initially marking them read-only. A write access will then trigger an MMU exception and the page can be marked 'dirty'. (2) A page fault exception may occur when there is no valid page table entry for the page being accessed. While this may be caused by an error in the program resulting in an access to a region in the address space that has not been allocated, page faults may also be generated by the operating system employing demand paging - a strategy that defers the creation of page table entries to the point when the pages are accessed for the first time.

In both cases, the execution of the hardware thread is stopped and the associated delegate thread is notified. The delegate thread will repeat the memory access that triggered the exception. The operating system will then either remove the condition that caused the exception (i.e. marking the page writable, or creating the page table entry) or - in the case of a programming error - terminate the process. In a safety-critical application the termination of the process gives the system software the opportunity to enter a safe state and take further measures, such as resetting the system or switching control to a backup system. If no error occurs, any cached page tables are flushed to system memory, where they can be accessed by the HWT-MMU and the hardware thread is resumed.

In order to stay consistent with the system page tables, the HWT-TLB is invalidated and the CPU data cache is flushed to system memory whenever the operating system changes or removes existing page table entries. We achieve this by modifying the Linux kernel's Cache Flush Architecture [15].

V. Analysis and Experimental Results

In order to assess the overhead of address translation we have analyzed the MMU state machine, and implemented and benchmarked the system on a modern platform FPGA.

Through an analysis of the MMU state machine, we can determine the platform independent address translation overheads. While a TLB hit only incurs 4 cycles overhead to the memory access, a TLB miss requires 11 cycles plus 2 read operations. This compares well to the PowerPC-MMU which handles TLB hits within 1 to 4 cycles (depending on the location of the matching entry within the TLB) and requires an additional context-switch in the case of a TLB miss [16]. A page fault or an access violation encountered in the HWT-MMU cause additional overheads through exception handling on the main CPU.

In order to determine the relative overheads of address translation in practice, we implemented the system on a Xilinx XC2VP30-FPGA with an embedded PowerPC 405 processor running at a clock frequency of 300MHz. The single hardware thread used for the benchmarks as well as the memory bus were clocked at 100MHz. We performed latency measurements of single word and 32-word burst accesses with address translation involving TLB hits, misses and page faults. For the TLB hit/miss values the average of 102.4 million individual measurements were taken. Page fault values were averaged over 1024 single measurements. The overhead of address translation was then calculated as a percentage of total memory access latency. The results obtained are presented in Table I.

The performance overhead caused by address translation for two common use cases can be derived from the measurements: The additional latency incurred on random

Table I
Memory access latencies for different access modes and address translation paths

	translation path		
access mode	TLB hit	TLB miss	page fault
single read	$0.50\mu s$ (+8.7%)	$1.22\mu s$ (+165%)	$1314\mu s$
single write	$0.27\mu s$ (+17.4%)	$1.09\mu s$ (+374%)	$1293\mu s$
burst read	$0.89\mu s$ (+4.7%)	$1.59\mu s$ (+87%)	$1324\mu s$
burst write	$0.63\mu s$ (+6.8%)	$1.35\mu s$ (+129%)	$1301\mu s$

For TLB hits and misses the calculated relative overhead of the address translation is given in percent.

access patterns on large memory regions where the TLB offers no advantage can be taken from Table I as 165% for single word read and 374% for single word write accesses. In contrast, sequential burst accesses make good use of the TLB, causing 1 miss, followed by 31 hits for each accessed page. This results in an average overhead of 7.3% for sequential reading and 10.6% for writing.

Since page faults caused by demand paging are very costly when encountered by a hardware thread, it is advisable to initially access allocated pages in software first or to turn off demand paging entirely.

VI. Conclusion and Outlook

Due to their combination of reconfigurability and performance, modern platform FPGAs represent an implementation platform with high potential for safety-critical applications. While multithreaded programming for reconfigurable hardware offers considerable benefits in exploiting the flexibility of these platforms, missing support for data separation via memory virtualization, as requested by certification guidelines, currently prevents the adoption of this promising development approach in safety-critical systems.

In this paper, we have presented a reconfigurable architecture for multithreaded hardware, which extends OS-controlled virtual memory services to hardware threads. Our hardware-thread-aware address translation units enable reconfigurable hardware threads to transparently use virtual memory addresses, which ensures both data integrity of functionally independent processes and a unified memory model across hard- and software. Thus, our approach enhances system safety as well as designer productivity and opens the area of safety-critical systems as a new application domain for multithreaded reconfigurable hardware.

In future research, we are going to extend ReconOS to provide not only spatial but also temporal separation of hardware threads by further investigating predictable scheduling techniques using partial reconfiguration. These mechanisms will also require adaptive management of the HWT-MMUs to cope with changing hardware threads from possibly different processes. In consequence, we are also going to develop application case studies from safety-relevant domains to demonstrate our separation mechanisms in a real-world setting.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n 257906. This work was also partly supported by the German Research Foundation under project number PL471/2-1.

References

[1] Xilinx, "Virtex-6 FPGA Family Overview (DS150)," Data Sheet, January 2010.

[2] Altera, "Stratix V Device Family Overview," White Paper, January 2011.

[3] Xilinx, "Zynq-7000 EPP Extensible Processing Platform," Product Brief, 2011.

[4] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems (TECS), Special Issue CAPA*, 2009.

[5] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions for Reconfigurable Computing," *IEEE Transactions on Large Scale Integration (VLSI) Systems*, 2008.

[6] N. W. Bergmann, J. A. Williams, J. Han, and Y. Chen, "A Process Model for Hardware Modules in Reconfigurable Systems-on-Chip," in *Architecture of Computing Systems, Dynamically Reconfiguable Systems Workshop*, 2006.

[7] A. Keller, B. Plattner, E. Lübbers, M. Platzner, and C. Plessl, "Reconfigurable Nodes for Future Networks," in *Proc. IEEE Globecom Workshop on Network of the Future (FutureNet)*. IEEE, Dec. 2010, pp. 372–376.

[8] S. Samara, "Partitioning granularity, communication overhead, and adaptation in OS services for Distributed Reconfigurable Systems on Chip," in *The 13th IEEE International Conference on Computational Science and Engineering*. IEEE Computer Society, Dec. 2010.

[9] M. Happe, E. Lübbers, and M. Platzner, "An Adaptive Sequential Monte Carlo Framework with Runtime HW/SW Repartitioning," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, Dec. 2009.

[10] G. J. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," in *International Workshop on Field-Programmable Logic and Applications (FPL)*, 1996.

[11] C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Realtime Tasks," *IEEE Transactions on Computers*, 2004.

[12] M. Vuletic, L. Pozzi, and P. Ienne, "Virtual Memory Window for Application-Specific Reconfigurable Coprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2006.

[13] P. Garcia and K. Compton, "A Reconfigurable Hardware Interface for a Modern Computing System," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, 2007.

[14] H. Lange and A. Koch, "Low-Latency High-Bandwidth HW/SW Communication in a Virtual Memory Environment," in *Field Programmable Logic and Applications (FPL)*, 2008.

[15] D. S. Miller, "Cache and TLB Flushing Under Linux," Linux Kernel Documentation, December 2008, linux/Documentation/cachetlb.txt.

[16] "Xilinx UG011 PowerPC Processor Reference Guide," 2007.