Simple Symmetric Multithreading in Xilinx FPGAs

Spring 2002 CS 252

Semester Project

Yury Markovskiy, Yatish Patel
{yurym,yatish}@cs.berkeley.edu

May 16, 2002

Abstract

Current methods to improve the performance of a microprocessor require significant investments in time and yield larger complicated designs. This paper explores a transformation called C-slow retiming to quickly and automatically convert a standard single threaded microprocessor into a multithreaded microprocessor with improved performance. Our experiments have demonstrated multithread instruction throughput improvement of 21% on a 2-slow and 31% on 3-slow design with minimal area cost.

1 Introduction

Field Programmable Gate Arrays (FPGAs) offer numerous readily available resources to pipeline designs to improve their throughput and performance. Yet when implementing a microprocessor or a similar design, the overhead involved in pipelining for a higher clock rate becomes highly significant, notably due to the much increased bypass logic, additional instruction latency, and more complicated control logic.

One way around this limitation is Simple Symmetric Multithreading. In this case, the register file and TLB size are increased, and each pipeline and control logic register is replaced by several registers, which are then moved to balance the combinational circuit delays. These transformations, known as C-slow and retiming, will be discussed in Section 3. Note that the complexity of the bypassing is not increased, nor are there significant changes in the control logic. Separate instruction streams (threads) are run through the pipeline in a round robin fashion, making the single processor core (running at a higher clock rate) behave like an SMP without a significant increase in complexity or circuit area. The aforementioned transformations offer additional advantages of being simple, straightforward and can be performed automatically by appropriate tools.

This works demonstrates the application of C-slow and retiming transformations on a microprocessor core resulting in improvement of multithreaded performance. Although an FPGA was targeted, the results and conclusions can also be extended to targets such as ASICs.

We begin this report with a discussion of alternative approaches to improve throughput and performance in microprocessors in Section 2. C-slow and retiming transformations and their application to a microprocessor core are discussed in Section 3 and Section 4. The subsequent Sections 5 and 6 describe our methodology and analyze the results. Conclusions and the future directions will be discussed in the last two sections.

2 Related Work

With the modern on-chip transistor budget, the designers are finding new and creative ways to efficiently utilize processor resources (*e.g.* execution units) and, at the same time, hide variable latencies from memory and communication systems. Microprocessor architectures that concurrently execute several software threads and allow efficient resource sharing, can address both of these issues. In addition, multithreading offers significant gains in available instruction level parallelism resulting from interleaving several independent streams together. Two main techniques have been examined in academia and the industry to enable instructions from multiple independent streams to share processor resources: *fine-grained* and *simultaneous* multithreading.

Tera[1] multi-processor system is a good example of a *fine-grained multithreaded* superscalar processor which issues instructions from independent threads in a round robin fashion, context switching on every cycle. The tightly integrated memory subsystem is equipped to handle multiple concurrent contexts. *Fine-grained multithreading* can eliminate most of the pipeline hazards by alternating between threads with *independent* instructions. By increasing the distance between dependent instructions, a multithreaded processor eliminates most of the stalls. Thus many "Conventional Multithreaded"

processors do not employ any bypassing, since if the number of actively running threads is high (*e.g.* about 70 for Tera), the processor's resources and the memory subsystem are fully utilized without resorting to bypassing. However, should the number of active threads fall below an implementation specific threshold (typically the number of pipeline stages), the throughput may suffer dramatically.

We summarize the key performance metrics for *fine-grained multithreaded* processor in the first column "Conventional Multithreading" of the Table 1. We call the number of threads required to fully utilize all processor resources a *saturation point*. The *fine-grained multithreaded* processor has a high *saturation point* proportional to the number of pipeline stages and a low single-threaded throughput inversely proportional to the number of pipeline stages. Hence the *fine-grained multithreaded* processor is a bad candidate for deeper pipelining which is essential to further increase processor instruction throughput.

To remedy these concerns in the *fine-grained multithreaded* processor, the architecture from University of Washington[3] takes a completely different approach with *simultaneous multithreading*. The processor fetches instructions from several threads, allowing hardware to dynamically schedule them onto execution units by keeping track of dependencies (*e.g.* Tomasulo). This architecture can achieve higher resource utilization by taking advantage of combined instruction level parallelism from several threads.

While *simultaneous multithreading* is comparatively simple to implement and requires only minor modifications to an out-of-order core, it is only applicable to superscalar architectures with a large number of execution units. This technique would not work in a simple, classic five stage pipeline in-order processor, due to the lack of infrastructure to support multiple instructions in flight, leaving the choice of fine-grained multithreading with its shortcomings as discussed above.

The alternative, super/hyper-pipelining, does increase the operating clock frequency and throughput, but at the cost of complexity of implementing forwarding logic and control (in addition to branch mis-prediction penalty, *etc*). The third column in the Table 1 summarizes critical metrics for this technique. While super-pipelining yields a full-featured single threaded processor, which can be used as a multithreaded processor, the high design complexity may make the implementation impractical.

This project targets low cost, embedded processors, that cannot benefit from *simultaneous multithreading*, while attempting to reduce the impact of the performance and complexity problems of fine-grained and super-pipelined processors described above. In particular, we are interested in a simple approach that minimizes single-thread performance penalty resulting from the transformation of a simple core into a multithreaded processor.

3 C-slow retiming

C-slow retiming[4] is the transformation obtained by first C-slowing a circuit to allow multiple threads and then retiming the circuit to reduce the critical path delay.

The first part of C-slow retiming is C-slowing. C-slowing involves replacing each of the registers in a circuit with C registers in series. The resulting circuit has $C\times$ as many registers as the original but no additional combinational logic. The transformation does not create any additional feedback paths in the circuit to compensate for the added registers, thus the output latency of the circuit increases by a factor of C. C independent data streams must then be interleaved in order to fully utilize resources of the new circuit while maintaining correct functionality.

The second step in the transformation is retiming the circuit, which is the process of moving registers around in a circuit to balance delays in the critical path. Registers are either pushed forward or backward in the circuit to minimize the critical path, but unlike C-slowing, retiming does not affect the output latency of the circuit. Retiming may, however, increase the register count of the circuit, such as in the following example. If a register moves from the output of a two input AND gate to the inputs, the new circuit will require two registers, one for each input of the gate.

The combination of C-slowing and retiming yields a simple yet powerful method of transforming a single threaded circuit into a multithreaded circuit running at a higher frequency. C-slowing replaces each register with a chain of C registers which allows retiming to push the newly created registers across combinational logic to balance delays between registers. In other words if the critical path was T, the new critical path will approximately be T/C as the following example will illustrate.

An example of C-slow retiming a circuit is shown in Figure 1. Figure 1a is a simple circuit that takes a stream of numbers as input and outputs the maximum value seen in the input stream. The circle represents combinational logic and the rectangle is a register used to store the maximum value. Figure 1b is the result obtained after performing a 2-slow transformation on the original circuit with the only difference being the duplication of the register. After 2-slowing, the input to the circuit must consist of two interleaved, independent data streams. The output of the circuit will alternate between the maximum value of the two data streams. Figure 1c is the 2-slowed circuit after retiming has been performed to balance the critical path delay. The combinational logic was split into two equal pieces with one of the registers located in

Category	Conventional Multithread	Hybrid (C-slow)	Super/Hyper-pipelined
Examples	HEP[6]; Tera[1]; Intel IXP 1200[2]	C-slow LEON (SPARC)	P4
Description	• No bypass • Hazard avoidance • Issue instrns from alternating threads	• Keep bypassing of 5 stage core • C-slow retime • Issue instrns from $\leq C$ threads in alternation • SMP semantics	• Single Thread proc w/ deep pipeline • Stage-to-stage data forwarding and hazard detection
Resource saturation	N threads	C threads	N/A
Single thread	• Instr latency NT • Throughput $\frac{1}{NT}$	• Latency NCT' • Throughput $\frac{1}{CT'}$	• Latency NT • Throughput $\frac{1}{T}$
Multithread throughput	$\frac{k}{NT}$, where $1 \leq k \leq N$	$\frac{k}{NCT'}$, where $1 \leq k \leq C$	N/A
Area/design complexity	• Larger memory B/W + caches • Dynamic scheduler (hazard detection) • Stored state increased by N	• Larger memory B/W + caches • No scheduler, assignment by strict alternation • Simple forwarding (*i.e* classic 5 stage) • Stored state increased by C	• Complex forwarding • Hazard detection • Same size regfile and cache

Table 1: This table outlines the design space of pipelined microprocessors from conventional multithreaded to hyper-pipelined processors. In between, we envision the hybrid kind, which combines the advantages of fine pipelining and multithreading without incurring the costs of more complex forwarding control and logic. **Variables:** T the cycle time, T' cycle time after C-slow and re-timing transformation, N the number of pipeline stages, and k is the number of available threads to run.

between the two pieces. The resulting circuit can be clocked at nearly $2\times$ the frequency of the original, with an increase in throughput of $2\times$ when two independent data streams are available.

The elegance of the C-slow retiming transformation is that it lends itself well to automation by tools. [4] presents a polynomial time algorithm that performs synchronous circuit retiming optimizing the critical path. Some commercial tools such as Synplicity's HDL synthesis program, Synplify, contain options to apply retiming algorithms to designs. However, currently no tool supports automatic C-slow transformation.

4 C-slow retiming a microprocessor

Applying C-slow retiming to a microprocessor datapath transforms it into a multithread processor with SMP semantics. Ideally, the performance gains are significant, resulting in an increase in instruction throughput by a factor of C. However, in practice the limitations of pipelining, retiming, and C-slowing limit the performance gains. Pipelining limitations include register set up and clock-to-Q times that begin to dominate the critical path as the combination logic delay shrinks; retiming—inability of tools to "cut" through combinational logic in arbitrary places; C-slowing—a factor of C increase in the number of required registers.

Application of C-slow transform on a microprocessor core involves several steps. The datapath storage elements, including pipeline and status registers, the register file and the caches must be replicated by a factor of C. The register file and caches require special handling. In order to maintain the illusion of a single threaded machine to the thread, each thread strictly accesses its own register file segment.

Multiple options are available to modify the cache. First, the size of the cache can remain the same, however, with multiple threads competing for the original cache space, the

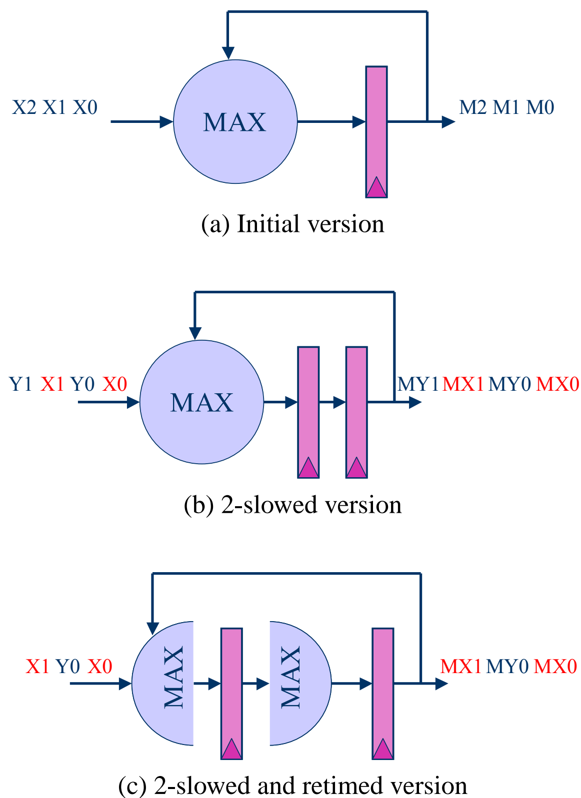

Figure 1: A circuit that outputs the maximum value of a stream of numbers.

cache miss rate is bound to increase dramatically. The second alternative is to increase the cache associativity and assign a block in the set to each individual thread. The cache size can be increased by a factor of C to keep the miss rate the same as in the original processor. A more flexible solution would be to allow all threads access to all blocks in the cache, opening the door to both destructive and constructive (synergy) interthread interaction.

The primary memory interface requires a modification to tag all memory transactions with an associated thread identification. This allows data returning from memory to be reinserted into the pipeline when the corresponding thread is active. The alternative would be to stall all the threads until the memory transaction completes resulting in poor performance.

After C-slowing, retiming the microprocessor will potentially improve the clock frequency up to a factor of C. The new datapath will maintain similar single thread performance to the original, but will be capable of executing instructions up to $C\times$ faster than the original if C independent threads are available. The transformed core will have SMP semantics to the operating system.

Table 1 presents some metrics to compare the C-slow retimed core with traditional multithreading which does not employ any bypassing. Since the bypassing logic is retained, the transformed core resembles C interleaved *original* processors, resulting in a lower number of threads required to fully utilize the datapath (*saturation point*) and better single thread performance than the traditional multithreaded processors. The key is that performance in the transformed core depends on C and not on the number of pipeline stages N. In practice since $C << N$, it is more likely that a workload will contain C independent threads to execute in parallel than N threads.

5 Methodology

The experiment consisted of applying the C-slow retiming transformation to an actual SPARC processor core. The SPARC core chosen for modification was the LEON 1[5]. LEON 1 is a fully functional SPARC V8 compatible five stage pipeline processor core written in synthesizable VHDL capable of being targeted to various FPGA architectures and ASICs. The core was synthesized with Synplify, an HDL synthesis tool from Synplicity, for Xilinx Virtex[7] XCV800 FPGA with the built-in automatic retiming enabled.

The LEON VHDL source was modified to allow an arbitrary value of C for C-slow transformation. All registers were identified and replaced with special C-slowable registers. In addition to replacing all the registers, the caches and register files required modifications. The synthesis tool maps the HDL description of the caches and register file to Block RAM Virtex primitives on the FPGA. Due to deficiencies in Synplify's retiming algorithms enumerated in Section 6, manual retiming was necessary in order to improve the results. A counter was added to the design to identify the current thread in a given stage of the datapath and to tag memory requests.

6 Results

	Throughput (Mhz)			
	Multithread		Single Thread	
C	Absolute	Normalized	Absolute	Normalized
1	43	1.00	43	1.00
2	52.2	1.21	26.1	0.61
3	56.7	1.32	18.9	0.44

Table 2: A summary of performance results obtained after LEON transformations. The table shows increasing aggregate multithread throughput and resulting single thread performance penalty.

C	LUTs (logic)			FF		
	Absolute	Normalized	% of chip	Absolute	Normalized	% of chip
1	1222	1.00	6	602	1.00	3
2	1522	1.25	8	1366	2.27	7
3	1526	1.25	8	1926	3.20	10

Table 3: A summary of area cost associated with the performed transformations. Two main types of FPGA resources are look-up tables (LUTs) that implement arbitrary logic and edge-triggered flip-flops (FF) used for storage. As expected, the number of flip-flops increases approximately linearly with C, while the LUT count grows at a slower pace.

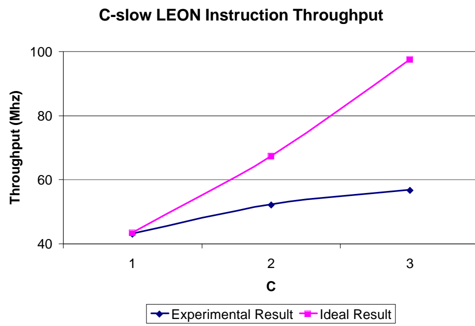

Figure 2: Measured instruction throughput on the transformed LEON.

6.1 Instruction Throughput

Both the C-slow and retiming transformations performed on the LEON core have a common property that they do not modify bypassing and control logic in a microprocessor core. Short of adding a simple instruction to identify a processor thread to the operating system, the ISA of the modified processor is identical to the original. These two factors, combined, result in a modified processor with the *identical* multithread IPC (Instructions per Cycles) measure for the same workload as the original core. This allows us to compare the instruction throughput of our modified cores by analyzing the maximum running frequency resulting from retiming.

The metrics presented include:

1. MultiThread Instructions per Time:

$$MT\ IPT(C) = \frac{IPC}{T_{cycle}(C)} = IPC \times F(C) \qquad (1)$$

   IPC is the Instruction per Cycle measure for our processor core. $T_{cycle}(C)$ and $F(C)$ are respectively the minimum clock cycle time and the maximum running frequency obtained form C-slow retiming transformations.

2. Single Thread Instructions per Time:

$$ST\ IPT(C) = \frac{IPC}{C \times T_{cycle}(C)} = \frac{IPC \times F(C)}{C} \qquad (2)$$

   Our approach does not employ dynamic thread scheduling, hence each thread receives exactly $\frac{1}{C}$ of the aggregate multithread instruction throughput.

The Figure 2 presents both the empirical and the ideal multithread instruction throughput. Both curves demonstrate the expected behavior in that they scale with C, however, our experimental results fall short of the ideal for the reasons discussed below. Nevertheless, empirical data shows approximately 21% multithread performance improvement on a 2-slow retimed core and over 30% improvement on the 3-slow version. Table 2 summarizes multithread and single thread instruction throughput from our experiment.

The LEON core has been developed in synthesizable VHDL, requiring appropriate tools and compilers to map the description into FPGA primitives. Storage elements (registers) must be identified in the circuit in order to perform C-slow transformation by hand. However, a wide "semantic gap" between the behavioral VHDL description and the netlist from the FPGA mapping process preclude effective hand retiming. The Synplify Pro synthesis tool, used in our experiments, supports retiming in Virtex FPGAs, although its algorithm is limited in several key aspects. Even without access to the retiming procedures in the Synplify, our experiments have identified the following deficiencies that limit the tool's ability to fully exploit performance opportunities of sequential circuit retiming.

- **Retiming across Block RAMs**
  LEON uses FPGA Block RAMs to implement the register file and the caches, which must be retimed in order to reduce the cycle time by a factor of C. Synplify treats the Block RAMs as "black boxes" (*i.e.* is not aware of their functionality) and hence conservatively does not move registers across these modules. We moved the registers by hand to solve this problem.
- **Retiming across registers with "clock enable"**
  "Clock enable" register inputs (or gated clocks) are frequently used to stall the processor pipeline. However, they

also seriously constrain the tool's ability to move registers across logic. We found no solution for this problem.

- **Optimization take precedence over retiming**
  As Synplify compiler makes multiple optimization passes over the code, the effective opportunities for retiming may disappear if the intermediate format transformations are applied in the wrong order. For example, if several registers connected in a shift chain are synthesized into a "shift register" primitive, the retiming algorithm cannot separate individual registers in order to "spread them out" across combinational logic. We solved this problem by re-coding pipeline registers to prevent the VHDL compiler from performing the optimization above.

We believe that these deficiencies, particularly "clock enabled" registers, are key constraining factors in Synplify retiming, and hence a possible reason why our experimentally obtained instruction throughput does not track the ideal well. These problems will be addressed in the future with our own retiming tool.

According to the static timing analysis in the Synplify, the critical path of 2 and 3-slow cores is dominated by bypass control logic (a chain of priority encoders and multiplexers) in the decode stage. Given the limitations of the Synplify's retiming algorithm, it is not clear whether bypass control logic is an actual critical path or a product of sub-optimal retiming transformation.

6.2 Area Cost

The target FPGA XCV800 contains two main types of resources: the look-up tables and the flip-flops. The first is a four input single output programmable table capable of implementing an arbitrary four input Boolean logic function. The latter is a simple flip-flop with asynchronous clear, reset, and preset. The FPGA components can be connected via programmable interconnect signal routing fabric.

Table 3 summarizes the area cost associated with the transformations performed in the experiment. The numbers match our expectations: the normalized number of flip-flops (storage that was replicated by C-slow) closely matches C, while the number of look-up tables (required logic) grew at a slower rate.

7 Future Work

This work has identified several key limitations of the retiming algorithm implemented in Synplify. However, to our best knowledge, no other commercial synthesis tool even supports retiming transformations. In the course of this work, we began working on our own post-synthesis retiming tool, intended to interface with the existing Xilinx tool flow.

Currently, the critical data structures, the retiming algorithm, and the basic C-slow transformations are complete. However, parsing through the proprietary Xilinx netlist format consisting of FPGA primitives has proven to be challenging and still requires more work. Once completed, the tool, in addition to retiming, will be able to intelligently C-slow shared state components (*i.e.* caches, instead of replicated, must be increased in size and additional address lines provided).

8 Conclusion

This work presents the application of C-slow and retiming transformations on a complete microprocessor core. C-slow and retiming are simple, well-defined, and amenable to automation with appropriate tools. If properly supported by synthesis, this combination is capable of dramatically increasing a throughput on pipelined designs with performance bound by feedback cycles. Applied to a processor, it yields a simple symmetric multithreaded core (or a SMP).

Ideally, the aggregate instruction throughput of the transformed processor is a factor of C larger than in the original design with a minimal single thread performance penalty. Since performing these transformations by hand is unrealistic and extremely error prone, they must be supported in the synthesis tools. Due to the Synplify's limitations outlined in Section 6, we were not able to obtain results that closely resemble the ideal. Nevertheless, we have shown over 21% percent improvement in instruction throughput on a 2-slow core, and over 32% on 3-slow. With our tool, we hope to have more control over the transformations and achieve superior results to Synplify.

References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the1990 International Conference on Supercomputing*, pages 1–6, 1990.

[2] Intel Corporation. Intel IXP 1200 Network Processor Family. intel.com/design/network/products/npfamily/ixp1200.htm.

[3] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, /1997.

[4] Leiserson and Saxe. Optimizing Synchronous Circuitry by Retiming. In *CTVLSI: Proceedings of the 3rd Caltech Conference on Very Large Scale Integration*, 1983.

[5] Gaisler Research. LEON SPARC-Compatible processor. http://www.gaisler.com/leonmain.html.

[6] B. Smith. Architecture and applications of the HEP multiprocessor computer system, 1981.

[7] Xilinx. Virtex Data Sheet Module Descriptions. http://www.xilinx.com/partinfo/ds003.htm.