High Performance Branch Predictors For Soft Processors

by

Di Wu

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

High Performance Branch Predictors For Soft Processors

Di Wu
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2014

Branch prediction has been extensively studied in the context of application specific custom logic (ASIC) implementations. Since the tradeoffs are different for reconfigurable logic, naïvely porting ASIC-based branch predictors to FPGAs may prove slow and/or resource-inefficient. Accordingly, this work studies the FPGA implementation of several commonly used branch predictor designs and does so in the context of simple pipelined processors, the most commonly used general purpose soft processor architecture due to its excellent balance of performance and resource cost. For this purpose, it assumes a pipelined processor implementation representative of Altera's Nios II-f and investigates the performance and resource cost of various branch predictors. The analysis confirms that existing designs are not efficient nor high-performing on reconfigurable logic. Accordingly, this work proposes FPGA-specific modifications that improve accuracy, resource cost, or both.

Acknowledgements

TODO: put acknoledgements here
????????

Contents

List of Tables

List of Figures

Chapter 1

Introduction

FPGA-based designs often incorporate one or more general purpose soft processors. As the range of FPGA applications broadens and evolves, it is likely that the performance demand from soft processors will increase. A key performance enhancing technique that even simple general purpose processors use is branch prediction. Without branch prediction, a branch has to execute completely before the processor can fetch the instructions that follow. Branch prediction eliminates these stalls by guessing the target address of branches. Current state-of-the-art branch prediction techniques, e.g., TAGE [2], rely on dynamically collected information about past branch behaviour. Such techniques have been proven to be very effective even in deeply pipelined, highly speculative, high-performance custom processor designs.

Branch prediction has been extensively studied in the context of application specific custom logic (ASIC) implementations. Since the tradeoffs are different for reconfigurable logic, naïvely porting ASIC-based branch predictors to FPGAs may prove slow and/or resource-inefficient. Accordingly, this work studies the FPGA implementation of several commonly used branch predictor designs and does so in the context of simple pipelined processors, the most commonly used general purpose soft processor architecture due to its excellent balance of performance and resource cost. For this purpose, it assumes a pipelined processor implementation representative of Altera's Nios II-f and investigates the performance and resource cost of various branch predictors. The analysis confirms that existing designs are not efficient nor high-performing on reconfigurable logic. Accordingly, this work proposes FPGA-specific modifications that improve accuracy, resource cost, or both.

This work proposes a minimalistic single-cycle branch predictor for Nios II-f. In more detail, this work makes the following contributions: (1) It studies the FPGA-implementation of Branch Target Buffers (BTB), including designs that fuse the BTB and the direction predictor and shows that, contrary to ASIC implementations, it is best to avoid a BTB and instead to calculate branch target addresses on-the-fly. (2) It studies the FPGA implementation of the three most commonly used branch direction predictors (DIR): bimodal [8], gshare, and gselect [9]. The analysis corroborates the results of past studies showing that gshare achieves the best accuracy among the three for practical table sizes, but also shows that unlike an ASIC implementation, frequency suffers with gshare on FPGAs. It proposes *gRselect*, an FPGA-friendly gselect implementation that uses a simple indexing scheme to outperform gshare by 11.4%. (3) It demonstrates that a conventional Return Address Stack (RAS) maps well onto the MLABs of FPGAs improving performance with little additional cost.

Other more accurate, albeit more elaborate predictors exist, such as perceptron [7] and TAGE [2].

These predictors often use multiple tables and tagged entries, which require comparator-driven multiplexers and thus may not map well onto FPGAs. As a result, single-cycle accesses may not be possible for these predictors. This work also considers to increase the number of pipeline stages in Nios II-f, use a simple bimodal predictor to provide a baseline prediction in a single cycle, and then the more elaborate branch predictors as overriding predictors. It shows that XXXXXXXXXXXXX????????????????????????????????????

Chapter 2

Background and Goals

This chapter covers the background of branch prediction and the goal of this thesis.

2.1 Why Branch Prediction?

In modern microprocessors, instruction pipelining is a key feature to improve performances. At each cycle, a new instruction is read from the memory at the location indicated by the program counter (PC). The pipeline works ideally when there is no transfer of control, i.e., PC is incremented by a constant at each cycle to fetch the next instruction. However, when there is a transfer of control (e.g. a taken branch instruction), the next instruction to fetch is unknown at this point. The processor has to stall the pipeline to wait until the branch is resolved to fetch the correct instruction, hence severely impacts performance.

Branch prediction is a technique to improve performance by guessing the branch target (i.e. the next instruction) to allow the processor to run speculatively. The processor fetches the predicted target, and compares the prediction with the actual target when the branch is resolved several cycles later. If the prediction was correct, the processor fetched the correct instructions without stalling the pipeline, as if the instruction stream were not disrupted. On the other hand, if the prediction was incorrect, the processor fetched wrong instructions, and the intermediate results related to those wrong instructions has to be squashed, possibly with an extra maintenance penalty. Therefore, prediction accuracy is the key to high performance.

2.2 Branch Prediction Scheme

Branch prediction has two aspects, i.e., direction prediction and target prediction. Branch direction prediction predicts whether a branch instruction will be taken or not, while branch target prediction predicts the target instruction address if the branch is taken.

Fig. 2.1 shows the organization of a typical branch predictor comprising: (1) a direction predictor (DIR), (2) a Branch Target Buffer (BTB), and (3) a Return Address Stack (RAS).

The predictor operates in the fetch stage where it aims to predict the program counter (PC), that is the address in memory, of the instruction to fetch in the next cycle using the current instruction's PC and other dynamically collected information.

The DIR guesses whether the branch will be taken or not. The BTB and the RAS guess the address for predicted as taken branches and function returns respectively. The multiplexer at the end selects based on the branch type and the direction prediction whether the target is the fall through address (PC+4 in Nios II), the target predicted by the BTB, or the target provided by the RAS. Since, at this point in time, the actual instruction is not available in a typical ASIC implementation, it is not directly possible to determine whether the instruction is a return, a branch, or some other instruction. Accordingly, a Selection Logic block uses either pre-decode information or a PC-based, dynamically populated lookup table to guess which target is best to use. With the latter scheme, when no entry exists in the lookup table, some default action is taken until the first time a branch is encountered. Once the branch executes, its type is stored in the lookup table where it serves to identify the branch type on subsequent encounters. This scheme is not perfectly accurate due to aliasing.

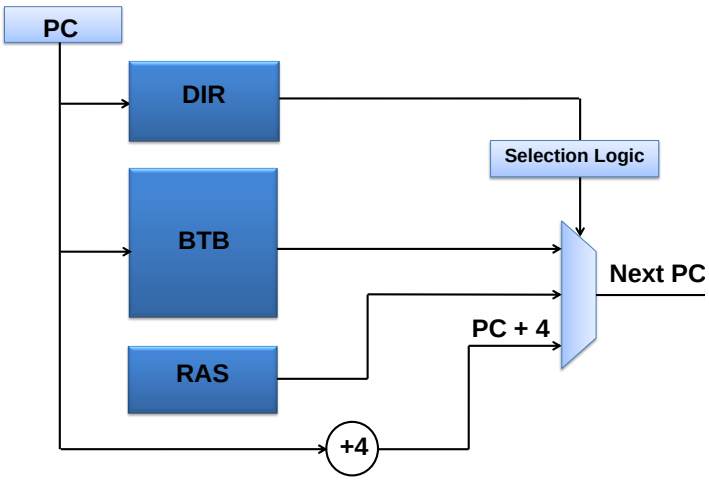

Figure 2.1: Canonical Branch Predictor.

2.3 Branch Target Predictors

Branch Target Prediction usually requires a Branch Target Buffer (BTB), a cache-like structure that records the addresses of the branches and the target addresses associated with them. If a branch is predicted to be taken and there is also a BTB hit, then the next PC is set to be the predicted target. A BTB can also have set-associativity to reduce the impact from aliasing.

Another common structure for branch target prediction is a Return Address Stack (RAS). It is a stack-like structure that accurately predicts the target address of function returns. When a call instruction executes, the return address of that call is pushed on RAS. When the processor executes the corresponding return instruction, RAS pop the return address and always provides the accurate prediction. Most modern processors have a shallow RAS because typical programs generally do not have very deep call depths. RAS will fail to provide correct target prediction if it is overflowed.

2.4 Branch Direction Predictors

This section describes the branch direction predictors considered in this thesis.

2.5 Static Predictor

Static branch direction predictors statically predicts a branch to be taken or non-taken. This strategy is based on the observation that most branches are taken. The main weakness of static branch predictors is its inability to adapt to various applications. There are some other variations of static branch predictors such as "backward-taken forward-not-taken". However, the improvements in accuracy is not significant. As the pipeline in modern processors becomes deeper, the misprediction penalties increases, which necessitates more accurate *dynamic* branch predictors.

2.5.1 Bimodal, Gshare and Gselect Predictor

Bimodal, Gshare and Gselect [9] are some of the simplest dynamic direction prediction schemes. Fig. 2.2 shows the organization of these three predictors. Each of these predictors has a *Pattern History Table* (PHT). The entries of a PHT are called *2-bit saturating counters*, which are used to record the directions of the previously seen branches. The 2-bit saturating counters are incremented/decremented when their corresponding branch is taken/not taken, and the high order bit is used as direction prediction.

The PHT in bimodal is indexed by selected bits of PC. Since not all bits of PC are used to index the PHT, multiple different branches can be assigned to the same 2-bit saturating counter. This *aliasing* can be destructive to the direction records, hence degrades performance.

Gshare and gselect utilize a structure called a *Global History Register* (GHR). The GHR is a shift register that stores recently resolved branch directions. The only difference between bimodal, gshare and gselect is the indexing of the PHTs. Bimodal uses partial PC, gshare uses partial PC XORed with GHR, and gselect uses partial PC concatenated with GHR. Gshare and gselect has better accuracy over bimodal because it reduces aliasing by correlating local history (i.e. PC) and global history (i.e. GHR).

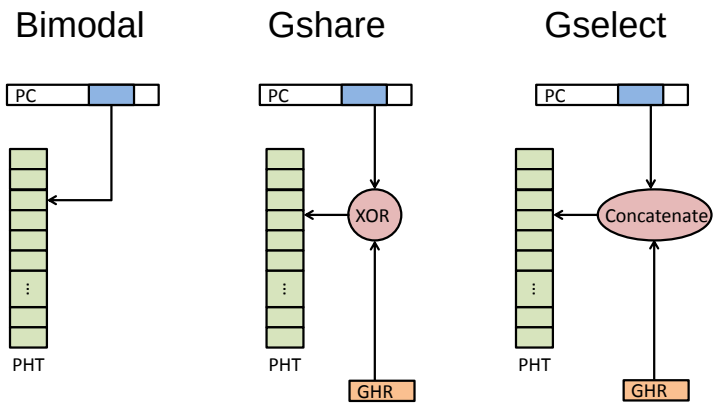

Figure 2.2: Bimodal, Gshare and Gselect.

2.5.2 Perceptron Predictor

The perceptron predictor use vectors of weights (i.e., perceptrons) to represent correlations among branch instructions [7]. Fig. 2.3 shows the structure of a perceptron predictor.

Perceptron uses a table to store vectors of weights. When making a prediction, a weight vector is loaded from the table, and each weight is multiplied by the corresponding global branch history (i.e. 1 if taken and -1 if not-taken) from the GHR. Finally, the multiplication results are summed up, and perceptron predicts taken if the sum is positive, and not taken otherwise.

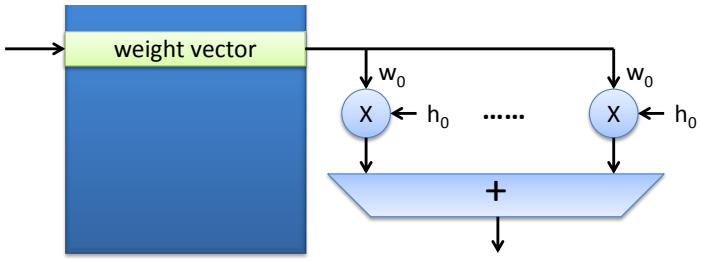

Figure 2.3: The preceptron branch predictor.

2.5.3 TAgged GEometric history length Branch Predictor (TAGE)

The TAGE predictor features a bimodal predictor as base predictor T_0 to provide a basic prediction and a set of M tagged predictor components T_i [2]. These tagged predictor components T_i, where $1 \leq i \leq M$, are indexed with hash functions of the branch address and the global branch/path history with various length. The global history lengths used for computing the indexing functions for tables T_i form a geometric series, i.e., $L(i) = (int)(\alpha^{i-1} \times L(1) + 0.5)$. TAGE achieves its high accuracy by utilizing very long history lengths

Fig. 2.4 shows a 5-component TAGE predictor. Each table entry has a 3-bit saturating counter *ctr* for prediction result, a *tag*, and a 2-bit useful counter *u*. The indices of the tables are produced by hashing PC and global history with various lengths. A valid prediction result from each table is provided only on a tag match (i.e. a hit). The final prediction of TAGE comes from the hitting tagged predictor component that uses the longest history.

2.6 Field Programmable Gate Arrays (FPGAs) and Soft Processors

Field Programmable Gate Arrays (FPGAs) are chips that can be reconfigured by designers. Modern FPGAs such as Altera's Stratix IV [3] have large number of logic blocks connected by reconfigurable interconnects. The logic blocks usually consist of Look-up Tables (LUTs), registers and Block RAMs (BRAMs) etc. Designers program digital circuits with Hardware Description Languages (HDLs) such as Verilog and VHDL, and then use CAD tools provided by the FPGA vendors to synthesize the design.

FPGAs can be reconfigured to fit specific requirements of various applications, therefore they are popular choices for hardware acceleration. Due to the increasing cost and time of designing ASIC, an increasing number of embedded systems are being built using FPGA platforms [12]. These systems

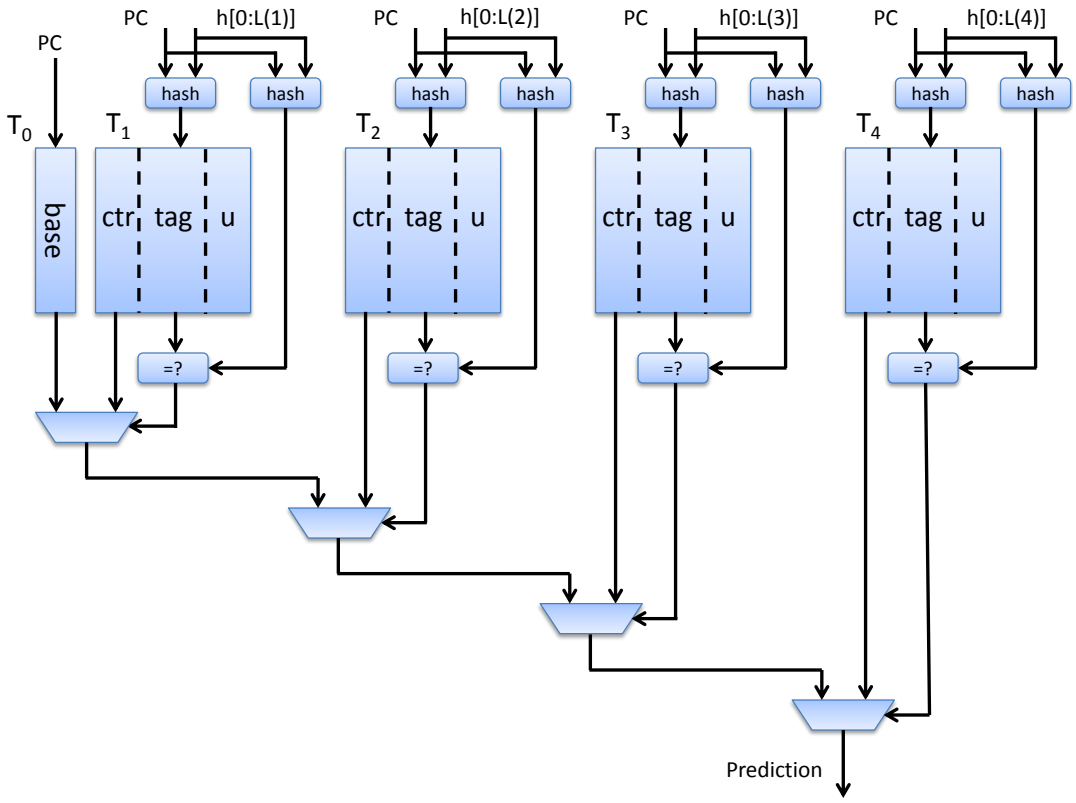

Figure 2.4: A 5-component TAGE branch predictor.

usually contain one or more embedded processors, necessitates high-performance FPGAs-based *soft processors*.

Since an FPGA platform is significantly different than ASIC, hard and soft processors have distinct design tradeoffs. Commercial soft processors such as Alteras's Nios II-f [6] and Xilinx's MicroBlaze [11] usually have shallow pipelines, mostly because the relative speed gap between memory and logic is much smaller than on ASIC. Accordingly, branch predictor designs for soft processors also have to be re-evaluated.

2.7 Design Goals

This work aims to design a branch predictor that (1) balances operating frequency and accuracy to maximize execution performance, and (2) uses as few on-chip resources as possible.

This work proposes a minimalistic branch predictor for Altera's highest performing soft-processor Nios II-f. Since Nios II-f uses three BRAMs in total and only one BRAM is used for branch prediction [6], the proposed minimalistic branch predictor has a limited resource budget of one BRAM; each additional BRAM would represent a more than 1/3 overhead in terms of BRAM resources. The design of the minimalistic predictor considers the most commonly used direction predictors, bimodal, gshare and gselect. These predictors use a single lookup table and map relatively well onto a single BRAM.

This work also includes an investigation of the state-of-the-art perceptron and TAGE predictors. These more elaborated branch predictors are generally more accurate, but often cannot be accessed

within a single cycle. Therefore, this work considers an overriding predictor that uses a simple bimodal predictor to provide base prediction, and then a more accurate predictor to override the base prediction if it disagrees with the bimodal predictor.

Chapter 3

The Minimalistic FPGA-Friendly Branch Predictor

This chapter discusses the architecture of the various minimalistic FPGA-friendly branch predictors considered. Section 3.1 discusses target prediction, while Section 3.2 discusses direction prediction.

3.1 Target Prediction

A conventional method to predict branch targets is to use Branch Target Buffers (BTB). A BTB is a table that caches branch target addresses. When a branch executes for the first time, the BTB stores the target address so that it can be used on subsequent encounters of the branch. Ideally, the BTB would be large enough so that each branch can use a separate entry. In a practical implementation, however, aliasing will occur reducing prediction accuracy.

The simplest BTB design does not use an address tag per entry and directly predicts the target address for all instructions. Not using a tag results in a fast design that uses one direct SRAM lookup. In addition, not filtering non-branch instructions is desirable since at the time of access the instruction opcode is not available. Unfortunately, as Section 5.2 shows, when all instructions use the BTB, high destructive aliasing results in poor accuracy. To reduce aliasing, a small decode logic can prevent non-branches from updating the BTB. However, as Section 5.4 shows, while this solution increases target prediction accuracy by 30%, the additional logic reduces the maximum frequency by 35%. An alternative is to calculate the target address during the fetch cycle.

3.1.1 Target Address Pre-calculation

ASIC processor implementations use a BTB since the cache latency dominates the clock cycle leaving no room for further action. This is not true in an FPGA implementation where memory is generally faster than logic. This creates an opportunity to pre-calculate the target address for branches and thus eliminate the BTB. In this scheme the processor fetches the instruction from the cache and then, during the same cycle, calculates the instruction's taken address. As an added benefit, address pre-calculation may improve accuracy since, if possible, it is always correct. Unfortunately, it is not possible to pre-calculate the target address for all branches. The Nios II ISA includes two types of branches: *direct*

and *indirect*. The target of a direct branch can be calculated using the current PC and an offset that is embedded in the instruction. Indirect branch targets are read from the register file.

This work proposes enhancing target prediction with *Full Address Calculation* (FAC), which as Fig. 3.1 shows, calculates the target address for all direct branches and uses a BTB or some other storage for indirect branches. A selection logic identifies direct branches which can benefit from FAC. FAC selects among four possible addresses depending on the branch type. The Nios II ISA supports two schemes for direct branch target addresses, one uses a 16-bit offset (IMM16) and the other a 26-bit range (IMM26). Combined with the fall-through address (i.e., PC + 4) and the predicted address coming from the BTB, BTB+FAC uses a four-way multiplexer to select among these four possible addresses. Unfortunately, this multiplexer falls into the critical path.

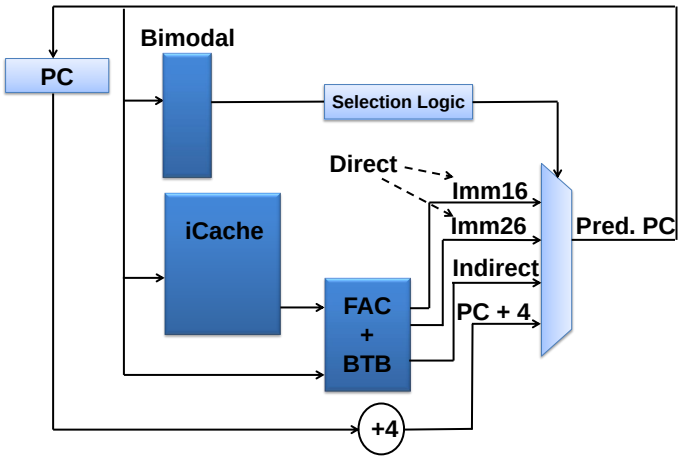

Figure 3.1: BTB with Full Address Calculation.

A lower cost and faster alternative to FAC is *Partial Address Calculation* (PAC) which relies on typical program behavior to reduce the number of choices for the final address multiplexer. Fig. 3.2 reports the relative frequency of the various branch types (see Section 5.1 for the methodology). Since IMM26 branches are far less frequent than IMM16 branches, PAC precalculates IMM16 branches and uses the BTB for IMM26 and indirect branches.

3.1.2 Return Address Stack

The RAS is a hardware, stack-like structure that accurately predicts the target address of function returns. When a call instruction executes, it also pushes its return address onto the RAS. Upon fetching a return instruction, the branch predictor can pop the top value from the RAS accurately predicting the return address. As long as the RAS has enough entries, it will accurately predict all return instructions. Since the call depth of typical workloads is not deep, virtually all high performance processors incorporate a shallow RAS. For the workloads studied a 16-entry RAS proves sufficient. The RAS for a simple pipeline is simple to implement on an FPGA. Deeper pipelines may require support for speculative RAS insertions and deletions complicating its design.

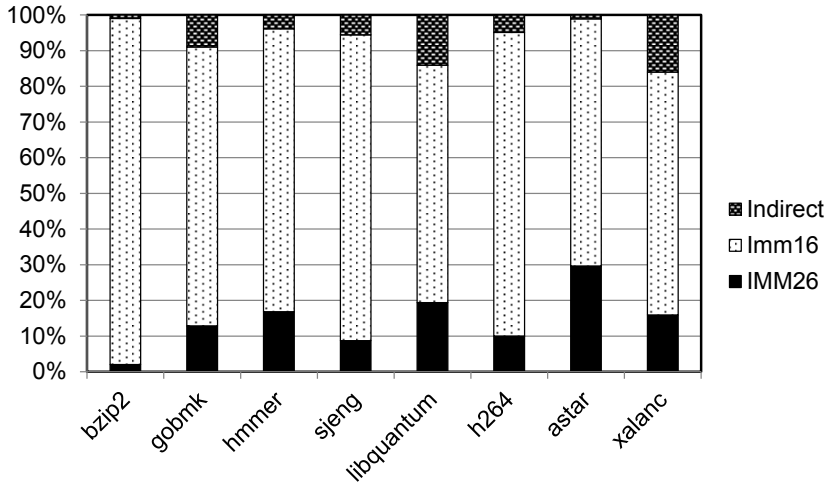

Figure 3.2: Branch Target Type Distribution.

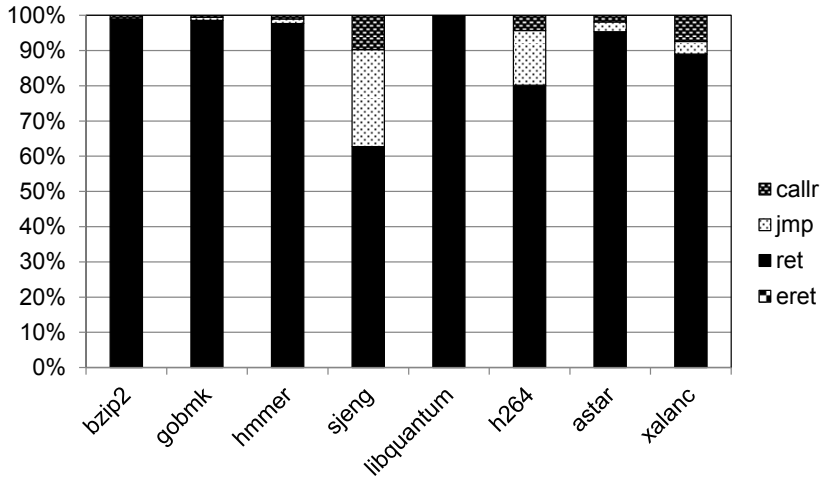

Figure 3.3: Indirect Branch Instruction Type Distribution.

3.1.3 Eliminating the BTB

Fig. 3.3 shows that for the workloads studied in this work, 97% of the indirect branches are returns (other workloads, e.g., those using virtual functions, may behave differently). Once a RAS is included along with FAC, the BTB ends up being used for only less than 1% of all branches. Accordingly, the BTB can be eliminated, and instead use a static, not-taken predictor for all indirect branches other than returns. Section 5.2 shows that removing the BTB in the presence of RAS reduces accuracy negligibly. Section 3.3.1 explains that lower-level FPGA related considerations also favor eliminating the BTB.

3.2 Direction Prediction

A bimodal branch direction predictor is a table of two-bit saturating counters that is indexed with a portion of the PC [8]. The counters are updated up or down depending on whether the branch is

taken or not respectively. The lower bit provides hysteresis to changes while the upper bit provides the prediction. As Section 5.3 corroborates, increasing the number of bimodal entries does not proportionally improve accuracy. Eventually, using a larger bimodal ceases to provide any improvement since bimodel is fundamentally limited on the branch prediction patterns it can predict.

Gshare is a pattern-based predictor wihch uses a combination of the PC and a global direction history register (GHR) to index the counter table [9]. GHR stores the direction of the last few branches in a bit vector. Each GHR bit stores the direction (taken or not) of a previous branch. Section 5.3 shows that gshare is far more accurate than bimodal. However, as Section 3.3 explains, latency suffers with gshare due to its more complex indexing scheme. Gselect, an alternative to gshare, indexes the counter table using a simple concatenation of the GHR and the PC [9]. For practical table sizes, gshare proves more accurate than gselect. Section 3.3.2 explains that with proper modification, gselect proves faster than gshare on an FPGA while sacrificing little in accuracy.

3.3 FPGA implementation optimizations

This section discusses additional FPGA-specific implementation optimizations. While this section assumes a modern, Altera FPGA, the optimizations presented should be broadly applicable.

3.3.1 Eliminating the BTB

As Section 2.7 explained, this work aims to use one M9K BRAM. An M9K BRAM can be configured as wide as 36 bits with 256 rows [4], and it can be used to implement a fused BTB and direction predictor. Specifically, each BRAM row can store one BTB entry along with up to three direction prediction entries for a total of 768 direction entries and 256 target entries. This fused BTB+DIR predictor works well with a bimodal DIR. The PC indexes a row which contains a single target prediction entry and up to three direction prediction entries. Another portion of the PC selects one of these direction prediction entries. The target is used only from taken branches.

Unfortunately, it is not possible to use one BRAM for both a BTB and the most accurate of the direction predictors considered, gshare. There are two reasons why: (1) gshare uses a different indexing scheme than the BTB, and (2) there is a limited number of ports per BRAM [4]. As Section 3.3.1 discussed, the BTB can be eliminated when address precalculation and a RAS are used. Eliminating the BTB frees up the entire BRAM for direction prediction.

3.3.2 FPGA-Friendly Direction Predictor Indexing

This section investigates which of the three branch direction predictors is best to use on an FPGA. Focusing just on accuracy gshare would be the best. However, performance is not the highest with gshare since its indexing scheme results in low clock frequency. Fig. 3.1 depicts why the predictor's indexing scheme, when implemented on an FPGA, falls into the critical path. At every clock cycle the predicted PC is used to index the direction predictor table for the next instruction. Since BRAMs are synchronous, their index must arrive before the clock edge and thus it cannot be a registered signal of the Predicted PC [4]. Moreover, the setup time for the BRAM is longer than that of simple registers. Therefore, the entire path starting from the BRAM data output, through the prediction logic and back into the lookup address of the BRAM forms the critical path. This is especially a concern with gshare

that uses the exclusive-or of the global history register (GHR) with Predicted PC to index the BRAM. This extra XOR logic prolongs the critical path, reducing the operating frequency.

Contrary to gshare, gselect has a simpler indexing scheme. Specifically, gselect uses a simple *concatenation* of the GHR with Predicted PC as index. Not only is gselect's indexing fast, but it can also be tailored to map well onto FPGAs. This work proposes *gRselect* which breaks the BRAM-to-BRAM critical path by breaking the BRAM access into two parts the first of which does not need to be "predicted PC". It uses GHR, which is a registered signal, to index the BRAM to retrieve one wide row of counters. Fig. 3.4 shows the gRselect scheme in more detail.

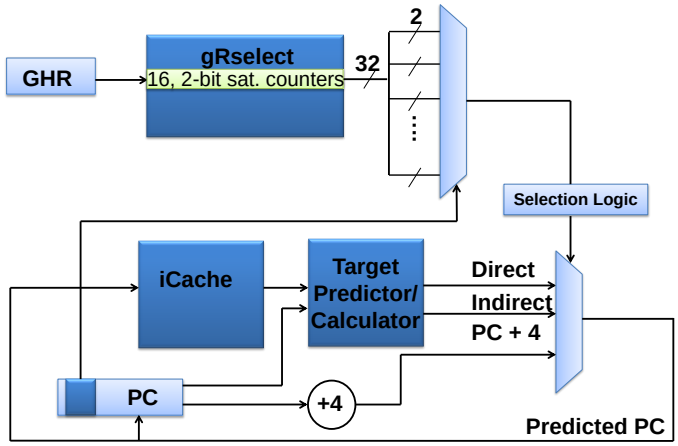

Figure 3.4: FAC with gRselect.

3.3.3 Instruction Decoding

To be able to select the appropriate target, the predictor needs to determine whether an instruction is a branch and if so, what kind of branch it is. This information is needed to select the corresponding predicted target through the output multiplexer. However, the decode logic lies in the critical path. To eliminate this delay, the predictor pre-decodes the instructions prior to installing them in the instruction cache. The pre-decode information is stored along with the instruction. This is similar to typical ASIC implementations.

Chapter 4

The Advanced FPGA-Friendly Branch Predictor

Chapter 3 proposes gRselect as the best performing minimalistic design that has a hardware budget limited to 1 M9K BRAM. In this chapter, we loose the constraint on hardware budget and consider more advanced and accurate branch prediction technologies to achieve better performance.

4.1 Perceptron Predictor

Section 2.5.2 introduced that perceptron predictor maintains vectors of weights in a table. It produces a prediction through the following steps: (1) a vector of weight (i.e., a perceptron) is loaded from the table. (2) multiply the weights with their corresponding global history (1 for taken and -1 for not-taken). (3) sum up all the products, predict taken if the sum is positive, and not-taken otherwise. Each of these steps poses difficulties to map to the FPGA platform. The rest of this section addresses these problems.

4.1.1 Perceptron Table Organization

Each weight in a perceptron is typically 8-bit wide, and perceptron predictors usually use at least 12-bit global history [7]. The depth of the table, on the other hand, tends to be relatively shallower (e.g. 64 entries for 1KB hardware budget). This requires a very wide but shallow memory, which does not map well to BRAMs on FPGAs.

4.1.2 Multiplication

4.1.3 Adder Tree

4.2 TAgged GEometric history length Branch Predictor (TAGE)

Chapter 5

Evaluation

This section presents the experimental evaluation of the proposed branch predictors. Section 5.1 details the experimental methodology. Section 5.2 evaluates the accuracy of target address schemes showing that using a RAS with FAC is best. Section 5.3 compares the accuracy of various direction predictions, showing that a single BRAM gRselect is among the best performers. Section 5.4 reports resource usage and maximum operating frequency. Finally, Section 5.4.1 reports the overall performance showing that the predictor that combines FAC, RAS, gRselect and pre-decoding is best.

5.1 Methodology

To compare the predictors this work measures: (1) Accuracy as misses per kilo instructions (MPKI) which has been shown to correlate better with performance compared to prediction accuracy alone. Processor performance in (2) instructions per cycle (IPC), a frequency agnostic metric, that isolates the effects of implementation, and in (3) instructions per second (IPS), a true measure of performance. (4) Operating frequency, and (5) resource usage.

Simulation measures MPKI and IPC using a custom, cycle-accurate, full-system Nios II simulator. The simulator boots ucLinux [1], and runs a subset of SPEC CPU2006 integer benchmarks with reference inputs [10]. The evaluation uses a baseline predictor (Base) with a fused BTB and bimodal, as discussed in Section **??** both with 256 entries. Base does not decode instructions and thus uses the BTB and the bimodal for all instructions.

All designs were implemented in Verilog and synthesized using Quartus II 12.1 on a Stratix IV chip in order to measure their maximum clock frequency and area cost. The maximum frequency is reported as the average maximum clock frequency of five placement and routing passes with different random seeds. Area usage is reported in terms of ALUTs and BRAMs used.

5.2 Target Prediction

This section measures the accuracy of target predictors. by using the baseline direction predictor while considering combinations of BTB, PAC, FAC, RAS, and larger BTBs. Fig. 5.1 reports the reduction in target address mispredictions when using various target prediction mechanisms compared to Base. Using a decode logic to filter non-branches from the BTB (BTB-256) reduces mispredictions by 30%.

However, increasing the BTB size to 512 (BTB-512) or 1024 (BTB-1024) entries does not improve accuracy noticeably. In the rest of this section, all BTB configurations except for BASE use instruction filtering.

Using PAC or FAC with a 256-entry BTB reduces mispredictions by 81% and 90% respectively, whereas using just FAC reduces mispredictions by 84%. Finally, using FAC with a RAS (FAC+RAS) proves best. In conclusion, eliminating the BTB and relying instead on a RAS+FAC is best in terms of accuracy. An added benefit of RAS+FAC is that it allows for a standalone, thus larger and more flexible direction predictor.

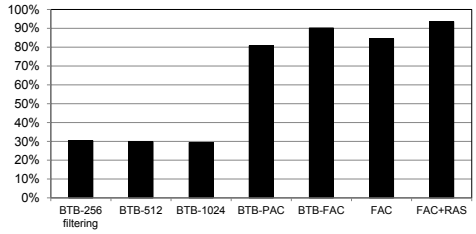

Figure 5.1: Target Address Prediction Schemes: Reduction in target address misprediction over BASE.

5.3 Direction Prediction

5.3.1 The Minimalistic FPGA-Friendly Branch predictor

Fig. 5.2 reports the improvement in MPKI for various direction predictors relative to BASE. Decoding the instructions and performing prediction only for branches improves MPKI by 17% (bimodal-256). Using a larger bimodal with 4K entries further improves MPKI by only 8% suggesting that bimodal is fundamentally limited in the branch sequences it can predict. However, using a 256- or a 4K-entry gshare improves MPKI by 79% and 82% respectively.

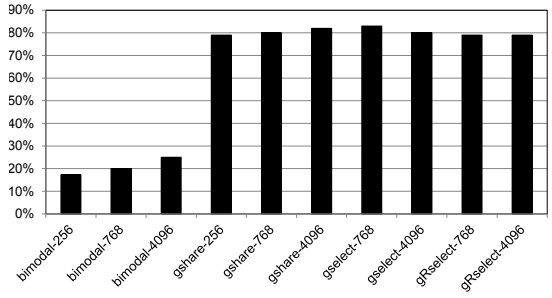

Figure 5.2: Direction Predictors: MPKI improvement over BASE.

Section **??** explained why gselect may be better to implement on an FPGA. Fig. 5.2 show that a conventionally indexed 4K-entry gselect results in competitive accuracy, improving BASE by 80%. Section 5.4 explained that the desired number of entries for the direction predictor is either 768 or 4K when fused with a BTB or not respectively. The figure shows that a conventionally indexed 4K-entry gselect improves MPKI by 80%, while the proposed FPGA-friendly organization, gRselect, improves MPKI by 79%. In conclusion, gshare achieves the best accuracy with gselect and gRselect offering

competitive accuracies.

5.3.2 The Overriding Branch predictor

adfasdfasdf

5.4 Area and Frequency

Fig. 5.3 shows the maximum frequency and area utilisation for each predictor design. All configurations use one BRAM. As expected, BASE is the fastest and least expensive. Adding instruction filtering reduces fmax from 353 MHz to 287 MHz, a 18% drop. By adding address calculation, frequency drops even further. However, removing the BTB partially recovers from this frequency drop. Finally, adding a RAS to a gRselect with pre-decoding, results in a predictor that operates at 259 MHz and that uses only 147 ALUTs.

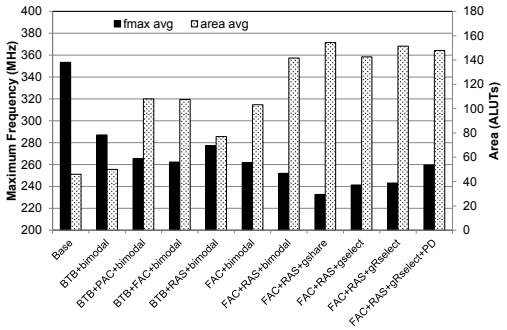

Figure 5.3: Maximum frequency and area utilisation. (PD = pre-decoding)

5.4.1 Performance

Fig. 5.4 reports average IPC gain compared to BASE. The bimodal predictor results in the lowest IPC while gselect performs almost as well as gshare.

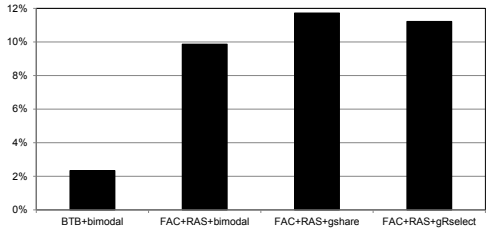

Figure 5.4: Improvement in IPC over BASE.

IPC is proportional to performance only when the clock frequency remains the same. Actual performance depends on IPS, the product of IPC and clock frequency. Fig. 5.5 reports overall performance in IPS. This experiment assumes a 250MHz maximum clock speed for the processor, the maximum clock frequency of Nios-II-f on the Stratix IV [5]. The best performing predictor is a 4K-entry gRselect with FAC+RAS, no BTB, and that uses pre-decoded instructions.

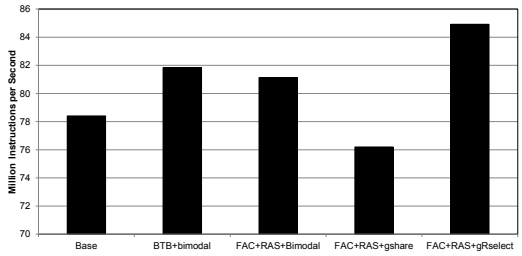

Figure 5.5: IPS comparison of processors with various predictors.

Chapter 6

Conclusion

This is more like a conclusion

Bibliography

[1] Arcturus Networks Inc., uClinux. http://www.uclinux.org/.

[2] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length predictors. In *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2006.

[3] Altera Corp. *Stratix IV Device Handbook*, Feb. 2011.

[4] Altera Corp. *TriMatrix Embedded Memory Blocks in Stratix IV Devices*, Dec. 2011.

[5] Altera Corp. *Nios II Performance Benchmarks*, Dec. 2012.

[6] Altera Corporation. *Nios II Processor Reference*, May 2011.

[7] D. A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Intl' Symposium on High-Performance Computer Architecture*, January 2001.

[8] J. E. Smith. A study of branch prediction strategies. In *Annual Symposium on Computer Architecture*, June 1981.

[9] S. McFarling. Combining Branch Predictors. TN-36, Digital Western Research Laboratory, June 1993.

[10] Standard Performance Evaluation Corporation. SPEC CPU 2006. http://www.spec.org/cpu2006/.

[11] Xilinx Inc. *MicroBlaze Processor Reference Guide*, July 2012.

[12] Peter Yiannacouras, Jonathan Rose, and J Gregory Steffan. The microarchitecture of fpga-based soft processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 202–212. ACM, 2005.