ABSTRACT

This paper describes a number of microarchitectural techniques for supporting multithreading in soft processor cores. These include a new thread scheduler that combines interleaved and block multithreading; a table of operation latencies (TOOL) for determining instruction latencies; support of arbitrary-latency custom computational units; and a multi-banked register file for supporting simultaneous write-back operations from different threads. Our results show that four-way, multithreaded, processors achieve speedups of up to 26% over a single-threaded processor executing benchmarks that only use regular instructions, and up to 47% when executing benchmarks that include long-latency instructions.

1. INTRODUCTION

The high logic densities and rich on-chip features of platform FPGAs are increasing the proportion of FPGAs used to implement computational engines that couple software programmable processor cores with custom hardware blocks. Although it is common to find one or more hard processor core in many platform FPGAs, an increasing number of computational engines are being designed using soft processor cores [1, 2]. The datapath of a soft processor can be easily configured to match the needs of a target application, and its instruction set architecture can also be extended to support user-defined machine instructions, which often yield significant performance improvements. The microarchitecture of a soft processor can also be optimized to use available on-chip resources, which occupy less area and have lower latencies than logic implemented in the fabric of an FPGA. Performance and area can be further optimized by using suitable architectural styles. For example, interleaved multithreading can be used to eliminate costly data forwarding logic while block multithreading can be used to better utilize the datapath during long latency operations.

In this paper we present a number of microarchitectural enhancements we developed to support various forms of multithreading in soft processors. We evaluate the impact of these enhancements on area and performance using different implementations of the Xilinx MicroBlaze processor [1]. In Section 2 we describe related work and compare it with our own. In Section 3 we describe the microarchitectural enhancements we introduced to support multithreading. Then, in Section 4, we provide the results of our multithreaded processor implementations, and in Section 5 we evaluate the performance of the various implementations. Finally, in Section 6, we provide our conclusions and describe future work.

2. RELATED WORK

Although little work has been done on multithreaded soft processors, early studies provide interesting results. In [3], 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. 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, and it is not clear how multithreading affects performance. 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.

Other researchers have studied multithreading as a more area-efficient alternative to chip multiprocessoring, particularly when the processors are required to share computational or storage resources. In [4], the authors describe UTMT-II, a four-way, interleaved-multithreading soft processor based on the NIOS-II instruction set architecture [2]. The authors present techniques for reducing stalls among instructions associated with different threads that 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 that requires the latencies of CCUs to be multiples of the pipeline depth. In Section 3, we propose a more unified approach to solving similar problems, and demonstrate that they are more flexible and result in more efficient thread scheduling.

3. MICROARCHITECTURAL SUPPORT FOR MULTITHREADING

This section describes a number of microarchitectural techniques for supporting multithreading in soft processor cores. To evaluate these techniques, we developed a single-threaded implementation of the Xilinx MicroBlaze processor (ST-MB) that we used as our baseline. MicroBlaze is a widely used commercial soft processor core optimized for implementation in Xilinx FPGAs. We also developed two multithreaded implementations of the MicroBlaze processor. The first is based on an interleaved multithreading architecture (IMT-MB) while the second implements a hybrid form of multithreading that combines interleaved and block multithreading (HMT-MB). Since multithreading can be used to hide stall cycles due to various hazards, both the IMT-MB and HMT-MB processor do not use data forwarding logic.

3.1. Thread Schedulers

The thread scheduler is responsible for managing the execution of different threads. When a thread stalls due to data, control, or structural hazards, the thread scheduler must try to keep the datapath busy by scheduling instructions from other threads.

Figure 1 shows the instruction fetch pipeline stage for both the IMT-MB and HMT-MB processors. Multiple program counters (PCs) – one for each thread – are used to fetch instructions from different threads in different areas in memory. Depending on the states of the various threads, the thread scheduler produces the thread identifier (TID) that selects the PC to be used for fetching the next instruction from the corresponding thread. To keep track of the states of different threads, the scheduler associates a counter with each thread. The counter is initialized with the latency of the instruction fetched from memory. The counter is decremented every clock cycle and the thread remains suspended until the count reaches zero. When a thread is suspended no new instructions can be fetched from that thread. Normal instruction fetching resumes when the thread becomes active again.

Our thread schedulers were designed to support two forms of multithreading: IMT and HMT. In IMT, instructions are issued in a round robin fashion from different threads, respectively. If a thread is suspended, the thread scheduler issues a NOP in the thread’s issue slot. In HMT, instructions are issued from different threads in an interleaved manner like IMT. However, when a thread is suspended, the scheduler uses the thread’s issue slot to issue an instruction from another active thread, and instructions from active threads continue to execute in an interleaved manner. Figure 2 shows an example of how HMT works.

3.2. Table of Operation Latencies (TOOL)

As explained earlier, the latency of an instruction is an essential parameter used by the thread scheduler to manage the state of a thread. The table of operation latencies (TOOL) is a circuit that uses an instruction’s opcode field to generate the corresponding instruction latency used to initialize the thread counter in the scheduler. The TOOL can be easily configured to support instructions with arbitrary latencies.

3.3. Custom Computational Units (CCUs)

Unlike the original MicroBlaze processor, all of our processor implementations support the integration of arbitrary-
latency custom computational units (CCUs) in the datapath. CCUs enhance execution performance by implementing computationally demanding operations, such as floating-point multiplication or addition, in hardware. Since CCUs can be used to execute custom instructions with arbitrary latencies, our multithreaded processors, which use enhancements such as TOOL, are very effective in hiding these latencies.

3.4. Register File Organization

To support multithreading, the register files used in our processors associate a different register bank with each thread [6]. Since each bank has an independent write port there is no contention between threads. This greatly simplifies the scheduling of variable latency instructions and avoids the need for special synchronization logic to prevent instructions from different threads from competing for the same write port. The register file is also designed to provide up to three register operands from each bank. The TID is used to select the bank from which the operands must be fetched. Figure 3 shows the organization of our register file.

4. IMPLEMENTATION RESULTS

To evaluate our IMT-MB and HMT-MB processors, we implemented each to support four-way multithreading. We then used the reports generated by the Xilinx ISE tools (v. 8.2i) to compare their clock cycle times and areas relative to the ST-MB. To calibrate our results, we also compared against the commercial MicroBlaze processor. Table 1 summarizes our results.

Our results show that our implementation of the ST-MB processor is roughly 31% larger and 16% slower than version 5.0 of the commercial MicroBlaze processor. This difference is likely due to special optimizations (e.g., in placement and routing) implemented in the Xilinx CAD tools to enhance the performance and area of the MicroBlaze processor.

Our results also show that the IMT-MB and HMT-MB processors are 21% and 10% faster than the ST-MB processor, respectively. This is mainly due to the elimination of data forwarding logic and paths, which reduces the critical path in the datapath. Our results also show that the IMT-MB and HMT-MB are 33% and 38% larger, and use 62% more slice flip-flops, respectively, than the ST-MB processor. This increase is mainly due to the additional logic and pipeline latches needed to support the two forms of four-way multithreading. Finally, our results show that HMT-MB is 9% slower than IMT-MB. Although the degraded performance is due to the more complex thread scheduler, HMT-MB still runs 10% faster than ST-MB due to the elimination of forwarding paths.

5. PERFORMANCE ANALYSIS

In this section, we compare the performance of our three processor implementations when executing a suite of synthetic benchmarks that mimic the behavior of a workload consisting of four threads. In the case of the ST-MB processor, we assume the threads execute sequentially.

5.1. Base ISA

In our first experiment, we evaluated the performance of our processors using benchmarks that only contain instructions from the base MicroBlaze ISA. Benchmarks Blend 1 to Blend 3 exhibit an increasing proportion of pipeline stalls due to taken branches (2 stall cycles) and instructions that depend on the results of load instructions (1 stall cycle). While the former affects 5%, 8%, and 12% of the instructions in each benchmark, respectively, the latter affects 4%, 7%, and 8% of the instructions in each benchmark, respectively.

As expected, the ST-MB needs more time to complete executing all threads. Moreover, IMT-MB and HMT-MB become more effective as the frequency of stalls increases. Figure 4 shows that IMT-MB and HMT-MB improve performance from 12% to 26% by using multithreading to hide
all stall cycles. Since all instructions in the base ISA have short latencies, none of the threads becomes suspended and both processors consume the same number of clock cycles to execute the three benchmarks. However, differences in clock cycle time result in different execution times.

5.2. Base ISA + Custom Instructions

In our second experiment, we studied the impact of our multithreaded processors on the execution performance of benchmarks that use long-latency custom instructions (CIs). The latencies of the CIs varied from 5 to 16 to 32 cycles. In benchmark CMIX 1, CIs are used in only one of the threads where 4% of the issued instructions correspond to five-cycle CIs and 2% correspond to 16-cycle CIs. On the other hand, in CMIX 2 and CMIX 3, two threads use long-latency CIs. In CMIX 2, 5-, 16-, and 32-cycle CIs occur with frequencies of 4%, 2%, and 2%, respectively. On the other hand, in CMIX 3, CIs with similar latencies occur with frequencies of 4%, 5%, and 4%, respectively.

Figure 5 shows that ST-MB clearly performs the worst among all processors. On the other hand, both IMT-MB and HMT-MB exhibit a significant performance boost. Nonetheless, in both IMT-MB and HMT-MB, not all stall cycles can be eliminated. However, HMT-MB makes better use of the stall cycles by scheduling instructions from active threads. In CMIX 2, HMT-MB is 23% faster than IMT-MB, and in CMIX 3, it is 30% faster. HMT-MB is also 47% faster than ST-MB using CMIX 3.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we described a number of microarchitectural techniques for supporting multithreading in configurable soft processor cores. These include thread schedulers capable of supporting interleaved multithreading (IMT) along with a new, hybrid form of multithreading that combines interleaved and block multithreading (HMT). Both thread schedulers use a table of operation latencies (TOOL) for determining the latencies of different instructions, and this provides support for arbitrary-latency custom computational units. A multibanked register file also supports simultaneous register write-back operations from different threads. Our results show that four-way multithreading achieves a speedup of up to 26% on benchmarks that only use regular instructions. On the other hand, multithreading achieves a speedup of up to 47% on benchmarks that use long-latency custom instructions. To continue this work are currently investigating 8-way multithreaded architectures and testing larger benchmarks. Furthermore, we are developing an IEEE-compliant floating-point unit to evaluate the impact of multithreading on floating-point application benchmarks.

7. REFERENCES