What Limits the Operating Frequency of a Soft Processor Design

Kaveh Aasaraai and Andreas Moshovos
Electrical and Computer Engineering Department

University of Toronto, Canada
{aasaraai, moshovos}@eecg.utoronto.ca

Abstract

This work systematically explores what limits the operation frequency in a typical general purpose, soft processor design on a modern FPGA. The analysis mirrors a typical design cycle: It starts from a base implementation of a 5-stage pipelined core where correctness, modularity, and speed of development are the primary considerations. The analysis then proceeds in a series of identify-and-then-revise steps. At each step, the analysis identifies the critical path and then "removes" it. The result is a list of components and mechanisms that restrict the frequency of operation. A designer would have to cleverly redesign over these paths in order to improve the processor's operating clock frequency. Using the results of this analysis, this work proposes various optimizations to improve the efficiency of some of these components. The optimizations increase the processor clock frequency from 145MHz to 281MHz on Stratix III devices, while overall instruction processing throughput increases by 80%.

1. Introduction

General purpose soft processors are a key component in reconfigurable computing as they are easy to develop for, and excel at executing irregular applications. The need for soft processors has led to many designs both by the academic community and industry. For example, Altera's Nios II [1] and Xilinx's Microblaze [2] are two commonly used designs which provide adequate performance at a low cost.

Despite the popularity of soft processors and their widespread use, there has not been a systematic study of what limits their frequency of operation. The goal of this work is to present a systematic characterization of where speed is lost when implementing a commonly used soft processor architecture. We do not claim that the results of this characterization are previously unknown. Probably anyone that designed a soft processor has reached some similar conclusions at least partially. However, to the best of our knowledge this information has not been documented nor has there been a systematic approach at identifying these critical paths. Accordingly, it has to be rediscovered by anyone attempting a soft core design probably in an ad-hoc manner. Moreover, once a critical path cannot be removed, one typically cannot "see" further in what other critical paths lurk further ahead and thus may may miss opportunities that exist to further optimize the design.

In summary, this work makes two contributions: First, it systematically identifies the sources of clock speed inefficiency in a typical implementation of a 5-stage pipeline. This analysis focuses on operating frequency, identifying which components and mechanisms are the main sources of low clock speed. The result of the analysis is an ordered list of critical paths, along with the components and mechanisms incorporating them. Second, it proposes several architectural optimizations that improve the processor operating frequency and performance. The optimizations span over all pipeline stages, including delaying branch misprediction signaling and specializing forwarding paths. While these techniques are well known, they do serve to demonstrate that our analysis methodology can aid in the proper identification of the critical paths and help steer the designer in optimizing the design. These optimizations improve the processor's clock frequency from 145MHz to 281MHz. Overall, actual instruction processing throughput increases by 80%.

This work is a step toward systematically understanding and documenting the sources of inefficiency in soft processor designs. Future work may rely on the analysis presented here to improve soft processor designs and may follow a similar methodology to characterize other soft processor designs and architectures.

The rest of the paper is organized as follows. Section 2 discusses the critical path identification methodology. Section 3 details the experimental methodology of this study. Section 4 presents the baseline processor design. Section 5 presents the critical path analysis while Section 6 proposes several optimizations. Section 7 measures processor's overall performance with various optimizations. Section 8 reviews related work and Section 9 concludes this work.

2. Identifying Speed Inefficiencies

The ultimate goal of this work is to provide a processor designer with insight what limits performance on an FPGA, before rethinking the soft architecture. This work takes an iterative approach in which it starts with a typical pipelined processor implementation, and at every step only a few, preferably one, component or mechanism is investigated and improved. Once, and if, the current component is improved, another would now be the dominant one and the process can be repeated.

This work uses a best-effort approach where, at each step, it artificially removes the dominant component from the processor cycle time. A challenge we faced during the analysis is

978-1-4799-5944-0/14/$31.00

whether it is reasonable to eliminate the dominant components at each step. Eliminating the components would soon result in a virtually empty design. Instead, we used a number of heuristics that kept the components within the design but removed their constraints on the clock cycle (e.g., signals were artificially pipelined). Section 5 explains the elimination heuristics used on a case-by-case basis and further discusses the limitations and implications of this approach.

A limitation of the presented analysis is that actual optimizations may alter the relative importance of the various circuit paths or may give rise to other critical paths. However, we believe that the presented analysis represents a meaningful and useful approach in identifying the sources of inefficiency in FPGA-based designs in lieu of actual optimizations – the alternative would be to not even try.

3. Methodology

The entire processor under study is implemented in Verilog. The processor implements the Nios II ISA [1]. The processor's functionality was tested by running micro-benchmarks and by booting ucLinux [4] and running several SPECINT CPU2006 workloads [5].

All Verilog designs were synthesized using Quartus II 12.1 on a Stratix III chip. The TimeQuest timing analyzer measured the maximum clock frequency which the design can operate at. The target clock speed is set to 333MHz (3ns period) in the design constraint file. Our goal is to reach frequencies close to that of Nios II-f, which is 270MHz on Stratix III devices [6].

There can be many different interfaces and devices the processor may connect to. To identify the critical paths that are inherent to the processor design and to avoid artifacts caused by external components, the processor design is isolated for placement and routing. This is done by synthesizing the processor in a top level module containing only the processor. In order to reduce the effect of pin placement on the clock frequency (e.g., due to excessive global routing), all the processor's inputs and outputs are registered. These include the instruction and data busses, interrupt lines, and clock and reset signals. To minimize the number of pins used, all inputs are fed with shift registers. All wide outputs (e.g., data bus writedata) are reduced to one bit signals with XOR reduction operations.

This work measures processor performance in Instructions Per Cycle (IPC) and Instructions Per Second (IPS). To gather IPC data for various architectures, we simulate the processor architecture using a custom, cycle-accurate full-system Nios II simulator. After booting ucLinux, the simulator runs several of the SPECINT CPU2006 benchmarks. IPC data is gathered over one billion instruction runs, after skipping one billion initial instructions.

3.1. Critical Component Identification

We use the synthesis tool to extract the current critical path of the design after placement and routing. We trace the path to identify the components that it includes and report those as the critical components.

Most critical paths are tightly coupled with other parallel paths. However, our analysis focuses on the top failing path reported by the synthesis tool. Once the critical component and/or mechanism is identified, we refrain from proposing a solution right away and artificially eliminate it, for example, by introducing registers along the path.By artificially removing critical components, we eliminate sources of inefficiency by adding/removing as little logic as possible. Although the remaining processor design may not be functionally correct anymore, at this stage of this study we are focusing on identification of the sources of inefficiency.

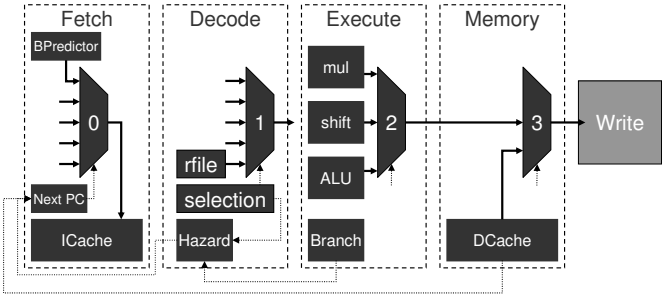

Fig. 1. The baseline 5-stage pipeline implemented in this work. Dotted lines represent control signals.

4. Processor Pipeline

This work implements a classic 5-stage processor pipeline in Verilog. Figure 1 shows the block diagram of the processor. The baseline implementation focuses on correctness, modularity and extensibility. This section describes the implementation of each pipeline stage.

4.1. Fetch Stage

The Fetch stage includes an instruction cache, capable of fetching one instruction per cycle if the address hits in the cache. This stage also includes a bimodal branch predictor [7]. In addition, a Branch Target Buffer (BTB) predicts the target address for taken branches [7]. The predictor resembles the structure of the predictor presented in [8] as it has been shown to provide storage and frequency advantages on FPGAs.

4.2. Decode Stage

The Decode stage is responsible for preparing all data and control signals for the Execute stage. Depending on the instruction type and pipeline state, data operands may come from the register file, an immediate value from the instruction bits, or forwarded from the later stages of the pipeline to avoid bubbles.

The Decode stage is also responsible for detecting hazards in the pipeline. Hazards can occur for multiple reasons, for example, if an instruction requires a data yet to be produced in the pipeline. When a hazard is detected, the pipeline is stalled.

4.3. Execute Stage

The Execute stage computes the result of arithmetic instructions. In addition, it calculates the target address of branch instructions. The calculated branch target is compared to the predicted target address which was provided by the branch predictor during Fetch. A misprediction signal is broadcast to all earlier pipeline stages if the two addresses don't match, and the pipeline is flushed in the same clock cycle.

4.4. Memory Stage

The Memory stage includes a data cache, which is acccessed by load and store instructions. Loads complete in a single cycle, while stores take two cycles to complete. The data cache is a 2KB blocking, write-back cache [9]. It consists of two storage units implemented using BRAMs, one for tags and one for data. For all non-memory instructions, the memory stage is a pass-through stage.

4.5. Writeback Stage

The Writeback stage writes the result of instructions back to the register file.

5. Critical Component Analysis

In this section we iteratively identify critical components of the processor design through critical path analysis. The goal of this analysis is to identify various critical components in lieu of actual optimizations. Table 1 reports the list of critical components found in 15 iterations. The table reports the maximum operating frequency with the corresponding path included. The baseline processor design can operate at 145Mhz. If it was possible to eliminate all the reported paths, the processor would operate at 281.68Mhz. We stop the analysis after the 15th iteration as the maximum clock frequency reached (281 MHz) is higher than our target frequency (270 MHz).

Removing most paths results in a monotonic increase in the operating frequency except between paths (D) and (E). Removing path (D) results in an isolated efficient routing configuration, mainly caused by the random nature of the placement and routing process. We conclude that the list of the various paths is more important rather than their relative order. The results also reconfirm that there is no single path that, if eliminated, would result in a significant improvement in clock frequency. Instead, the designer has to contend with multiple, tightly coupled paths.

A: This path includes the multiplier and forwarding data path. It starts from the data operand registers provided by the Decode stage, through the multiplier in the Execute stage, routed back through the forwarding logic to the Decode stage and ends at the data operand registers. This represents data computation and communication. In order to remove this path we registered the output of the multiplier, and allowed bypass only from the memory stage.

B: This path includes the branch misprediction logic and pipeline redirection. It starts from the data operands to the Execute stage, and continues in the ALU's comparator for branch outcome determination. It also includes the address comparator for misprediction identification which signals the Fetch stage to redirect the program counter. This path is for branch misprediction identification followed by fetch stage redirection. We removed this path by registering the branch mispredict signal broadcast to the Fetch stage. This effectively delays branch misprediction detection by one cycle.

C: The third critical path includes the branch misprediction logic and stall signal sent to the Decode stage. When a branch is identified as mispredicted at the Execute stage, the instruction currently at the Decode stage must be annulled. To remove this path we registered the branch outcome signal. This signal determines whether a branch is taken or not-taken. Similar to path (C), branch misprediction identification is delayed by one cycle.

D: The fourth critical path includes the shifter in the ALU and the forwarding logic. It starts from the data operand registers, follows through the shifter and forwarding logic back to the data operand registers. This is another data computation and communication path. We register the output of the shifter to eliminate this path. This effectively eliminates one bypass path from the Execute stage back to the Decode stage.

E: The hazard detection logic in the Decode stage dominates the fifth path. The hazard signal is broadcast to the Fetch stage where it stalls the fetch process. We register the

TABLE 1. Processor critical paths.

Path	Max. Freq. (MHz)	Main Component	Type
A	144.99	Multiplier	Data
B	184.71	Branch	Control
C	199.72	Branch	Control
D	211.01	Shifter	Data
E	200.84	Hazard Detection	Control
F	201.90	Memory Stalls	Control
G	206.95	Hazard Detection	Control
H	211.46	Forwarding	Control
I	214.68	Forwarding	Data
J	230.95	ICache Hit	Control
K	231.59	Forwarding	Data
L	242.72	Multiplier	Data
M	249.50	ICache Hit	Control
N	249.75	DCache Hit	Control
O	281.69	Memory Mux	Data

fetch redirect signal to remove this path.

F: This path includes the stall signal from the Memory stage to the rest of the pipeline. We remove this path by registering the memory stall signal.

G: This is another hazard detection dominated path. Hazards are identified in the Decode stage by checking all forwarding lines. We register the forwarding selection logic signals to remove this path.

H: The next critical path is in the data path including the forwarding data lines from the Memory stage to the Decode stage and ending in the data operand registers. We reduced the data operand multiplexer size by removing one of the inputs (immediate value for shift operations).

I: The critical path is still through the forwarding logic from the Memory stage to the Decode stage. We remove two more inputs from the data multiplexers in the Memory stage (shift and multiplication results).

J: The instruction cache hit signal contributes to this path. This signal directs the address multiplexer in the Fetch stage to select the next instruction address. We remove this path by eliminating one input to the multiplexer.

K: The forwarding logic from the Memory stage to the Decode stage surfaces again. This path includes the sign extension logic required after loading the data from the data cache. We remove this path by eliminating load instructions from the forwarding logic.

L: At this point the multiplier alone is the critical path. Both the inputs and the output of the multiplier are registered. We remove this path by replacing the multiplier with a simple XOR logic.

M: The instruction cache hit signal to fetch address selection path surfaces again. We remove this path by registering the ready signal from instruction cache to the Fetch stage.

N: This path includes the data cache's lookup address selection logic. The lookup address is either from load/store instructions or from the write-back logic. The selection depends on the cache's next state. We remove this path by using the current stage (a register) to select the address.

O: This path includes the multiplexer to select between shift, multiplication, loads from data cache or all other instruction results in the Memory stage. The result is passed on to the Writeback stage.

The results of this analysis show there is no single path that dominates the clock frequency. Instead, removing each problematic path results in a relatively small improvement. Only if several paths are eliminated, operating frequency can improve substantially.

6. Improving Critical Components

This section proposes solutions to improve the critical components that Section 5 identified. All solutions proposed are applicable to the processor design at the architecture level. That is, they are compiler independent and only change the implementation of the processor. No compiler options are changed during optimizations. The proposed solutions preserve the processor's functionality while increasing its clock frequency. Some of the proposed optimizations increase clock frequency at the expense of introducing pipeline bubbles under certain scenarios. Such bubbles may delay certain instructions, leading to lower IPCs. However, as long as they are sufficiently infrequent, the gain in frequency can compensate for the loss in IPC. Section 7 measures the resulting performance in IPS, considering both IPC and clock frequency.

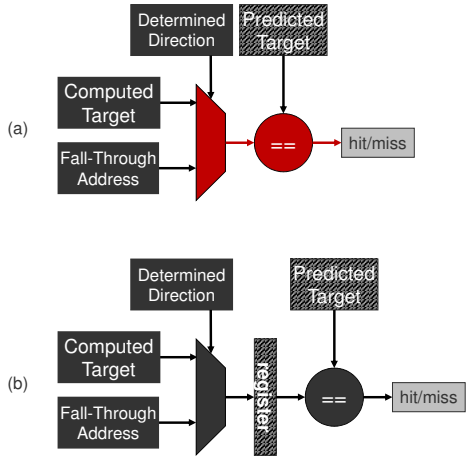

Fig. 2. Branch misprediction detection before (a) and after (b) optimization. Dashed boxes represent registers.

6.1. Multiplier and Shifter

The original processor implementation included a multiplier and a shifter in the Execute stage. Although this reduces the number of cycles required for multiplication and shifting, it also leads to low clock frequency and is manifested as critical paths (A), (D) and (L). Instead, we can delay the forwarding of multiplication and shifting operations in the pipeline by eliminating the bypass path from the execute stage back to the decode stage. This will introduce bubbles in the pipeline when the next in order instruction in the pipeline requires the result of the multiplier or shifter.

6.2. Branch Misprediction Detection

When a predicted branch's actual outcome is computed in the Execute stage, it must be compared to the one predicted earlier in the Fetch stage. If a mismatch is detected, any incorrectly introduced instructions must be flushed from the pipeline and fetching must be redirected to the computed target address.

Branch misprediction detection includes three steps: 1) The outcome of the branch is determined. 2) The target address of the branch is calculated. 3) The actual target of the branch is compared to that of predicted address. Figure 2-a shows the block diagram of this mechanism.

In order to shorten the long combinatorial paths (B) and (C), we can delay the branch misprediction detection by one clock cycle. As shown in Figure 2-b, branch outcome and target are calculated in the first clock cycle (Execute stage), and the comparison with the predicted target occurs in the next clock cycle (Memory stage). This shortens the combinatorial path by introducing a register in the path. This optimization increases branch misprediction recovery time by one clock cycle.

6.3. Data Forwarding

A full-blown forwarding logic, as show in Figure 3-a, forwards data from every pipeline stage after the Execute stage, and requires a large multiplexer. Furthermore, the selection logic for the forwarding multiplexer proves to be relatively complex. We improve the data forwarding delay with the two optimizations described next.

6.3.1. Decoupling Forwarding Planning and Communication. The most critical path in data forwarding is due to the selection logic manifested in path (G). This logic is large and performs various operations sequentially. We shorten this long combinatorial path using the following observation: Data source identification and data selection don't have to occur at the same cycle. Instead, they can be performed in two separate cycles. In the first cycle, the forwarding logic can determine the source for a particular data operand. In the next clock cycle, the actual data selection occurs. This scheme effectively cuts the long path of data forwarding into two smaller paths.

6.3.2. Delayed Data Forwarding. Delaying multiplication and shift operations by one cycle requires forwarding their result from the Memory stage to the Decode stage as Figure 3-a shows. Combined with the loads and ALU instructions, this requires a 4-to-1 multiplexer in the Memory stage. This multiplexer manifests in critical path (I) since it resides directly in the forwarding path. We can delay multiplication and shifting results by one more cycle and forward them from the Writeback stage. This reduces the multiplexer size down to 2-to-1.

As Figure 3-a shows, load data from memory passes through a sign extension logic. This further prolongs the forwarding path manifested in path (K). We can remove load data forwarding to the Decode stage, therefore eliminating the multiplexer in the Memory stage altogether. This further shortens the forwarding data path as Figure 3-b shows. Both optimizations may delay certain instruction combinations.

6.4. Fetch Address Selection

While the baseline Fetch stage uses the branch predictor to guess the next instruction address it does not have to do so in all cases. More specifically, there are five options for the next instruction address: **(A1)** Reset vector, **(A2)** IRQ vector, **(A3)** Redirect address due to branch misprediction, **(A4)** Current PC due to instruction cache miss, and **(A5)** The predicted next address by the branch predictor.

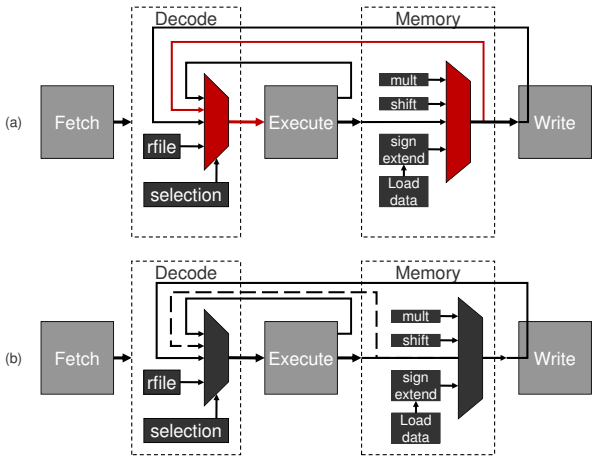

Fig. 3. Forwarding data path before and after optimization in the pipeline. Dashed lines are removed/added forwarding paths.

These options lead to a large 32-bit 5-to-1 multiplexer in the Fetch stage. Furthermore, the select signal depends on the following control signals: **reset, interrupt, branch misprediction, instruction cache miss, data hazard, and memory stall**. Having a large number of combinatorial signals as inputs, the multiplexer in the Fetch stage gives rise to paths (E) and (J).

We can reduce the size of multiplexer to 3-to-1 and reach a reduced combinatorial path as follows: We observe that A1-A3 are redirection addresses. In addition, we expect that A5 will be the common case, with A4 being less common and A1-A3 occurring infrequently. Accordingly, we propose delaying options A1, A2, and A3 by one clock cycle. We introduce a *redirect address* register, holding the redirection address, selected among options A1, A2, and A3. We use the redirect register to steer the fetch accordingly in the next cycle.

One can also include option A4 in the redirect register by observing that if the Fetch stage is allowed to advance the PC even if the instruction cache misses, returning back to the previous fetch address can be treated as a redirection. Therefore, we can remove the instruction cache miss signal from the multiplexer select input. Figure 4 shows this scheme in detail.

6.5. Data Operand Specialization

In the Nios II ISA the second operand for shift/rotate operations can come from only two sources: the register file or an immediate value from the instruction bits. However, other instruction types have four options for the second operand. The original, modular Verilog code of our processor implementation included all possible data sources for all types of instructions. However, it is not necessary to use the same data multiplexer for all instruction types. We can use a separate

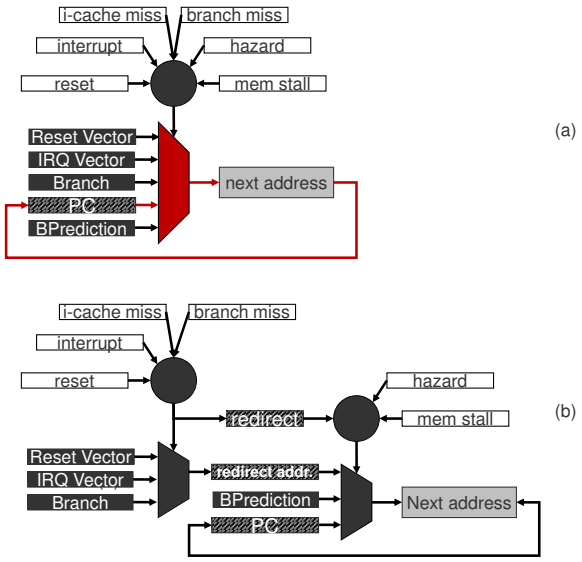

Fig. 4. Next address selection data path in the Fetch state before (a) and after (b) optimization. Dashed boxes represent registers.

2-to-1 multiplexer for shift/rotate instructions, shortening path (H).

7. Performance

The optimizations presented in Section 6 improve processor clock speed but may increase the number of pipeline stalls. Overall processor performance depends on both the IPC rate and the clock frequency. This section studies the performance of the processor pipeline taking both into account. Figure 5 reports IPC along with the IPS throughput for the various processor designs shown along the x-axis. The baseline configuration is shown at the leftmost side. From left to right, the graph reports instruction throughput as all paths listed along the x-axis are removed. For example, configuration I has paths A through E and I removed. The IPS results show that frequency gains due to optimizations more than compensate for loss in IPC. Processor performance starts at 47 million IPS and reaches as high as 85 million IPS after applying the optimizations, an 80% improvement.

8. Related Work

To the best of our knowledge no previous work exists that systematically characterizes the critical paths in a general purpose soft processor implementation. The closest work is by Wong et. al., who compare the area and delay of processors implemented on custom CMOS and FPGA substrates [10]. They find that SRAMs and adders are efficient on FPGAs mainly due to having dedicated resources. However, CAMs and multiplexers are found to be extremely inefficient. They also find that data forwarding is inefficient on FPGAs compared to custom CMOS implementations. Yiannacouras et. al.,

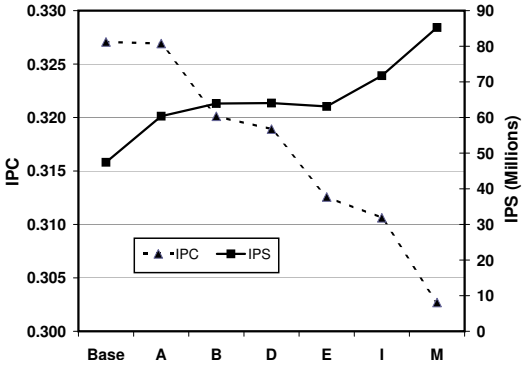

Fig. 5. IPC and relative IPS improvement for the processor after removing critical paths.

explore the impact of soft processor customization on its performance [11]. They consider pipeline depth and organization, data forwarding, multi-cycle operations and ISA subsetting. They show that fine grain microarchitectural customizations can yield higher overall performance compared to a few hand-picked optimizations.

9. Conclusion

This work considered a typical pipelined processor design and implemented it on a modern FPGA. It followed a systematic way to analyze which components limited operation frequency. It found that major components contributing to its low speed were branch misprediction detection, data forwarding, fetch address selection, certain computations, and memory stalls. Finally this work proposed various optimizations to improve processor clock speed and achieve high performance.

References

[1] *NIOS II Processor Reference Handbook*. Altera, 2014.
[2] *MicroBlaze Processor Reference Guide*, Xilinx Inc., Mar. 2012.
[3] E. S. Research and T. Centre, "Leon3 multiprocessing cpu core," http://www.gaisler.com/doc/leon3_product_sheet.pdf/.
[4] Arcturus Networks Inc., "uClinux," http://www.uclinux.org/.
[5] Standard Performance Evaluation Corporation, "SPEC CPU 2006," http://www.spec.org/cpu2006/.
[6] *Nios II Performance Benchmarks*, Altera Corporation, Dec. 2012.
[7] J. E. Smith, "A study of branch prediction strategies," in *8th Annual Symposium on Computer Architecture, pages 135-147*, June 1981.
[8] D. Wu, K. Aasaraai, and A. Moshovos, "Low-cost, high-performance branch predictors for soft processors," in *23rd International Conference on Field Programmable Logic and Applications (FPL)*, September 2013.
[9] N. P. Jouppi, "Cache write policies and performance," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 191–201. [Online]. Available: http://doi.acm.org/10.1145/165123.165154
[10] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. custom cmos and the impact on processor microarchitecture," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950419
[11] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and customization of fpga-based soft processors," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 266–277, 2007.