Supporting Multithreading in Configurable Soft Processor Cores

Roger Moussali, Nabil Ghanem, and Mazen A. R. Saghir
Department of Electrical and Computer Engineering
American University of Beirut
P.O. Box 11-0236 Riad El-Solh, Beirut 1107 2020, Lebanon
{ram32,nwg02,mazen}@aub.edu.lb

ABSTRACT

In this paper, we describe the organization and microarchitecture of MT-MB, a configurable implementation of the Xilinx MicroBlaze soft processor that supports multithreading. Using a suite of synthetic benchmarks, we evaluate five variations of MT-MB and show that multithreading is very effective in hiding the variable latencies associated with custom instructions and custom computational units. Our experimental results show that interleaved and hybrid multithreading achieve speedup factors of 1.10× to 5.13× compared to our single-threaded baseline soft processor.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architectural Styles—*adaptable architectures*; B.5.1 [**Register-Transfer-Level Implementation**]: Design—*control design, data-path design*; C.4 [**Performance of Systems**]: Design Studies

General Terms

Design, Experimentation, Measurement, Performance

1. INTRODUCTION

Contemporary field programmable gate arrays (FPGAs) are characterized by high logic densities and rich sets of embedded hardware components that, together with advanced CAD tools, are transforming FPGAs into powerful computational engines. Today, an increasing number of these computational engines are designed using *soft processor cores* that are specified in a hardware description language and implemented in the logic fabric of the FPGA [1, 2, 3]. The datapath of a soft processor core can be easily configured to meet specific functional requirements or design constraints of a target application. Its instruction set architecture can also be extended to support custom machine instructions that typically yield significant improvements in execution performance. Since custom instructions are application dependent and may have arbitrary latencies, it is often difficult to design well-balanced datapaths and pipelines that are suitable for both general-purpose and custom applications. That is why *multithreading* has recently emerged as a promising architectural style for enhancing the performance and area efficiency of configurable soft processors [4, 5]. For example, *interleaved multithreading* [6] can be used to eliminate pipeline stalls due to various hazards among instructions from the same thread. It can also be used to eliminate data forwarding logic, which reduces both the area and clock cycle time of the datapath. Finally, it can be used to share a pipelined computational unit among different threads.

In this paper we describe a number of microarchitectural enhancements we developed to support multithreading in the Xilinx MicroBlaze [1], a widely-used, single-threaded, soft processor core. We also evaluate the impact of these enhancements on both performance and area. The main contributions of our paper include the introduction of a configurable thread scheduler and table of operation latencies (TOOL) that simplify thread management; the introduction of *hybrid multithreading*, which combines interleaved and block multithreading and results in efficient reuse of instruction issue slots associated with stalled threads; the introduction of an area-efficient and scalable register file that enables multiple threads to access independent register contexts; the integration of variable-latency custom computational units (CCUs) into the MicroBlaze datapath; and a quantitative evaluation of these techniques.

Our paper is organized into five sections. In Section 2 we describe related work and compare it to our own. In Section 3, we describe the various microarchitectural enhancements we introduced to support multithreading. Then, in Section 4, we evaluate the impact of the various enhancements on performance, and we analyze our results. Finally, in Section 5, we summarize our work and present our conclusions.

2. RELATED WORK

Although little work has been done on multithreaded soft processors, early studies provide interesting results. In [4], the authors describe the CUStomisable Threaded ARchitecture (CUSTARD), a multithreaded soft processor with a customizable instruction set architecture. Several aspects of the CUSTARD processor are configurable including the number of threads the processor can execute, the datapath width, and the multithreading technique used. The processor supports two multithreading techniques: block multithreading (BMT) and interleaved multithreading (IMT) [6].

Although the authors compare their own implementations of single-threaded and four-way interleaved and block multithreaded processors, their results show that most of the performance is due to custom instructions used to accelerate performance-critical code blocks. However, it is not clear what levels of performance are achieveable through multithreading. Moreover, since the various datapaths are specified in Handel-C, the resulting areas and clock cycle times are considerably poorer than those of commercial or better-optimized soft processors [1, 2].

Other researchers have studied multithreading as a more area-efficient alternative to chip multiprocessing, particularly when the processors are required to share computational or storage resources. In [5], the authors describe UTMT-II, a four-way, interleaved-multithreading soft processor based on the Altera NIOS-II instruction set architecture [2]. The authors present techniques for reducing stalls among instructions belonging to different threads. These include pipelining computational units and buffering stalled instructions to enable instructions from other threads to continue execution. The authors also describe a technique for synchronizing writebacks in long-latency instructions to avoid having different instructions from the same thread in the pipeline. This is achieved by extending the latencies of CCUs so they become multiples of the pipeline depth. In Section 3, we propose a more unified approach to solving similar problems, and demonstrate that our approach results in more flexible and efficient thread scheduling.

3. MT-MB: A CONFIGURABLE MULTI-THREADED SOFT PROCESSOR

In this section we describe the organization of MT-MB, our multithreaded implementation of the Xilinx MicroBlaze processor. We also describe the different microarchitectural features we used to support different forms of multithreading.

3.1 Datapath Organization and Pipeline Structure

Figure 1 shows the datapath of the MT-MB processor, which is designed around a single-instruction-issue, five-stage pipeline similar to that of the MicroBlaze processor [1]. However, unlike the MicroBlaze processor, its datapath and pipeline are configurable with a number of features that support different forms of multithreading that include single-threaded (ST-MB), interleaved (IMT-MB), and hybrid (HMT-MB) multithreading. ST-MB is our own implementation of the MicroBlaze instruction set architecture, which we also extended to support tightly integrated custom computational units (CCUs). IMT-MB is a version of the MicroBlaze datapath designed to support n-way interleaved multithreading, and HMT-MB is a new datapath designed to implement a hybrid form of n-way interleaved and block multithreading.

During the instruction fetch stage, the next instruction is fetched from the appropriate thread based on the specified form of multithreading. In the ST-MB implementation, a single program counter (PC) is used to fetch the next instruction from the instruction memory (IMEM) bank. In both the IMT-MB and HMT-MB implementations, a configurable *thread scheduler* is used to generate a thread identifier (TID) that selects an appropriate PC for fetching the next instruction from the corresponding thread. To support n-way multithreading, a PC block containing n PCs is used. In Section 3.2, we explain the operation of our thread scheduler in greater detail.

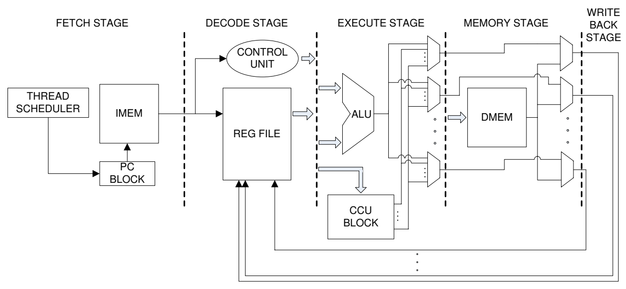

Figure 1: Datapath of the MT-MB Soft Processor

During the decode stage a fetched instruction is decoded and its operands are read from the register file. To support n-way multithreading, our register file is designed to support n, separate, thread contexts. The MT-MB register file is organized into n, 32×32-bit banks where n is the number of threads and where each bank is associated with a different thread [7]. Since MicroBlaze instructions operate on as many as three source operands and may produce a single result, each bank provides three read ports and one write port. The register banks are implemented using dual-ported block RAMs (BRAMs) embedded within most Xilinx FPGA devices [9].

During the execute stage instructions are executed in corresponding computational units. Although most instructions are executed in the ALU and have a latency of one clock cycle, the MT-MB processor also supports the execution of variable-latency, custom instructions on tightly-integrated custom computational units (CCUs). This differs from the MicroBlaze processor, which only supports CCUs as external hardware blocks accessible through dedicated communication channels. To maintain compatibility with the MicroBlaze instruction set architecture, each CCU is constrained to operate on at most three register operands and generate a single result. However, we place no constraints on the latencies of custom instructions, or on whether a CCU is pipelined or not.

During the memory stage, load and store instructions access a unified data memory bank. Although all threads share the same address space, we assume each thread operates in its own memory region. We also assume threads do not access each other's memory regions.

Finally, during the write back stage, instructions write their results back to the register file. Since multiple instructions from different threads may reach the write back stage simultaneously, our register file is also designed to support multiple simultaneous write backs from different threads.

In MT-MB, only the ST-MB datapath implements data forwarding to eliminate data hazards. The IMT-MB and HMT-MB datapaths rely on their thread schedulers to schedule instructions from the same thread sufficiently far apart to eliminate data hazards. Earlier work has shown that in an interleaved-multithreading processor with an m-stage pipeline, instructions from the same thread must be scheduled at least $m - 1$ pipeline stages apart [4, 5]. However, our pipeline allows instructions from the same thread to be scheduled as few as $m - 2$ pipeline stages apart due to the low latency of the register file, which enables results from the write-back stage to update the register file before new operands are read in the decode stage. This, in turn, enables overlapping instructions from the same thread in the decode and write-back stages. Reducing the intra-thread instruction issue window increases the per-thread throughput of the processor and helps reduce the per-thread CPI. It also enables the processors to tolerate executing fewer threads before having to insert empty instruction slots.

3.2 Configurable Thread Scheduler

Our MT-MB processor supports two forms of multithreading: interleaved multithreading (IMT) and a hybrid form of interleaved and block multithreading (HMT). Like interleaved multithreading, the HMT-MB datapath attempts to execute a new instruction from a different thread on every instruction cycle. Once a thread stalls due to a hazard or long-latency operation, the thread is temporarily suspended while other threads continue to execute in an interleaved fashion. However, rather than issue a NOP like IMT, the instruction issue slot of the stalled thread is used to issue an instruction from *another* active thread. Once the stalled thread becomes active again, its instruction issue slot is returned to it. Since the suspension of a long-latency thread resembles the thread switch in a block multithreading processor, we refer to this form of multithreading as *hybrid multithreading* (HMT).

To implement the required form of multithreading and manage the execution of threads, we use a configurable *thread scheduler*. To support n-way multithreading, our thread scheduler uses a modulo-n *thread counter* to generate thread identifiers (TIDs) in a round-robin manner. A thread whose TID corresponds to the value in the thread counter is said to be the *active thread*. For each thread, the thread scheduler also uses a separate *status counter* to monitor the status of the thread. When an instruction is fetched from memory, its opcode is used to load the corresponding status counter with the latency of the instruction, and the status counter is decremented on every instruction cycle. As long as the value in the status counter is not zero, the corresponding thread is considered *busy*. When IMT is being used and the status counter of the active thread is not equal to zero, the thread scheduler prevents a new instruction from the same thread from being fetched and inserts a NOP into the pipeline. On the other hand, when HMT is being used and the status counter of the active thread is not equal to zero, the thread scheduler ensures that a new instruction from the next available ready thread is fetched, and the thread counter is updated accordingly.

3.3 Table of Operation Latencies (TOOL)

The latency of an instruction plays a central role in managing the execution of the corresponding thread. If the thread scheduler knows the latency of an instruction it can determine the time it needs to stall the thread. It can also attempt to schedule instructions from other threads in the instruction slots corresponding to a stalled thread. Finally, instruction latencies can be used to resolve structural hazards on computational resources that are shared among different threads.

Table 1: Processor Implementation Results

Processor	Multithreading Type	Clock Rate	Slices	Slice FFs
MB	1-way	106 MHz	683	553
ST-MB	1-way	84 MHz	732	550
IMT-MB-4	4-way, interleaved	99 MHz	1,648	988
HMT-MB-4	4-way, hybrid	98 MHz	1,713	988
IMT-MB-8	8-way, interleaved	90 MHz	1,937	1,561
HMT-MB-8	8-way, hybrid	89 MHz	2,016	1,121

Since the latencies of most instructions are known at design time, we developed a hardware component called the *table of operation latencies (TOOL)* that returns the latency of an instruction based on its opcode. The latencies generated by TOOL are used by the thread scheduler, which determines when the next instruction in a thread can be fetched. Currently, the latencies produced by TOOL are generated by a hardwired circuit that is configured at design time. TOOL can also be used to store the latencies of system events, such as cache misses, to dynamically update instruction latencies in response to these events.

3.4 Datapath Implementation Results

Table 1 summarizes the clock rates and FPGA resource utilization of the Xilinx MicroBlaze processor (v.5.0) and five implementations of our MT-MB processor. We obtained our results from the reports generated by the Xilinx EDK (v.8.2) and ISE (v.8.2.03i) tools targeting a Xilinx XC2V1000-4BG575 Virtex-II FPGA. Our results show that our ST-MB processor is about 20% slower and 7% larger than the commercial MicroBlaze processor. Our results also show that our multithreaded processors are faster but larger than the ST-MB processor. The faster clock rates are mainly due to the elimination of data forwarding, which reduces the critical delays in the datapath, while the larger datapath areas are due to the more complex instruction fetch stages, which include the thread schedulers, TOOL, and PC blocks.

4. RESULTS AND ANALYSIS

In this section we explain our benchmarking methodology and present the results of the experiments we conducted to evaluate the performance of our four-way and eight-way multithreaded processor implementations.

4.1 Benchmarking Methodology

To evaluate the effectiveness of multithreaded soft processors in hiding the latencies of various types of CCUs we developed a suite of *synthetic* benchmarks that mimic the behavior of a number of embedded applications taken from the MiBench benchmark suite [8]. Our suite consists of eight benchmarks whose execution times are aggregates of different time components that include: single-cycle instructions; multi-cycle custom instructions; and stall cycles due to dependencies on load instructions, taken branches, structural hazards, or external I/O events. The latencies of multi-cycle time components vary from 7 to 300 clock cycles and represent a wide range of custom instructions and system events. By assigning a weighting factor to each time component and varying the weights, we can mimic different program behaviors.

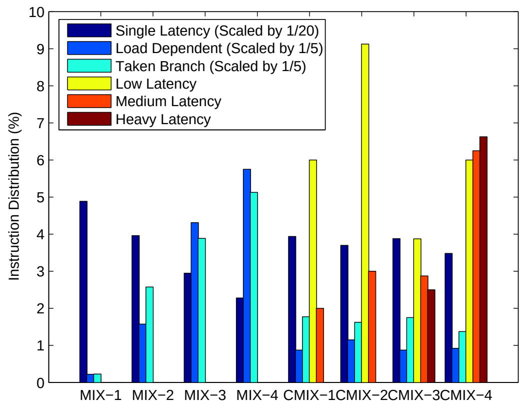

Figure 2: Mean Distribution of Execution Time Components

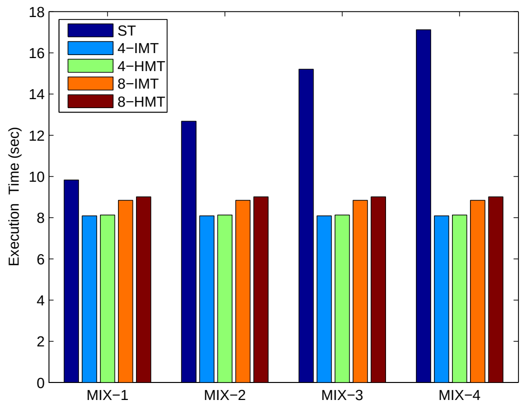

Figure 3: Execution time in the absence of CCUs

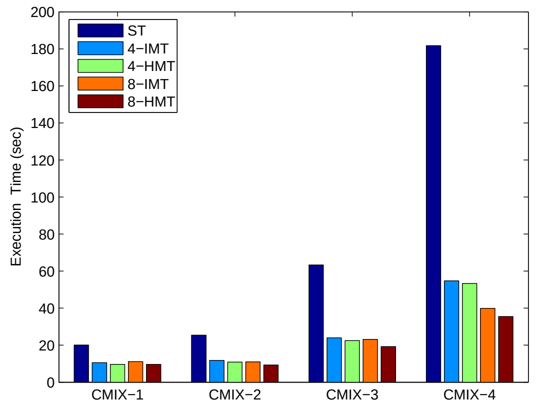

Figure 4: Execution time with pipelined CCUs

Figure 2 shows the average distribution of execution time components for different execution *mixes* used in this study. Each mix consists of variations of our eight benchmarks, and represents a certain class of applications with specific execution characteristics. When we evaluate the ST-MB processor, we combine the eight benchmarks in each mix to execute as a single thread. On the other hand, when we evaluate the four-way multithreading processors, we combine pairs of benchmarks in each mix to form four separate threads. Finally, when we evaluate the eight-way multithreaded processors, we treat each benchmark as a separate thread.

4.2 Datapaths without CCUs

In our first experiment, we assessed the performance of the multithreaded processors in the absence of CCUs. We used benchmark mixes MIX1 through MIX4, which mimic the behavior of application programs that use regular instructions only. Those benchmarks are also characterized by an increasing number of stall cycles due to data and control dependencies when executed on the ST-MB processor. While MIX2 and MIX3 are modeled after the MiBench suite's *Basic Math* and *Dijkstra* benchmarks, which exhibit moderate amounts of stall cycles, MIX1 and MIX4 represent extreme cases with low and high stall cycles, respectively.

Figure 3 shows the wall clock execution time of all four mixes when executed on the different processors. As expected, ST-MB exhibits the worst performance since it cannot hide all stall cycles. On the other hand, each of the multithreaded processors completely hides the stall cycles in each thread. This is due to the degree of multithreading – four and eight threads – being greater than the largest number of stall cycles in any given thread. The result is that all multithreaded processors execute all benchmark mixes in the same number of clock cycles. However, the differences in execution time among the multithreaded processors are due to the differences in their clock cycle times. Overall, the multithreaded processors are 1.09 (HMT-MB-8) to 2.12 (IMT-MB-4) times faster than ST-MB.

4.3 Datapaths with pipelined CCUs

In our second experiment, we assessed the performance of the multithreaded processors in the presence of pipelined CCUs. We used benchmark mixes CMIX1 through CMIX4, which use a mix of custom instructions with increasing latencies. However, since all CCUs are pipelined, we assume no thread stalls due to a busy CCU.

Figure 4 shows the execution times for the different processors. Our results show that multithreading is very effective in hiding stall cycles due to long-latency custom instructions. For example, HMT-MB-8 is 2.09 to 5.13 times faster than ST-MB. Our results also show that HMT achieves better performance than IMT since it uses the instruction issue slots of stalled threads more effectively. In fact, in CMIX-2 and CMIX-3, HMT-MB-4 achieves similar performance to IMT-MB-8 even though it executes fewer threads.

4.4 Datapaths with non-pipelined CCUs

In our third experiment, we assessed the effectiveness of multithreading in the presence of non-pipelined CCUs. When a CCU is not pipelined, only one thread can use it at a time.

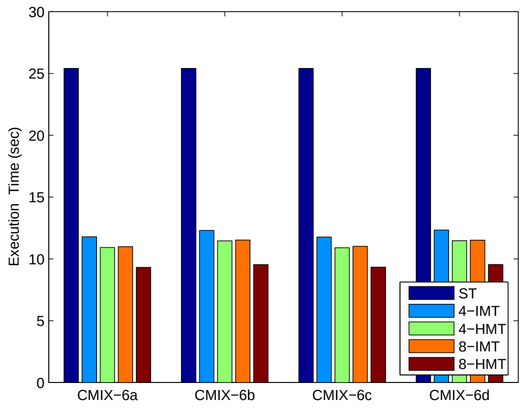

Figure 5: Execution time with non-pipelined CCUs

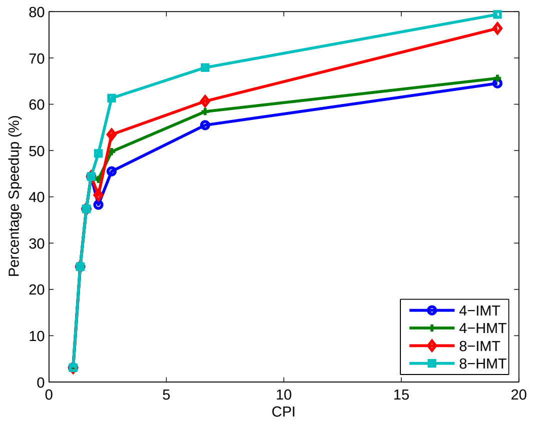

Figure 6: Percentage speedup vs. increasing single-threaded CPI

If the CCU has a high latency and competetion among threads to use it is high, more threads will stall and performance will degrade. We therefore created a new mix, CMIX-6, that uses three CCUs with 7, 27, and 60 cycle latencies, respectively. We then evaluated the performance of our processors when each CCU was made non-pipelined individually (CMIX-6a through CMIX-6c) and collectively (CMIX-6d). Figure 5 shows the resulting execution times.

As expected, the behavior of ST-MB remains unchanged since it does not exploit thread-level parallelism. Although we did not observe a significant difference in the results of various mixes, IMT-MB and HMT-MB still achieve speedup factors of 2.15 to 2.72 relative to ST-MB, which corresponds to a slight degradation in performance compared to the original average speedup factor of 2.74 achieved when pipelined CCUs are used. Our results again show that HMT-MB achieves better performance than IMT-MB since it is better at hiding stall cycles.

4.5 Effects of Increasing Single-Threaded CPI

In our last experiment, we plotted the percentage speedup achieved by our multithreaded processors as a function of increasing average cycles per instruction (CPI) for the baseline ST-MB processor. Increasing CPI correlates with increasing stall cycles due to various hazards and long-latency custom instructions and system events. Figure 6 shows the resulting graphs, which confirm that our multithreaded processors improve performance by hiding the cycles associated with stalled threads. The graphs also confirm that exploiting higher levels of multithreading achieves higher levels of performance, and that HMT is generally more effective than IMT. The graphs also show that the performance of HMT and IMT converge at high CPI values due to the corresponding increase in the number of stalled threads. In fact, it can be shown that, in the worst case, the performance of an n-way HMT processor designed around a five-stage pipeline degenerates to that of a 3-way IMT processor.

5. SUMMARY AND CONCLUSIONS

In this paper, we describe the organization and microarchitecture of MT-MB, our multithreaded implementation of the Xilinx MicroBlaze soft processor core. To support multithreading, MT-MB relies on a configurable, centralized, thread scheduler that uses instruction latencies to manage the execution of threads. The thread scheduler supports interleaved and hybrid multithreading, which combines interleaved and block multithreading. The datapath also includes a scalable and area-efficient register file for supporting a variable number of thread contexts. Our experimental results show that multithreading is very effective in hiding the variable latencies of custom instructions executed on custom computational units. Using a suite of synthetic benchmarks, we show that interleaved and hybrid multithreading achieve speedup factors of 1.10× to 5.13× compared to a single-threaded soft processor. Although hybrid multithreading generally achieves better performance than interleaved multithreading, their performance levels converge as the number of stalled threads increases.

6. REFERENCES

[1] *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 9.1i*, UG081 (v7.0), Sept. 15, 2006, `http://www.xilinx.com`.

[2] *NIOS-II Processor Reference Handbook*. `http://www.altera.com`.

[3] Cortex-M1 Product Brief. `http://www.actel.com`.

[4] R. Dimond, O. Mencer, and W. Luk, "CUSTARD - a Customizable Threaded FPGA Soft Processor and Tools", *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2005)*, pp. 1–6, IEEE, Aug. 24-26, 2005.

[5] B. Fort et al., "A Multithreaded Soft Processor for SoPC Area Reduction", *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pp. 131–142, IEEE, Apr. 24-26, 2006.

[6] T. Ungerer, B. Robic, and J. Silc, "A Survey of Processors with Explicit Multithreading", *ACM Computing Surveys*, pp. 29–63, ACM, Vol. 35, No. 1, March 2003.

[7] M. A. R. Saghir and R. Naous, "A Configurable Multi-Ported Register File Architecture for Soft Processor Cores", *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC 2007)*, pp. 14–25, Springer-Verlag LNCS 4419, Mar. 27-29, 2007.

[8] M. R. Guthaus et al., "MiBench: a free, commercially representative embedded benchmark suite", *Proceedings of the 2001 IEEE International Workshop on Workload Characterization (WWC-4)*, pp. 3–14, IEEE, Dec. 2, 2001.

[9] Xilinx Corporation, "Using Block RAM in Spartan-3 Generation FPGAs", *Xilinx Application Note XAPP463 (v2.0)*, March 1, 2005.