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. However, 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 predictors and does so in the context of simple pipelined processors, the most commonly used general purpose soft processor architecture. It assumes a processor implementation representative of Altera's Nios II-f [4] 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 and resource cost. Finally, this work proposes a minimalistic branch predictor *gRselect* with the hardware budget on par with Nios II-f, as well as an overriding TAGE [13][2] predictor *O-TAGE-SC* that delivers the highest performance among all the considered branch prediction schemes.

Acknowledgements

I have been fortunate enough to receive enormous help and support while completing this thesis. Firstly, I would like to express my deepest gratitude to my supervisor, Prof. Andreas Moshovos, for his unending guidance and support. Secondly, it has been my great pleasure to work with all the members of the AENAO research group for their feedbacks throughout this work. Special thanks to Kaveh Aasaraai and Ian Katsuno for their unreserved help on FPGA technologies.

Finally, I would like to thank my parents for all the unconditioned love and support during my undergraduate and graduate studies. Moreover, a heartful thank you to my beloved girlfriend Mengyun Shen for always being there cheering me up, I cannot imagine finishing this work without your constant understanding and faith in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are increasingly being used in embedded and other systems. Such designs often employ one or more embedded microprocessors, and there is a trend to migrate these microprocessors to the FPGA platform primarily for reducing costs. While these soft processors cannot typically match the performance of hard processors, soft processors are flexible allowing designers to implement the exact number of processors desired and to customize them to efficiently fit the application's requirements.

Current commercial soft processors such as Altera's Nios II [4] and Xilinx's Microblaze [22] use in-order pipelines with five to six pipeline stages. These processors are often used for less computation-intensive applications such as system control tasks that are often control flow intensive. To support more compute-intensive applications, a key performance improving technique is branch prediction. Without branch prediction, a branch has to execute 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, mostly in the context of application specific custom logic (ASIC) implementations. However, naïvely porting ASIC-based branch predictors to FPGAs results in slow and/or resource-inefficient implementations as the tradeoffs are different for reconfigurable compared to custom logic. Accordingly, this work studies the FPGA implementation of several commonly used and advanced 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 that is modeled loosely on Altera's highest performing soft-processor 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.

We use a two-pronged approach proposing two classes of predictors that are appropriate depending on the amount of resources we are willing to devote to the predictor component. Branch predictors make use of memories to store past branch prediction outcomes and their accuracy tends to improve the more memory resources they are given. Depending on the device, an FPGA may offer many or very few memory resources. Altera's Nios II-f uses three Block RAMs (BRAMs) in total and only one BRAM is used for branch prediction [8]. Therefore, each additional BRAM would represent a more than 1/3 overhead in terms of BRAM resources. Accordingly, we first consider minimalistic branch predictors that also use just one BRAM. In the second approach, we relax this restriction and consider more elaborate and accurate branch predictors and use as many BRAMs as necessary to improve accuracy.

1.1 Design Goals

This work aims to design branch predictors that (1) operate at a high operating frequency while (2) achieving high accuracy so that they improve execution performance. It proposes a minimalistic branch predictor that has a limited resource budget of one BRAM. The design of the minimalistic predictor considers the most commonly used direction predictors, bimodal [11], gshare and gselect [16] (reviewed in Chapter 2). These predictors use a single lookup table and map relatively well onto a single BRAM.

This work also implements more advanced Perceptron and TAGE predictors. As Chapter 4 will show, a single-cycle TAGE is prohibitively slow. Therefore, this work considers an overriding TAGE predictor [13] that produces a base prediction in one cycle while overriding that decision with a better prediction in the second cycle if necessary. Perceptron and TAGE both require large tables. Accordingly, this work investigates how their accuracy and latency vary with the amount of hardware resources they are allowed to use.

1.2 Contributions

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 using *Full Address Calculation* (FAC). It shows that a branch target predictor consists of a FAC and a Return Address Stack (RAS) [14] is 63.2% more accurate than a naïve BTB solution.

(2) It studies the FPGA implementation of the three most commonly used branch direction predictors: bimodal [11], gshare, and gselect [16]. 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% in terms of overall performance.

(3) It studies the FPGA implementation of two advanced branch predictors: the Perceptron [10] and TAGE [2] predictors. It optimizes Perceptron's maximum operating frequency by introducing (i) a complement weight table to simplify the multiplication that is otherwise necessary at prediction time, and (ii) Low Order Bit (LOB) Elimination for faster summation. It finds that TAGE is too slow for single-cycle access which negates its advantage in accuracy. Accordingly, this work proposes an overriding predictor *O-TAGE-SC* that uses a simple base predictor to provide an initial prediction in the first cycle which can be overridden in the second cycle should TAGE disagree with relatively high confidence. O-TAGE-SC achieves 5.2% better instruction throughput over gRselect.

The rest of the thesis is organized as follows. Chapter 2 reviews branch prediction basics and the branch prediction schemes considered in this work. Chapter 3 discusses the design of a minimalistic branch predictor that uses a hardware budget on par with that of Nios II-f. Chapter 4 presents the study on two more advanced branch predictors: the Perceptron and TAGE predictors. Chapter 5 presents the experimental evaluation results and finally Chapter 6 concludes.

Chapter 2

Background

This chapter reviews branch prediction basics and relevant past work. Section 2.1 introduces the importance of branch prediction. Section 2.2 describes the structure of a canonical branch predictor. Sections 2.3 and 2.4 overviews the branch prediction schemes considered in this thesis. Finally, Section 2.5 briefly introduces Field Programmable Gate Arrays (FPGAs) and soft processors.

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 assuming a fixed instruction length, which is true for Altera's Nios II instruction set. 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*, that is to determined its target address, to fetch the correct instruction, hence severely impacts performance.

Branch prediction is a technique to improve performance by guessing the branch target to allow the processor to fetch the corresponding instructions and execute them speculatively. The processor fetches the instruction at the predicted target, and compares the predicted 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 produced by these wrong instructions has to be squashed with an extra roll-back penalty. Therefore,

prediction accuracy is the key to high performance.

2.2 Branch Prediction Overview

Branch prediction entails two steps: *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), i.e., the memory address, 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.

2.3 Branch Target Prediction

Branch Target Prediction usually requires a Branch Target Buffer (BTB), a cache-like structure that records the addresses of the branches and their target addresses. 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 be set-associative to reduce aliasing.

Another common structure used for branch target prediction is the Return Address Stack (RAS), a stack-like structure that predicts the target address of function returns. When a call instruction executes, the return address of that call is pushed onto the RAS. When the processor executes the corresponding

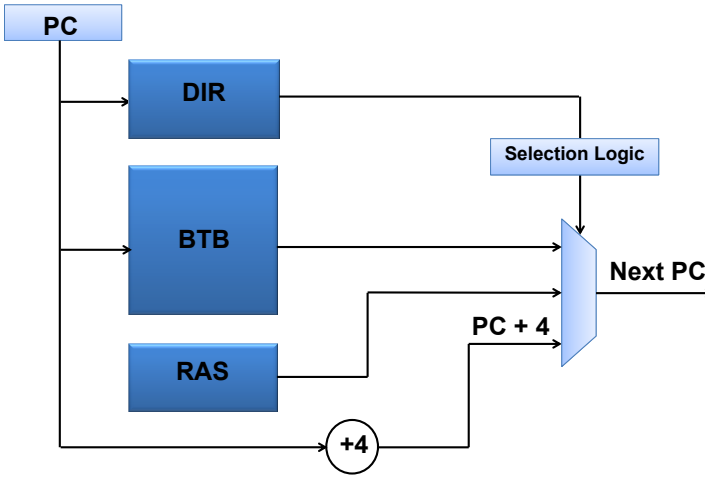

Figure 2.1: Canonical Branch Predictor.

return instruction, RAS pops the return address and provides a prediction. The prediction is accurate as long as the size of the RAS is greater than the current call depth. Most modern processors have a shallow RAS because typical programs generally do not have very deep call depths.

2.4 Branch Direction Prediction

This section describes the branch direction predictors. This section first briefly reviews *static* prediction schemes. Then, it reviews *dynamic* prediction schemes, starting with the simple schemes such as bimodal [11], gshare and gselect [16], to more complex schemes such as Perceptron [10] and TAGE [2].

2.4.1 Static Prediction Scheme

Static branch direction predictors statically predict a branch to be taken or non-taken. Initially, this strategy was motivated on the observation that most branches are taken. However, prior work has shown that approximately 30% of all branches are *unconditional* branches (e.g., functions calls and returns) [15], which are of course taken. The remaining 70% branches are *conditional* branches, and their directions may not be strongly biased towards one direction than the other.

The main weakness of static branch predictors is that they cannot adapt to various applications. A static prediction scheme variation called "backward-taken forward-not-taken" takes advantage of the observation that most backward branches are taken (e.g., the end of loops), while forward branches are often not taken. However, the improvement in accuracy is not as significant. As the pipeline in modern processors became deeper, the misprediction penalties increased, which necessitated more

accurate *dynamic* branch predictors.

2.4.2 Bimodal, Gshare and Gselect Predictor

Bimodal [11], Gshare and Gselect [16] 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) [24]. 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 the bimodal predictor is indexed by selected bits of the PC. Accordingly, the bimodal predictor works by identifying the temporal bias of each branch instruction. Since not all bits of the 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.

While many branches exhibit temporal biases which bimodal predictor can capture successfully, often there are branches that exhibit specific direction patterns or branches that are correlated with one another. Gshare and gselect take advantage of these phenomena and try to avoid aliasing. Specifically, to improve accuracy they 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 have better accuracy over bimodal because of the reduced aliasing by correlating local history (i.e., PC) with global history (i.e., GHR). They essentially record a series of branch directions with the direction of a subsequent branch. This allows them to discover patterns in branch behavior that may not be amenable to temporal bias based prediction.

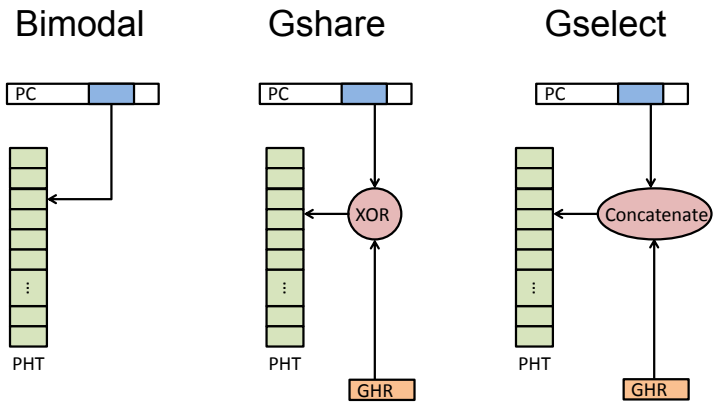

Figure 2.2: Bimodal, Gshare and Gselect.

2.4.3 Perceptron Predictor

The Perceptron predictor uses vectors of signed weights (i.e., perceptrons) to represent correlations among branch instructions [10]. It is capable of determining which preceding branches are correlated with the current branch contrary to Gshare and Gselect that correlate with all preceding branches in the history register. Fig. 2.3 shows the structure of a Perceptron predictor. It produces a prediction through the following steps: (1) A *perceptron* is read from the table. (2) The weights are multiplied with factors chosen based on the corresponding global history bits. The weights are multiplied by 1 for taken and -1 for not-taken. (3) The resulting products are summed up and a prediction is made based on the sign of the result; predict taken if the sum is positive, and not-taken otherwise. Formally, for a Perceptron predictor using h history bits, let G_i, where $i = 1...h$, be 1 for taken and -1 for not-taken, each weight vector has h weights $w_{0...h}$, where the bias constant $w_0 = 1$. The predictor has to calculate $y = w_0 + \sum_{i=1}^{h} G_i w_i$, and predict taken if y is positive and not-taken otherwise.

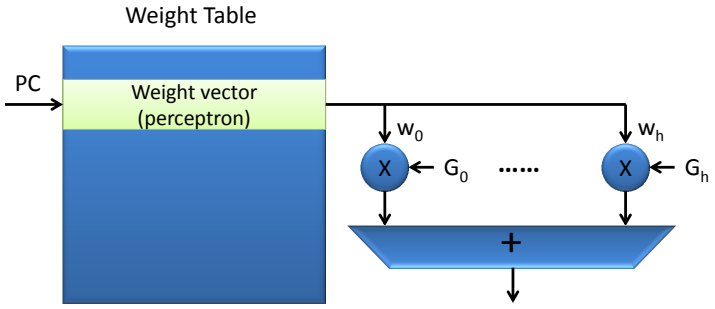

Figure 2.3: The Perceptron branch predictor.

When a branch is resolved, the corresponding perceptron is updated. Let y be the computed results at prediction time, t be the branch outcome, x_i be the i_{th} global history, and let θ be the *threshold*, the following algorithm is used to train the Perceptron:

```
if sign(y) ≠ t or |y| ≤ θ
      for i = 0 to n do
            w_i = w_i + t x_i
      end for
end if
```

In the preceding algorithm, t and x_i are bipolar, i.e., each of them is 1 if the branch outcome is taken and -1 otherwise. The algorithm increments the i_{th} weight when the outcome agrees with x_i, and decrements the weight otherwise. When there is a strong correlation, the absolute value of the weight will be large, which will have a large influence on the outcome. On the other hand, when there is a weak

correlation, the absolute value of the weight will be close to zero, hence contributes little to the result.

2.4.4 Tagged Geometric History Length Branch Predictor (TAGE)

The TAGE predictor features a bimodal predictor as a base predictor T_0 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 of various lengths. 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 judiciously. Essentially, the base predictor captures the bulk of branches that tend to be biased, while the remaining components capture exceptions by recording specific history events that lead to exceptions that foil the base predictor. Fig. 2.4 shows a 5-component TAGE predictor. Each table entry has a 3-bit saturating counter *ctr* for the prediction result, a *tag*, and a 2-bit useful counter u. The table indices are produced by hashing the PC and the global history using different lengths per table $L(i)$. All tables are accessed in parallel and each table provides a valid prediction only on a tag match and provided that the corresponding useful counter is saturated. The final prediction comes from the matching tagged predictor component that uses the longest history.

Seznec et al. defines the *provider component* as the predictor component that provides the prediction, and the alternate prediction *altpred* as the prediction that would have occurred without the provider component [2].

When updating TAGE, the useful counter u of the provider component is updated when the alternate prediction disagrees with the provider. u is incremented if the provider is correct, and decremented otherwise.

The prediction counter *ctr* of the provider is incremented if the prediction is correct and decremented otherwise. If the prediction from the provider T_i, where $i < M$, is incorrect, a new entry will be allocated on a predictor component T_k that uses a longer history, i.e., $i < k \leq M$.

2.5 Field Programmable Gate Arrays and Soft Processors

Field Programmable Gate Arrays (FPGAs) are chips that can be reconfigured by designers. Modern FPGAs such as Altera's Stratix IV [5] 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.

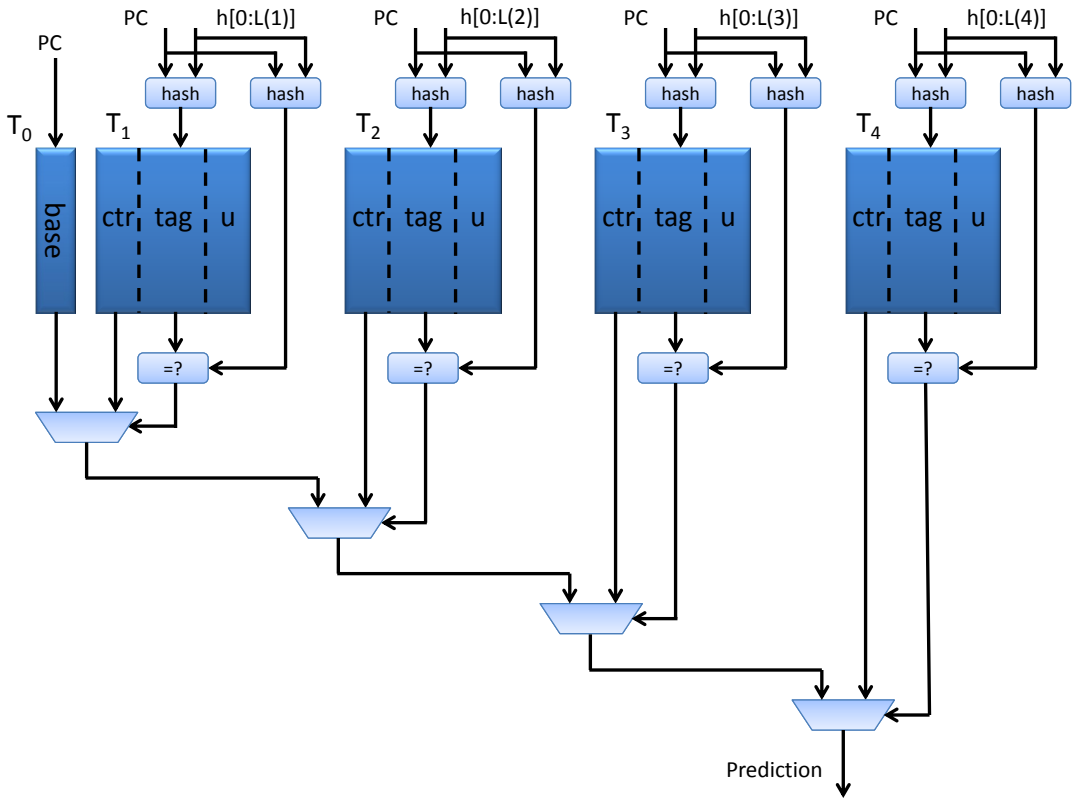

Figure 2.4: A 5-component TAGE branch predictor.

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 [25]. These systems usually contain one or more embedded processors, which necessitates high-performance FPGAs-based *soft processors*.

Since an FPGA platform is significantly different than an ASIC, hard and soft processors have distinct design tradeoffs. Commercial soft processors such as Alteras's Nios II-f [8] and Xilinx's MicroBlaze [22] usually have shallow pipelines, mostly because the relative speed gap between memory and logic is much smaller than on an ASIC. Accordingly, branch predictor designs for soft processors also have to be re-evaluated.

The following chapters discuss various FPGA-friendly branch predictor designs.

Chapter 3

The Minimalistic FPGA-Friendly Branch Predictors

This chapter discusses the architecture of the various minimalistic FPGA-friendly branch predictor designs. It proposes *gRselect* as the best performing predictor. 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 because the BTB is a much smaller storage than the memory.

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 5.3.2 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 to 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 that 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.

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 [14]. 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

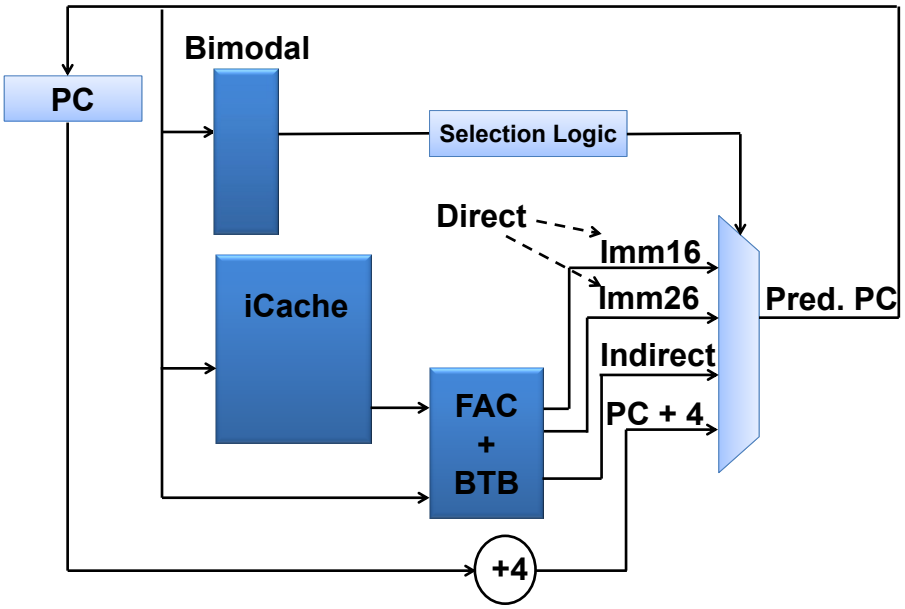

Figure 3.1: BTB with Full Address Calculation.

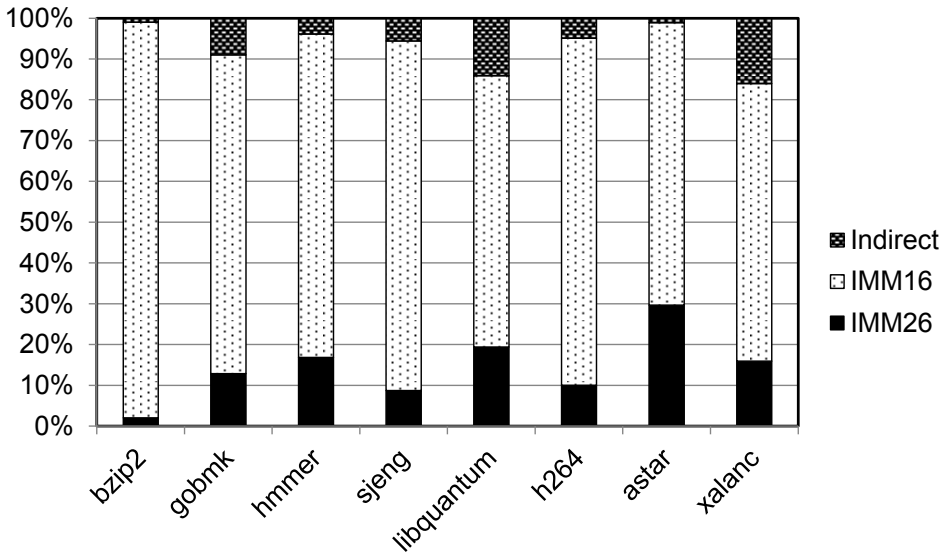

Figure 3.2: Branch Target Type Distribution.

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.

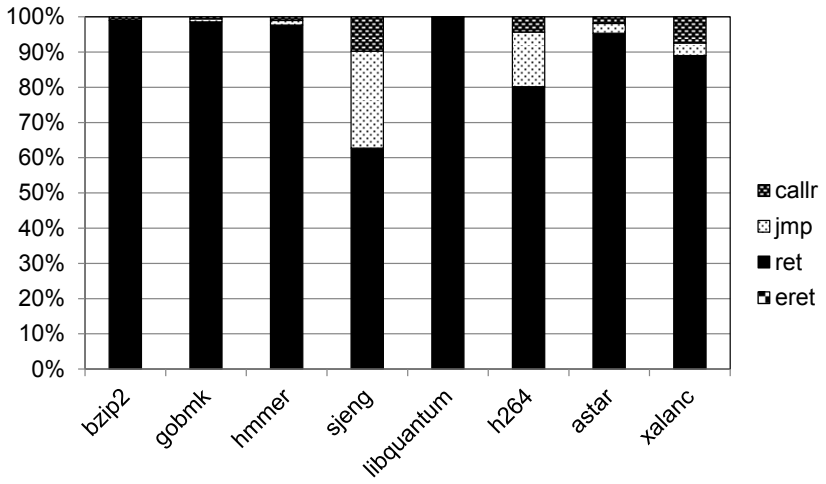

Figure 3.3: Indirect Branch Instruction Type Distribution. *callr* - call register, *jmp* - jump, *ret* - return, *eret* - exception return

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

As discussed in Section 2.4.2, a bimodal branch direction predictor is a table of two-bit saturating counters that is indexed with a portion of the PC [11]. 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 5.3.1 corroborates, increasing the number of bimodal entries does not proportionally improve accuracy. Eventually, using a larger bimodal predictor ceases to provide any improvement since bimodal is fundamentally limited on the branch prediction patterns it can predict.

Gshare is a pattern-based predictor that uses the PC *XORed* with a global direction history register (GHR) to index the counter table [16]. 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. Combining local (i.e., PC) and global information helps to better identify the current branch. Section 5.3.1 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 PC and the GHR [16]. Fig. 3.4 shows an example of a gselect that uses eight bits of the PC and eight bits of the GHR (i.e., a gselect 8/8). Effectively, the bits from the PC divide the counter table into regions, and the bits from GHR select the corresponding entry within each region. However, McFarling et al. have shown that global history information weakly identifies the current branch [16]. This observation suggests that a lot of the entries within each region will be underutilized, reducing the effective size of the counter table of gselect. Therefore, 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.

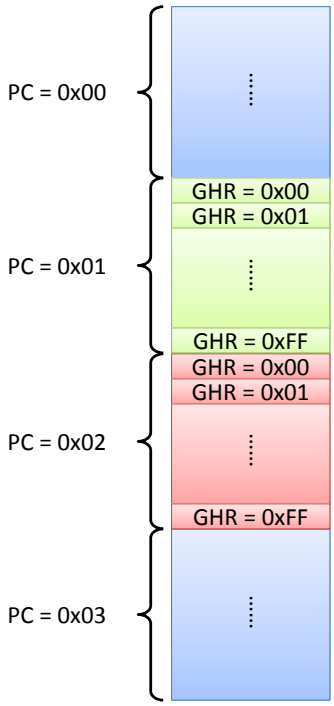

Figure 3.4: An example of gselect 8/8.

3.3 FPGA implementation optimizations

This section discusses additional FPGA-specific implementation optimizations. Specifically, Section 3.3.1 discusses fusing the BTB and the direction predictor over a single BRAM and explains that it is best to avoid the BTB altogether. Section 3.3.2 explains how gselect can be adapted to map well on an FPGA.

Finally, Section 3.3.3 discusses the use of pre-decoding. While this section assumes a modern, Altera FPGA, the optimizations presented should be broadly applicable.

3.3.1 Eliminating the BTB

As Section 1.1 explained, the minimalistic branch predictor aims to use one M9K BRAM. An M9K BRAM can be configured as wide as 36 bits with 256 rows [6], 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, as shown in Fig. 3.5.

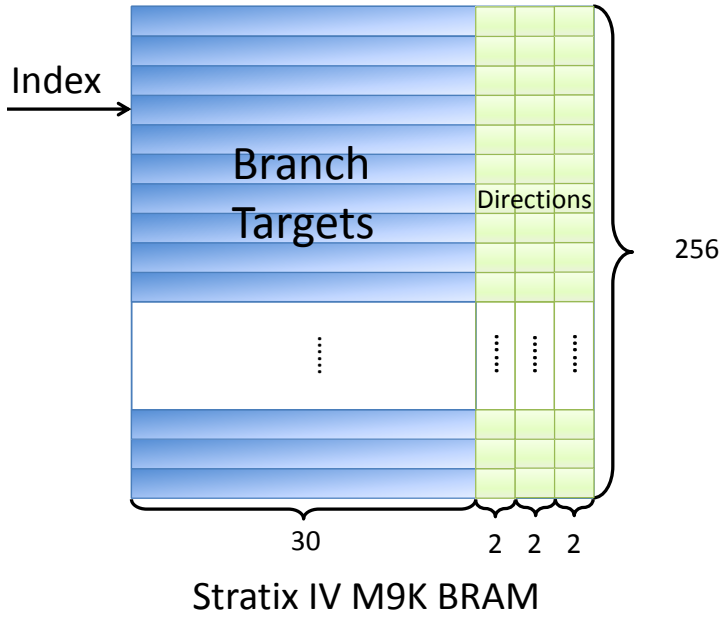

Figure 3.5: Fused BTB and Direction Predictor.

Fig. 3.6 shows the BRAM configuration of a BTB fused with direction predictors. The PC indexes a row which contains a single target prediction entry and up to three direction prediction entries. Fig. 3.6 (a) shows a BTB fused with a 256-entry bimodal predictor, where only one direction prediction entry is used. In this case, the last two columns of direction prediction entries are wasted. To utilize the entire BRAM, all three direction entries can be read from the BRAM, and another portion of the PC selects one of these direction prediction entries, as shown in Fig. 3.6 (b). This is essentially a BTB fused with a 768-entry bimodal predictor, although the larger capacity improves its accuracy, the extra multiplexer that selects between the three entries slows down the branch predictor. Fig. 3.6 (c) shows a configuration very similar to the previous case, except that it uses two bits from the GHR to select

between the three direction prediction entries. This configuration is essentially a BTB fused with a 768-entry gselect, which is much more accurate than a 768-entry bimodal predictor because it incorporate global history, but the extra latency due to the multiplexer persists.

Unfortunately, it is not possible to use one BRAM for both a BTB and the most accurate of the direction predictors considered, gshare. The reason is that the indexing of BTB is different than gshare: BTB uses the PC, while gshare uses the PC XORed with the GHR. It requires two read ports on a BRAM to access the BTB and the gshare in the same cycle. However, there are only two ports per BRAM [6]. One of the two ports has to be used for updating the predictor, which leaves only one port for accessing the predictor. Hence, implementing a BTB fused with gshare is impossible, as shown in Fig. 3.6 (d).

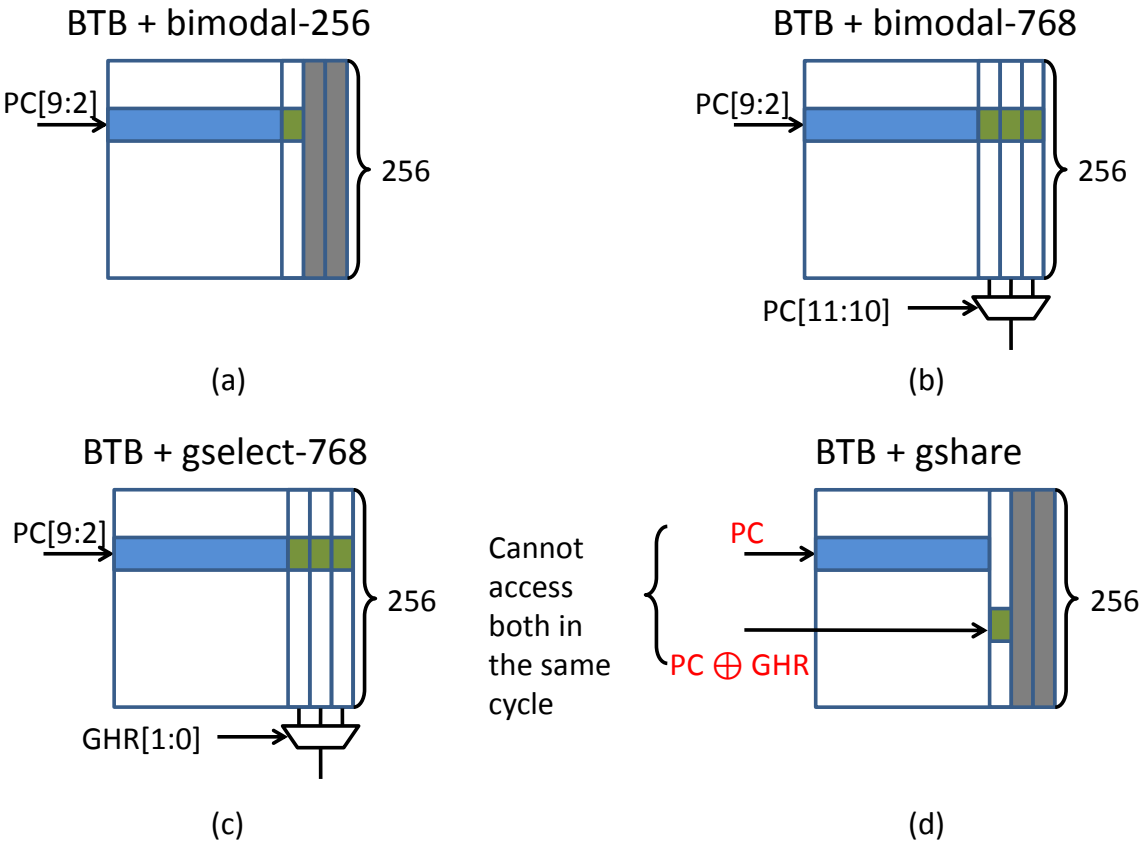

Figure 3.6: Fused BTB and Direction Predictor Usage.

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. The target is used only

for taken branches.

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. As Section 5.3.1 will show, focusing just on accuracy gshare would be the best. However, performance is not the highest with gshare since its indexing scheme results in slow clock frequency. Fig. 3.7 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, implemented with a BRAM, 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 [6]. Moreover, the setup time for the direction table 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 (the f unit in Fig. 3.7) prolongs the critical path, reducing the operating frequency.

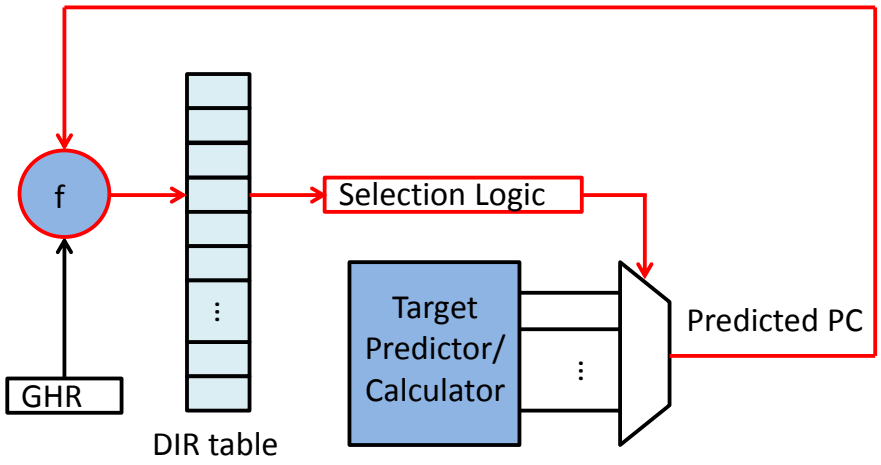

Figure 3.7: The critical path of gshare and gselect.

Contrary to gshare, gselect has a simpler indexing scheme. Specifically, gselect uses a simple *concatenation* of the GHR with the 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 the

"predicted PC". It reverses the indexing order and uses GHR, which is a registered signal, to index the BRAM to retrieve one wide row of direction prediction entries, and then uses the registered PC to select between these entries. Hence the naming *gRselect*, where *R* stands for "Reversed". Fig. 3.8 shows the gRselect scheme in more detail.

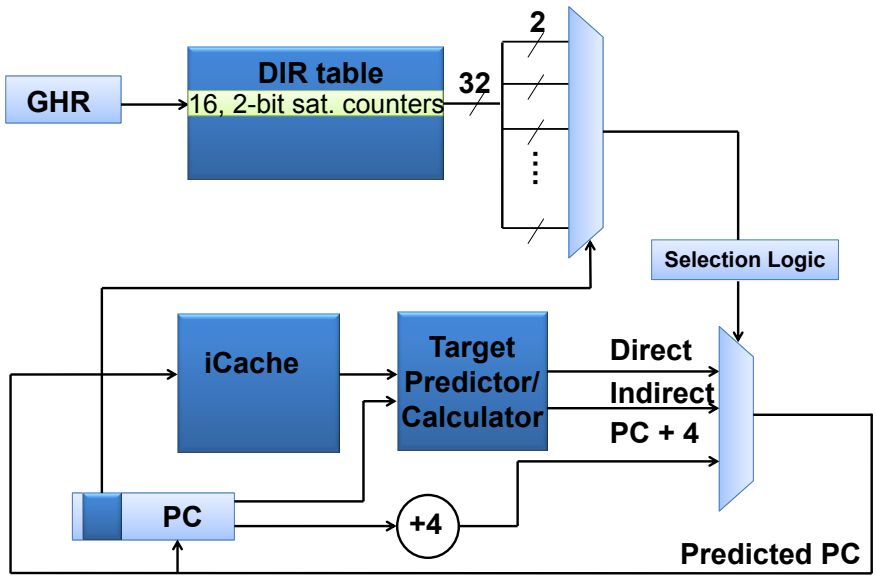

Figure 3.8: FAC with gRselect.

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.

The minimalistic branch predictor has a very strict hardware budget of one BRAM. Such a small hardware budget prevents us from considering more elaborate and accurate branch prediction schemes. The next chapter discusses branch predictor designs without this resource constraint.

Chapter 4

The Advanced FPGA-Friendly Branch Predictors

Chapter 3 proposed gRselect as the best performing minimalistic design that has a hardware budget limited to one M9K BRAM. In this chapter, we relax the constraint on hardware budget and consider more advanced and accurate branch prediction technologies to achieve better performance. Sections 4.1 and 4.2 discuss Perceptron and TAGE implementations respectively, while Section 4.3 discusses the branch target predictor.

4.1 Perceptron Implementation

Section 2.4.3 explained that the Perceptron predictor maintains vectors of weights in a table and produces a prediction through three steps. Each of these steps poses difficulties to map to an FPGA substrate. 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 a 12-bit global history [10]. The depth of the table, on the other hand, tends to be relatively shallow (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. For example, the widest configuration of an M9K BRAM on Altera Stratix IV chips is 36-bit wide times 1K entries [6]. If we implement the 1KB Perceptron as proposed by Jiménez et al. [10], which uses 96-bit wide perceptrons with 12-bit global history, it will result in a

huge resource inefficiency as shown in Fig 4.1. Stratix IV chips have another larger but slower and fewer M144K BRAM [6], which can be configured as wide as 72-bit times 2K entries, clearly the inefficiency problem persists and it would impact maximum operating frequency.

Since typically the Perceptron table does not require large storage space, the proposed Perceptron implementation uses MLABs as storage, which are fast fine-grain distributed memory resources. Since 50% of all LABs can be configured as MLAB on Altera Stratix IV devices, using MLABs does not introduce routing difficulty.

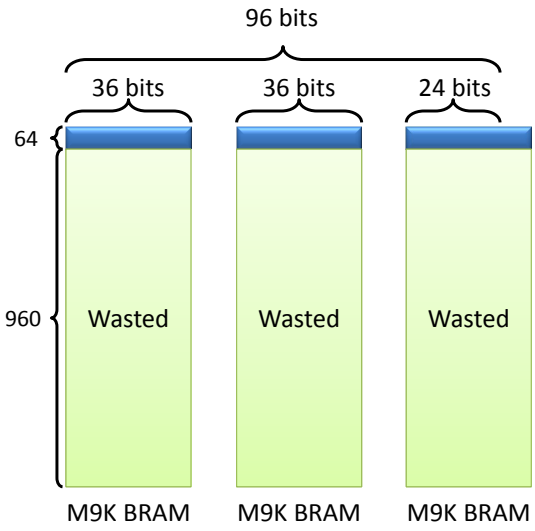

Figure 4.1: Inefficient use of M9K BRAMs to implement wide but shallow perceptron tables.

4.1.2 Multiplication

The multiplication stage calculates the products of weights in a perceptron and their global direction histories. Since the value of the global direction history can only be either 1 or -1, the "multiplication" degenerates to two cases, i.e., each product can either be the true form or the 2's complement (i.e., negative) form of each weight. A straightforward implementation calculates the negative of each weight and uses a mux to select, using the corresponding global history bit, the appropriate result, as Fig. 4.2(a) shows. To improve operating frequency, when updating the perceptron in the execution stage where the branch is resolved, both positive and negative forms of the updated weight can be pre-calculated, and the negatives can be stored on a complement perceptron table. This way, the multiplication stage at prediction time requires only a 2-to-1 mux, as Fig. 4.2(b) shows. This optimization trades off increased resources (it requires extra storage for the negative weights) for improved speed.

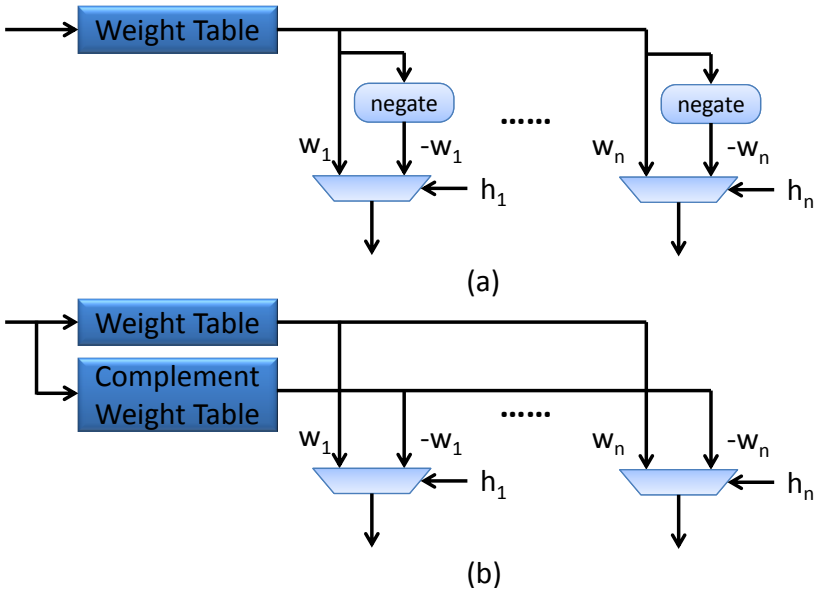

Figure 4.2: Perceptron multiplication implementation.

4.1.3 Adder Tree

The adder tree sums the products from the multiplication stage. As Section 5.4.1 will show, a global history of at least 16 bits has to be used to achieve sufficient accuracy. Implementing a 16-to-1 adder tree for 8-bit integers naïvely degrades maximum frequency severely. The maximum frequency has to be improved for Perceptron to be practical.

This work employs *Low Order Bit (LOB) Elimination* that was proposed by Aasaraai et al. [3]. LOB elimination ignores the Low Order Bits (LOBs) of each weight and only use the High Order Bits (HOBs) during prediction, while still using all the bits for updates. Section 5.4.1 shows that eliminating five LOB bits reduces accuracy by less than 1% compared to using all eight bits, but summing fewer bits results in 14.6% higher maximum frequency. Section 5.4.5 will show that using three HOBs for prediction achieves the best overall performance.

Cadenas et al. proposed a method to rearrange the weights stored in the table to reduce the number of layers of the adder tree [9]. Assuming a Perceptron predictor uses h history bits, instead of storing h weights w_i where $i = 1...h$, a new form of weights $\widetilde{w}_i$: $\widetilde{w}_i = -w_i + w_{i+1}; \widetilde{w}_{i+1} = -w_i - w_{i+1}$, for $i = 1, 3, ..., h-1$ is used. The perceptron prediction can now be computed by $y = w_0 + \sum_{i=1}^{h/2}(-G_{2i-1})\widetilde{w}_{2i-h_{2i-1}\bigoplus h_{2i}}$. This work applies this new arrangement because it pushes part of the calculation to the less time critical update logic of the Perceptron predictor so that only $h/2$ additions have to be performed, hence reduces the number of adders required by 50%. Table 4.1 gives an example of this new arrangement for $i = 1$.

h_1	h_2	$(-G_{2i-1})\widetilde{w}_{2i-h_{2i-1}\bigoplus h_{2i}}$ calculation for $i=1$
0	0	$(-G_1)\widetilde{w}_{2-0} = \widetilde{w}_2 = -w_1 - w_2$
0	1	$(-G_1)\widetilde{w}_{2-1} = \widetilde{w}_1 = -w_1 + w_2$
1	0	$(-G_1)\widetilde{w}_{2-1} = -\widetilde{w}_1 = w_1 - w_2$
1	1	$(-G_1)\widetilde{w}_{2-0} = -\widetilde{w}_2 = w_1 + w_2$

Table 4.1: Perceptron Weight Arrangement Example ($i = 1$).

However, if we look at its implementation closely, this new arrangement *selects* whether the sum or the difference of original weights w_i and w_{i+1}, where $i = 1, 3, 5, ...h-1$, should be calculated. That is, the $h/2$ adders saved during prediction are *replaced* with the same number of multiplexers. The latency difference between 3-bit 2-to-1 muxes and 3-bit adders are negligibly small, which seems to trivialize the usefulness of this arrangement. However, as Section 4.1.4 will show, this layer of muxes combined with the muxes shown in Fig. 4.2 map well on FPGAs, therefore it still proves to be beneficial to use this arrangement.

Using fast adders such as carry-lookahead adders does not help to reduce the adder tree latency as the problem is not summing few very wide numbers, but many narrow numbers. Most of the latency comes from going through layers of adders rather than propagating the carry bits. To further improve maximum frequency, this work adapts the implementation of a Wallace Tree [20]. A Wallace tree is a hardware implementation of a digital circuit that efficiently sums the partial products when multiplying two integers, which is similar to what is needed in Perceptron. The Wallace tree implementation proves to be 10.5% faster than a naïve binary reduction tree implementation.

4.1.4 Perceptron Predictor Structure on FPGA

Fig. 4.3(a) shows the Perceptron structure with the optimizations introduced earlier in this section. The positive and negative weights with the new arrangement are read from the weight table and the complement weight table. The history bits h_i and h_{i+1} are XORed to select between weights $\widetilde{w}_i$ and $\widetilde{w}_{i+1}$, where $i = 1, 3, 5, ...h-1$. They represent either the sum or the difference of the original weights w_i and w_{i+1}, where $i = 1, 3, 5, ...h-1$. These weights are fed into the first layer of muxes, where either the sum or the difference is selected. Then another layer of muxes determine the signs of the sums and/or differences. The outcomes are passed to the Wallace Tree to calculate the final result.

Modern FPGAs such as the Altera Stratix family [5] and the Xilinx Virtex family [23] feature full 6-input Lookup Tables (6-LUT) that can implement any 6-input function. The two layers of muxes

shown in Fig. 4.3(a) can be combined into one layer of 4-to-1 muxes, as shown in Fig. 4.3(b). Each 4-to-1 multiplexer is a 6-input function, hence can be implemented with a single 6-LUT. Therefore, the main benefit of replacing the layer of adders with the layer of muxes is that these muxes can be combined with the other layer of muxes. This latency reduction comes for free from the arrangement discussed in Section 4.1.3.

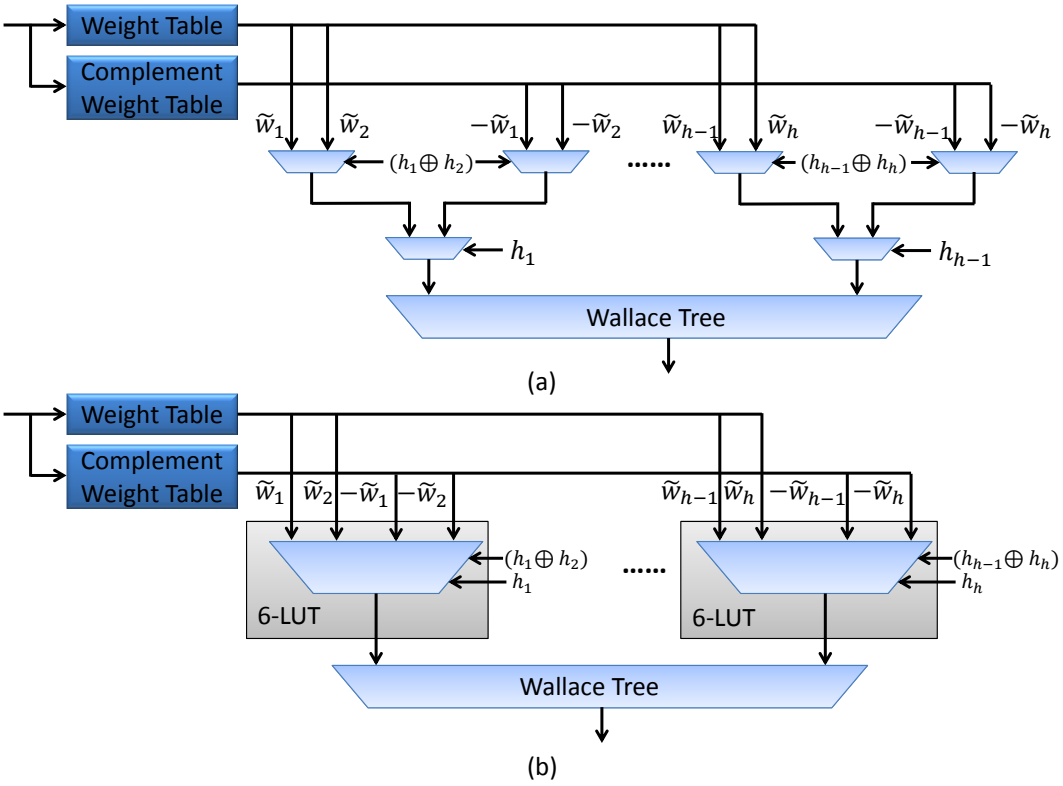

Figure 4.3: Perceptron Structure on FPGA.

4.2 TAGE Implementation

Section 5.4.3 shows that TAGE is the most accurate amongst all the direction predictors considered in this work when they use the same hardware budget. Unfortunately, TAGE uses multiple tables with tagged entries that require comparator driven logic, which does not map well onto FPGAs. Section 5.4.5 shows that the resulting frequency slowdown with TAGE is not amortized by the corresponding accuracy gains. Fortunately, TAGE can be used as an overriding predictor maintaining the accuracy gains and relatively high operating frequency.

The critical path of TAGE is as follows: (1) It performs an elaborate PC-based hashing to generate multiple table indices, one per table. (2) It accesses the tables and in parallel compares the tags of the

read entries to determine whether they match. (3) Finally each matching entry has to pass through cascaded layers of multiplexers to select the longest matching prediction. Although the latency of these operations is high, the path can be easily pipelined to achieve much higher operating frequency. Based on this observation, this work explores an overriding branch predictor implementation using TAGE.

Overriding branch prediction is a technique to leverage the benefits of both fast but less accurate, and slow but more accurate predictors. This technique has been used commercially, e.g., in the Alpha EV8 microprocessors [18]. In an overriding predictor, a faster but less accurate base predictor makes a base prediction quickly in the first cycle, and then a slower but more accurate predictor overrides that decision, at a latter cycle, if it disagrees with the base prediction.

In this work, the base predictor is the simple bimodal predictor included in TAGE itself, i.e., T_0 in Fig. 2.4. The bimodal predictor provides a base prediction in the first cycle, and the tagged components of the original TAGE provide a prediction at the second cycle. Sections 5.4.3 and 5.4.4 show that an overriding TAGE predictor outperforms all the other branch prediction schemes in terms of both accuracy and maximum frequency.

With an overriding predictor, there is no guarantee that the overriding component will indeed be correct. Accordingly, it is essential that any benefits gained when the overriding component is right are higher than the performance lost when it is wrong. For this purpose, this work proposes the use of a confidence mechanism for applying overrides judiciously. Specifically, the confidence mechanism implemented is a small table with 256 entries that is indexed by eight bits from the PC. Each entry is a 10-bit saturating counter. The counter is updated whenever the basic and the overriding component disagree. When they disagree, the counter is incremented when the overriding component is correct and reset otherwise. Overrides are activated only after the counter saturates. Seznec also suggested using a similar confidence mechanism, a *statistical corrector*, in his ISL-TAGE improvement over the original TAGE [17]. There, the statistical corrector is used in a single-cycle non-overriding TAGE predictor to avoid using the tagged components whenever the bimodal component proves better. Seznec's observation was that the tagged components fail at predicting branches that are statistically biased towards a direction but not correlated to the history path. On some of these branches, TAGE often performs worse than a simple PC-indexed table, e.g., a bimodal predictor.

The confidence mechanism/statistical corrector used in this work is similar to that proposed by Jacobsen et al. [12], except that the specific statistical corrector in this work is only updated when the basic and the overriding component disagree. The specific confidence mechanism performed better than Seznec's mechanism. This is no surprise as here it is used to guide overrides in an overriding TAGE predictor. Specifically, in Nios II-f where the branch resolution latency is only two cycles, the overriding

TAGE saves one cycle for each correct override, but loses two for each incorrect override. Hence, the overriding TAGE must be very confident to make an overriding decision, which necessitates the specific statistical corrector.

As a result, this work proposes four TAGE-based designs that use one or two cycles, with or without a confidence mechanism: (1) the single-cycle TAGE, which requires TAGE to provide a prediction in one cycle (i.e., in the fetch stage), (2) the Overriding TAGE (O-TAGE), which uses just the bimodal predictor (i.e., T_0) to provide a base prediction in the first cycle, and *always* overrides the base prediction if TAGE disagrees at the end of the second cycle, (3) the single-cycle TAGE with a Statistical Corrector (single-cycle TAGE-SC), which forces the predictor to use the base prediction unless TAGE consistently disagrees over several encounters of the same event, and (4) the Overriding TAGE with a Statistical Corrector (O-TAGE-SC), which is similar to the single-cycle TAGE-SC except that TAGE overrides the base prediction in the second cycle. Fig. 4.4 demonstrates these TAGE-based designs, the target predictor is not shown in these figures for simplicity. The accuracy and critical path of the Perceptron predictor did not justify investigating an overriding configuration based on Perceptron.

The specific configuration parameters of the original TAGE are summarized in Table. 4.2.

Structure	**Storage**
T_0	20 Kbits
T_1 and T_2	12 Kbits each
T_3 and T_4	26 Kbits each
T_5	28 Kbits
T_6	30 Kbits
T_7	16 Kbits
T_8 and T_9	17 Kbits each
T_{10}	18 Kbits
T_{11}	9.5 Kbits
T_{12}	10 Kbits
Total	241.5Kbits

Table 4.2: Sizes of Tables T_i in the four TAGE based predictors.

4.3 Branch Target Predictor

Chapter 3 has shown that when using one M9K BRAM – a hardware budget on par with that of Nios II-f – eliminating the BTB and using *Full Address Calculation* (FAC) together with a RAS results in better performance [21]. It has also shown that direct branches and returns comprise over 99.8% of all branches. Implementing FAC with RAS can cover these branches with 100% accuracy, therefore having a BTB to cover all branches results in negligible improvement in target prediction accuracy.

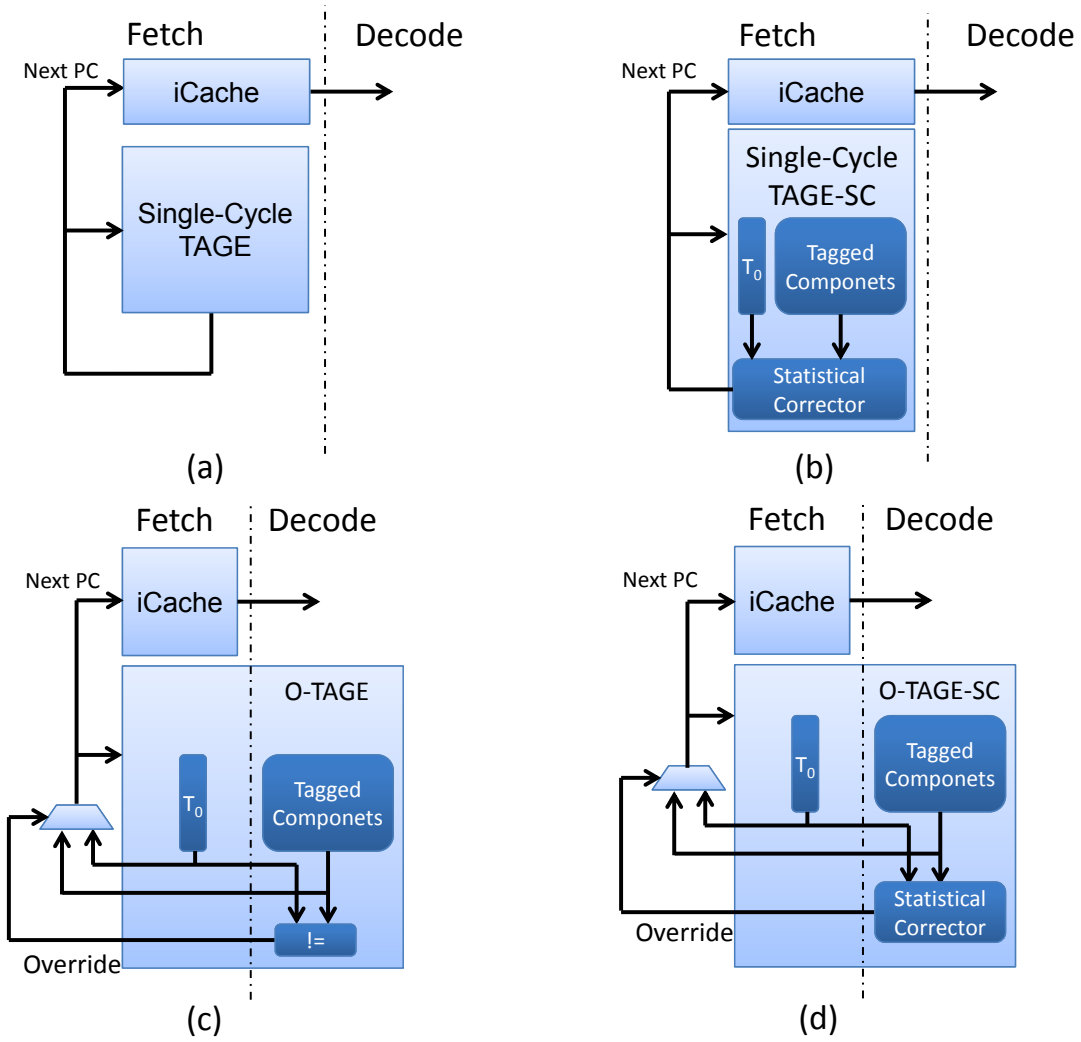

Figure 4.4: The four TAGE-based designs: (a) single-cycle TAGE, (b) single-cycle TAGE-SC, (c) O-TAGE, and (d) O-TAGE-SC.

Since, this chapter investigates how branch prediction accuracy can improve when additional hardware resources are used, adding a BTB for better target prediction coverage could improve target prediction accuracy. Accordingly, we consider reintroducing a BTB. However, simulations show that accuracy is still better without a BTB[1]. This is because when the target predictor only has FAC and RAS, it never predicts indirect branches that are not returns because it is not capable to do so. As a result, the destructive aliasing in the *direction* predictor is alleviated because fewer branches are being predicted. Based on this observation, the Perceptron and TAGE predictors in this chapter also use FAC with a 16-entry RAS as the branch target predictor.

[1]These simulation resutls are not included in this thesis.

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. Sections 5.3 and 5.4 reports the accuracy of various direction predictors, their resource usage and maximum operating frequency, and finally the overall performance of the minimalistic and the advance branch predictors respectively.

5.1 Methodology

To compare the predictors this work measures: (1) Accuracy as Mispredictions Per Kilo Instructions (MPKI), which has been shown to correlate better with performance compared to prediction accuracy alone. (2) The Instruction Per Cycle (IPC) instruction execution rate, a frequency agnostic metric that better reflects the accuracy of each predictor factoring away their latency, (3) Instructions Per Second (IPS), a true measure of performance that takes the operating frequency into account, (4) Operating frequency, and (5) Resource usage.

Simulation measures MPKI and IPC using a custom, cycle-accurate, full-system Nios II simulator. The modeled Nios II-f processor is a six-stage in-order pipeline with a two cycle branch resolution latency. The simulator boots ucLinux [1], and runs a representative subset of SPEC CPU2006 integer benchmarks with reference inputs [19]. Since Nios II instruction set does not include floating point units, this work uses the integer subset of the benchmarks. The benchmarks are compiled using the gcc ported for Nios II by Altera Corp. The simulations skip one billion instructions and run one billion instructions for each benchmark.

The evaluation of the minimalistic predictor uses a baseline predictor (Base) with a fused BTB and

bimodal predictor, as discussed in Section 3.3.1 both with 256 entries. Base does not decode instructions and thus uses the BTB and the bimodal predictor for all instructions.

The baseline predictors considered for the Perceptron and TAGE predictors are: (1) bimodal, (2) gshare and (3) gRselect. For fair comparisons, the baseline predictors are scaled to the same size as the largest Perceptron and TAGE predictors.

All the branch predictor designs were implemented in Verilog and synthesized using Quartus II 13.0 on a Stratix IV EP4SE230F29C2 chip in order to measure their maximum clock frequency and area cost. The fetch stage and parts of the execution stage that updates the branch predictors are built, while the rest of the processor is not synthesized. The resource costs only take the actual costs of the branch predictors into account. 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 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. The simulator uses a counter to count number of target mispredictions when using various target prediction mechanisms, and Fig. 5.1 reports the reduction in target address mispredictions 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.

5.3 The Minimalistic FPGA-Friendly Branch predictor

As discussed in Chapter 3, the minimalistic branch predictor uses one BRAM, the same hardware budget as Nios II-f. This section shows the results of various minimalistic branch predictor designs.

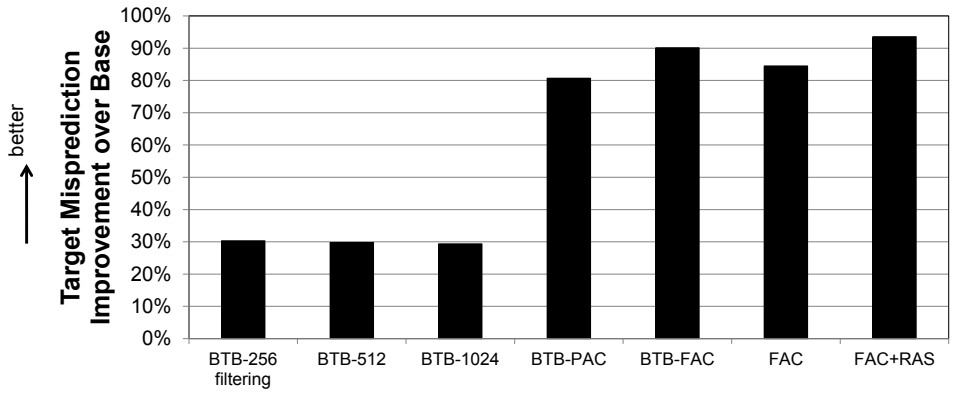

Figure 5.1: Target Address Prediction Schemes: Reduction in target address mispredictions over BASE.

5.3.1 Direction Prediction

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 predictor 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.

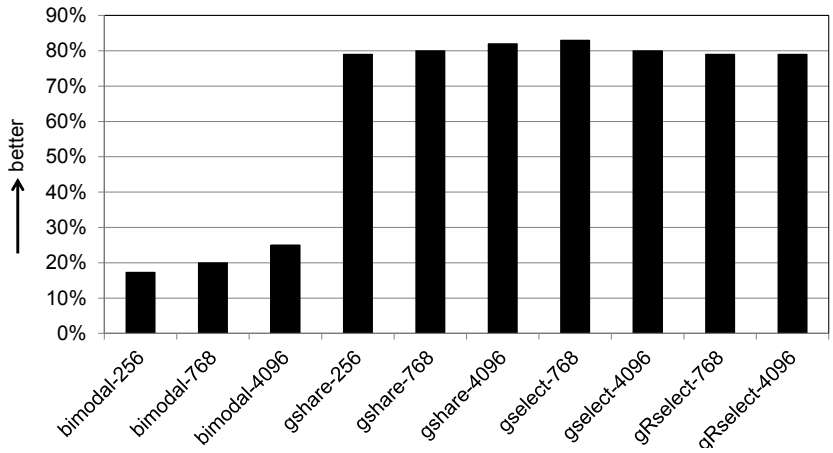

Figure 5.2: Direction Predictors: MPKI improvement over BASE.

Section 3.3.2 explained why gselect may be better to implement on an FPGA. Fig. 5.2 shows that a conventionally indexed 4K-entry gselect results in competitive accuracy, improving BASE by 80%. Section 5.3.2 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 Area and Frequency

Fig. 5.3 shows the maximum frequency and area utilization for each predictor design. The solid bars show the maximum frequency, and the patterned bars show the number of ALUTs used. All configurations use one BRAM. As expected, BASE is the fastest and least expensive. Adding instruction filtering reduces the maximum frequency from 353 MHz to 287 MHz, an 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.

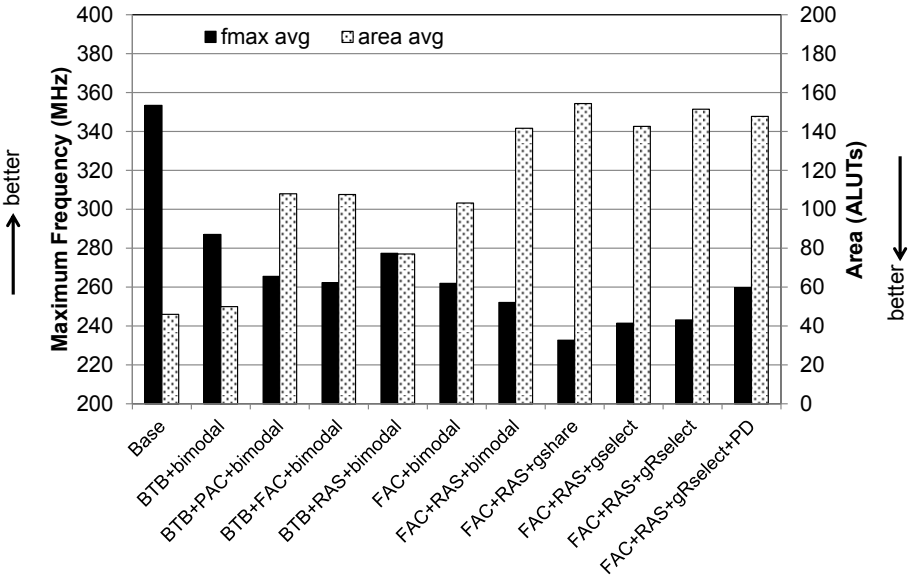

Figure 5.3: Maximum frequency and area utilisation. (PD = pre-decoding)

5.3.3 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.

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 270 MHz maximum clock speed for the processor, the maximum clock

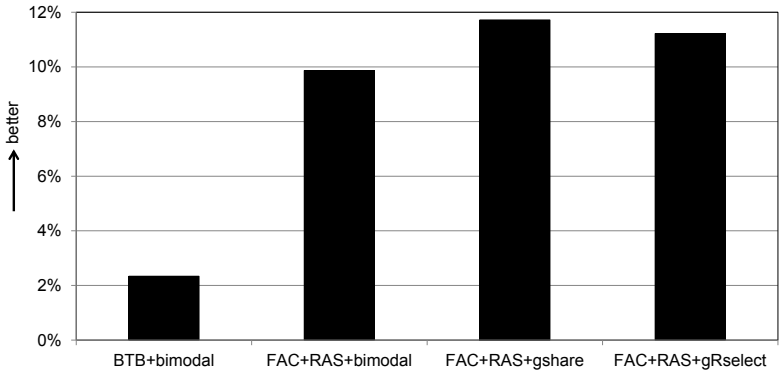

Figure 5.4: Improvement in IPC over BASE.

frequency of Nios-II-f on the Stratix IV [7]. The best performing predictor is a 4K-entry gRselect with FAC+RAS, no BTB, and that uses pre-decoded instructions.

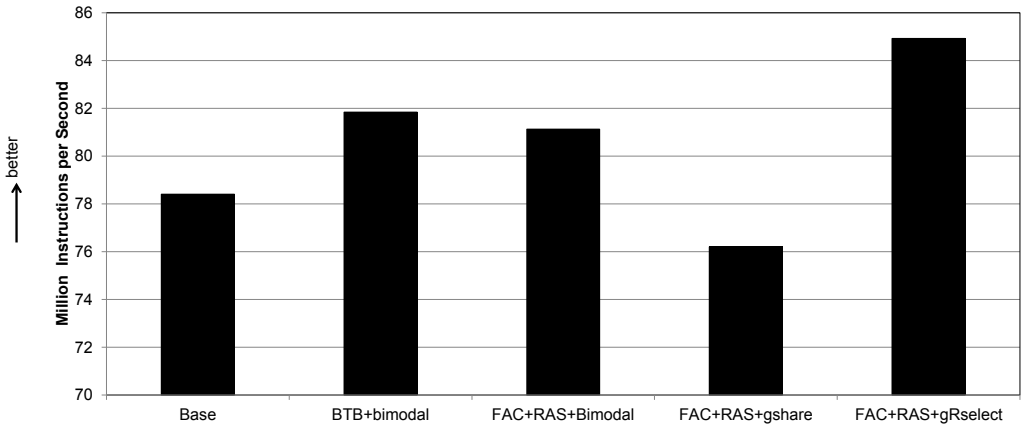

Figure 5.5: IPS comparison of processors with various predictors.

5.4 Advanced Branch predictors

This section discusses the direction prediction accuracy of Perceptron and TAGE predictors. As discussed in Section 4.3, the target predictor used in this section is also FAC+RAS. This section first presents data that justify the final design of Perceptron and TAGE configurations, then a comparison with bimodal, gshare and gRselect is presented.

5.4.1 Perceptron

This work considers Perceptron predictors with a hardware budget ranging from 1KB to 32KB. For each hardware budget, the number of global history bits is varied and the best performing Perceptron predictor within each hardware budget is chosen. Fig. 5.6 shows the most accurate Perceptron configuration for each hardware budget. All of these configurations use 16 history bits. As Section 5.4.5 will show, although the 32KB Perceptron is 3.2% more accurate than the 16KB Perceptron, its IPC saturates at the 16KB budget, therefore for the rest of this work the 16KB Perceptron predictor is used.

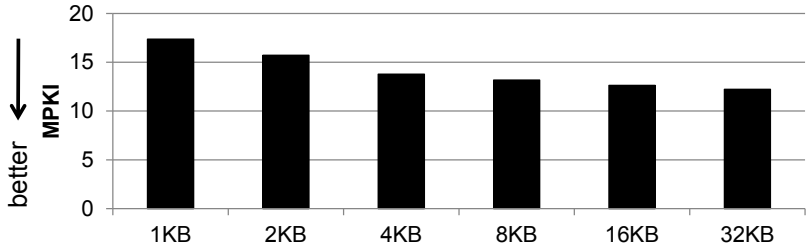

Figure 5.6: Perceptron: MPKI of the most accurate Perceptron configuration with various hardware budgets.

To determine how many HOBs the predictor should use, we took the 16KB Perceptron and experimented with all possible numbers of HOBs used. Fig. 5.7 shows the MPKI of this Perceptron when different numbers of HOBs are used. The data shows that using three HOBs degrades accuracy by less than 1% compared to using all eight bits. However, the MPKI doubles when using only two HOBs. Therefore the the implemented Perceptron designs use three HOBs to improve operating frequency without affecting accuracy.

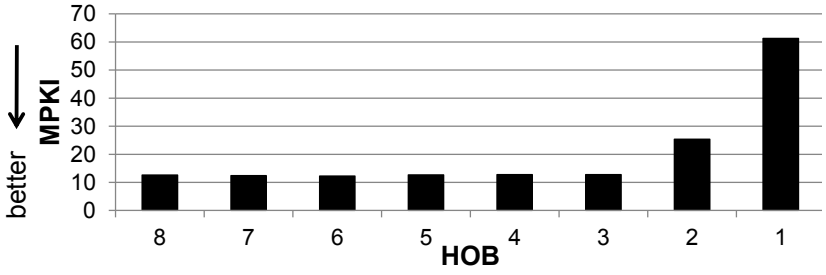

Figure 5.7: Perceptron: MPKI when using different number of HOBs for the most accurate perceptron configuration.

The IPC-wise best performing Perceptron uses 16 global history bits. It has a 16KB perceptron table, which stores 1K perceptrons. Each perceptron contains 16 8-bit weights with the arrangement discussed in Section 4.1.3. It also has a 6KB complement table that stores three HOBs per weight in its 2's complement form to improve frequency. Thus this thesis names this best performing Perceptron

the *16KB+6KB Perceptron*. This thesis will follow this convention in the remaining sections, but the hardware budgets for various Perceptron configurations in the remaining figures only refer to their perceptron table sizes.

5.4.2 TAGE

All TAGE configurations studied in this work use Seznecs original table configurations [2]. As listed in Table 4.2, the TAGE predictor alone uses 241.5Kbits or ~30.2KB storage. To design the statistical corrector, we vary number of table entries as well as the widths of the saturating counters to determine the best design. The proposed statistical corrector uses 2.5Kbits, therefore the total storage of TAGE-SC and O-TAGE-SC is 244Kbits or 30.5KB. All TAGE variations are within the 32KB budget. Adjusting the TAGE's size is a non-trivial task, moreover, the results of this work show that the 32KB O-TAGE-SC outperforms the other predictors. Accordingly, we do not vary the size of TAGE in this work. Fig. 5.8 shows the MPKI of the four designs that incorporate TAGE. It shows that the single-cycle and overriding predictors have virtually identical MPKI. The statistical correcter improves MPKI by ~2.4x.

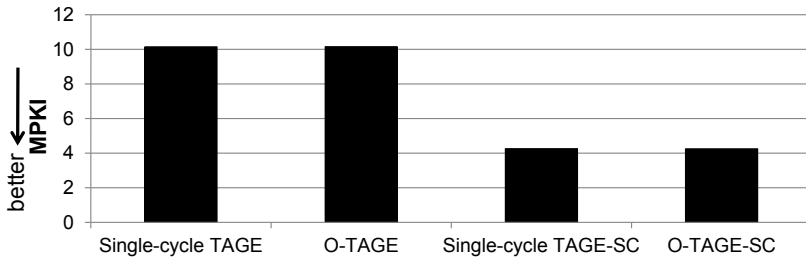

Figure 5.8: TAGE: MPKI of the four TAGE variations.

5.4.3 Accuracy Comparison

For fair comparisons, we scale bimodal, gshare and gRselect from 1KB to 32KB, which is the same hardware budget as TAGE and the largest Perceptron considered in this work. Fig. 5.9 shows the MPKI of various direction predictors. The TAGE variations use 32KB. All the branch predictors get more accurate as the hardware budget increases. The single-cycle TAGE-SC is the most accurate, followed by O-TAGE-SC with less than 0.06% difference. The single-cycle TAGE-SC is ~2.3x more accurate than the 32KB gRselect and the 32KB gshare.

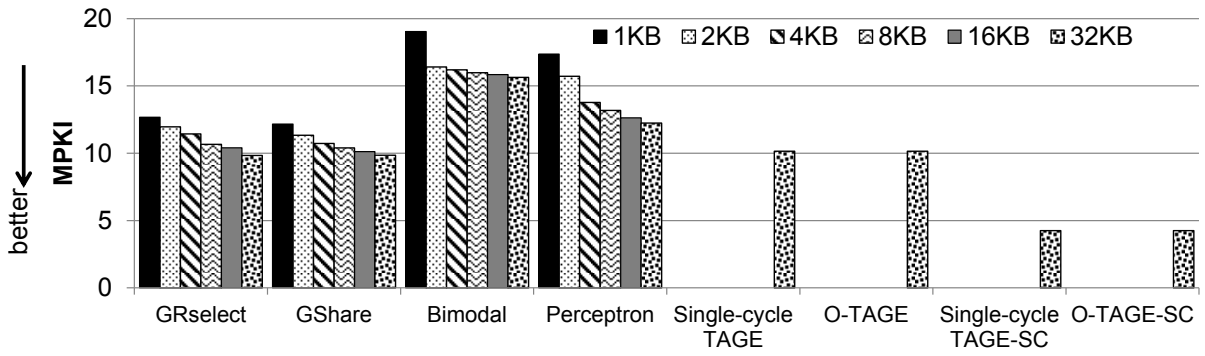

Figure 5.9: MPKI of the direction predictors.

5.4.4 Frequency

Fig. 5.10 shows the maximum operating frequency for each branch prediction scheme and for various hardware budgets.

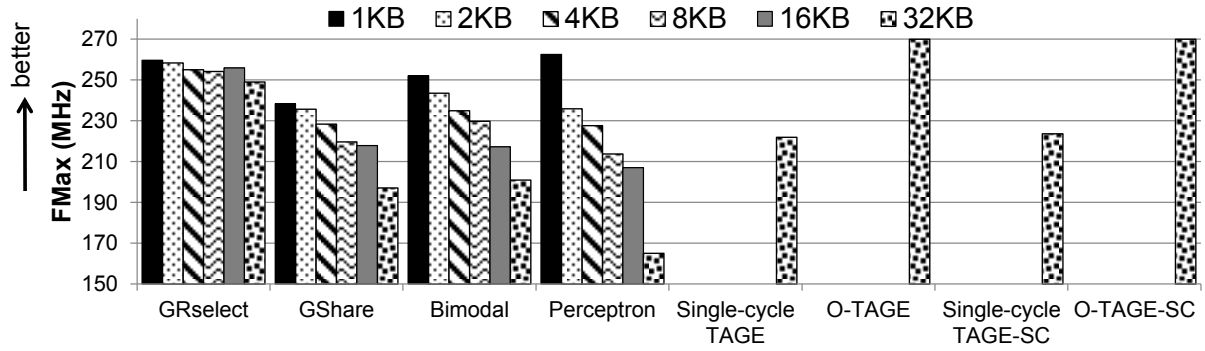

Figure 5.10: Maximum operating frequency of the considered branch prediction schemes with various hardware budget.

The fastest predictors are O-TAGE-SC and O-TAGE, both of them are capped at 270 MHz, the maximum frequency for Nios II-f on Stratix IV C2 speed grade devices [7]. The 1KB+384B Perceptron and the 1KB gRselect follow the O-TAGE variations. The maximum frequency of gshare, bimodal and Perceptron drop rapidly with increasing size, while gRselect's frequency does not suffer too much. Despite that the logic is larger and more difficult to place and route, the table indexing of gRselect comes from the GHR register. GRselect reads a wide entry and then using bits from the PC to select the appropriate ones. The indexing of gshare, bimodal and Perceptron uses the predicted PC. The PC is both the input and the output of the branch predictor. This loop forms the critical path of gshare, bimodal and Perceptron, which quickly gets slower as the sizes of the predictors increase. The single-cycle TAGE-SC operates at 223.7MHz, which is 14.8% slower than the 1KB gRselect and 17.2% slower

than O-TAGE-SC.

5.4.5 Performance and Resource Cost

Fig. 5.11 shows the IPC of the predictors. Although the MPKI of the 32KB+12KB Perceptron is higher than the 16KB+6KB Perceptron, they deliver identical IPC. The single-cycle TAGE-SC has the highest IPC, however, as this section shows, its high IPC cannot amortize the slowdown in operating frequency. O-TAGE is much faster, but its IPC drops significantly. Finally, the IPC of O-TAGE-SC is within 0.5% of the single-cycle TAGE-SC.

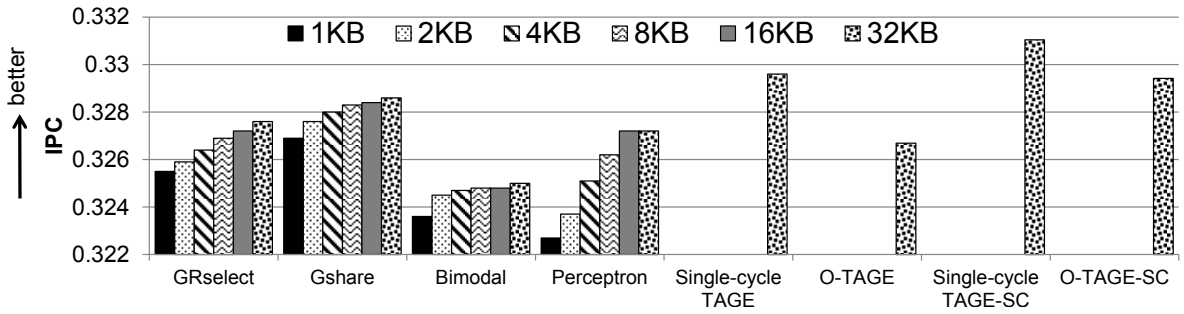

Figure 5.11: IPC of the considered branch predictors.

IPC is a measurement that does not take operating frequency into consideration. The actual performance of a processor is measured by Instructions Per Second (IPS), which is the product of IPC and the maximum operating frequency. Fig. 5.12 reports the overall performance in terms of IPS.

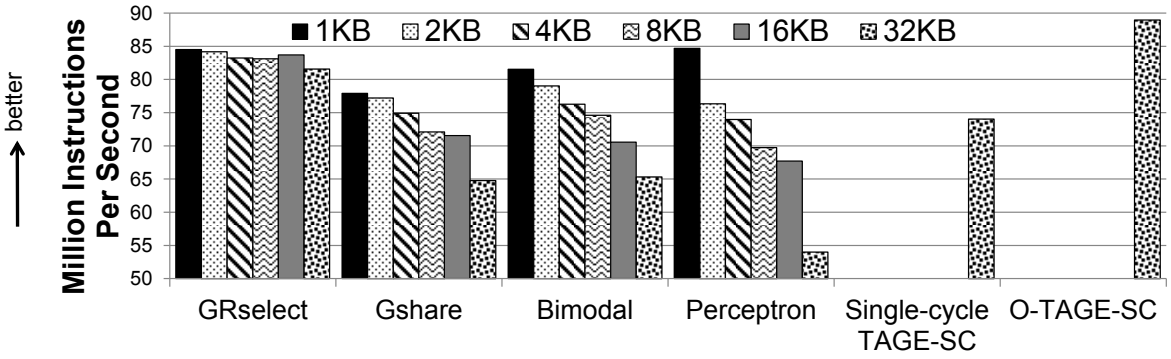

Figure 5.12: Processor IPS comparison with various predictors.

The IPS of gRselect, gshare, bimodal and perceptron drops as they scale, therefore we chose the smallest configurations of these schemes to maximize IPS. The best performing predictor is O-TAGE-SC, which delivers 5.2% higher IPS than the 1KB gRselect. Although the single-cycle TAGE-SC is the most

accurate, its IPS is lower than the best performing predictor in all other prediction schemes because its latency is too high. The 1KB Perceptron ends up being 0.2% better than the 1KB gRselect, because of the optimization efforts into improving its frequency.

Finally, Fig. 5.13 shows the resource usage in term of ALUTs used. O-TAGE-SC uses 2.93x ALUTs and Percpetron uses 6.45x ALUTs as the 1KB gRselect.

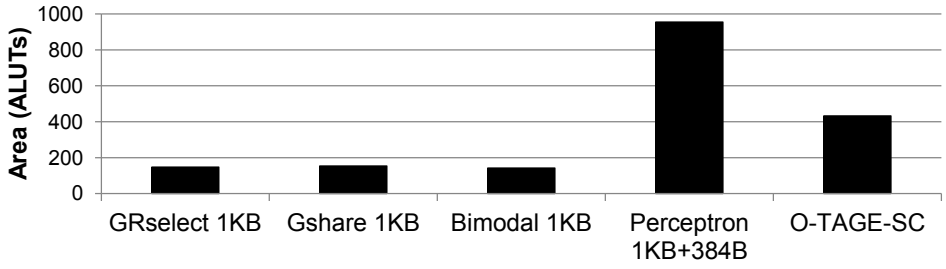

Figure 5.13: ALUTs usage comparison with various predictors.

Table. 5.1 summarizes the IPS, MPKI and resource usage of the best performing configurations of each branch prediction scheme in ascending order of IPS.

	Million IPS	**MPKI**	**ALUTs**	**Storage**
1KB gshare	77.90	12.16	154	1 M9K BRAM
1KB bimodal	81.55	19.04	142	1 M9K BRAM
1KB gRselect	84.50	12.68	148	1 M9K BRAM
1KB+384B Perceptron	84.70	17.37	955	18 MLABs (64 × 10 bits each)
O-TAGE-SC	88.94	4.25	433	34 M9K BRAM

Table 5.1: Summary.

5.5 1KB gRselect vs. O-TAGE-SC

The previous section has shown that O-TAGE-SC is ~3x as accurate as the 1KB gRselect, but it only outperforms the 1KB gRselect by 5.2% in terms of IPS. Moreover, only ~1% out of the 5.2% IPS gain comes from better IPC and the rest comes from faster operating frequency. Fig. 5.14 and Fig. 5.15 shows the MPKI and IPC of the 1KB gRselect and O-TAGE-SC over all the benchmarks used in this work. The average MPKI and IPC of the two predictors are shown as the two right-most columns.

From Fig. 5.14, O-TAGE-SC is 1.98x more accurate on average than the 1KB gRselect over all the benchmarks. The best case is *libquantum*, where O-TAGE-SC is 239.9x more accurate, and even in the worst case *astar*, O-TAGE-SC is still 19% more accurate. However, from Fig. 5.15, O-TAGE-SC is only

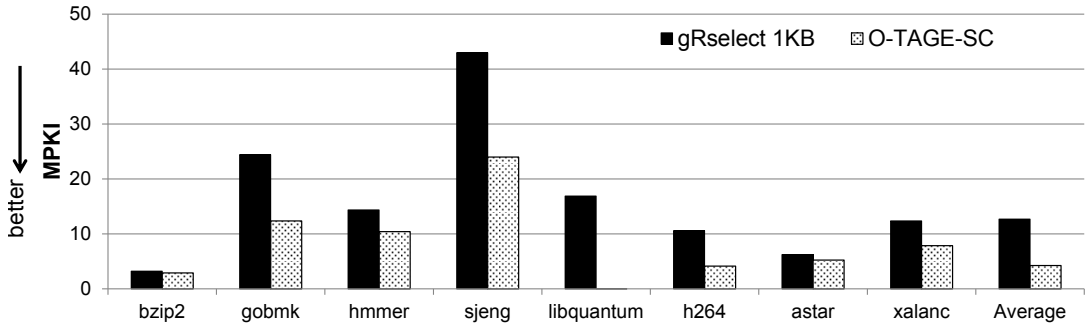

Figure 5.14: MPKI of the 1KB gRselect and O-TAGE-SC over all the benchmarks and the average MPKI.

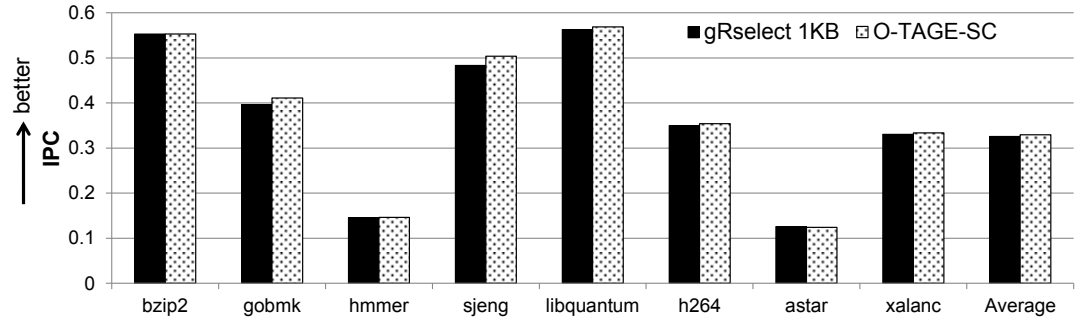

Figure 5.15: IPC of the 1KB gRselect and O-TAGE-SC over all the benchmarks and the average IPC.

1.2% better in terms of IPC. The best case is *sjeng*, where O-TAGE-SC is 4.2% better, and the worst case is *astar*, where O-TAGE-SC is 1.5% worse.

Looking at *astar*, O-TAGE-SC is 19% more accurate, but ends up 1.5% worse in IPC. The reason is that the MPKI of O-TAGE-SC counts the mispredictions of the final decision. In the scenario where TAGE correctly overrides the base predictor, the predcition is *functionally* correct, therefore the simulator considers it a correct prediction. However, it still takes one cycle (assuming a instruction cache hit) for the processor to fetch the instruction recommended by TAGE after the override. This extra cycle is not reflected in the MPKI of the predictor, but influences IPC. The accuracy advantage O-TAGE-SC in this benchmark is not enough to compensate the penalty of fetching the new instruction when O-TAGE-SC decides to override.

Another interesting benchmark is *libquantum*. O-TAGE-SC is ~240x more accurate, however it is only 1.5% better in IPC. Besides the reason discussed earlier, a more important factor is that Nios II-f has a very short single-issue in-order pipeline. The branch resolution latency in Nios II-f is only two cycles, which means the cycle count difference between a correct and an incorrect prediction is only two. This short pipeline limits the ultimate gain of branch prediction, therefore, although O-TAGE-SC is

vastly superior in accuracy, the IPC improvement is limited.

Chapter 6

Conclusions

This thesis studied the implementation of branch predictors for general purpose soft processors. It targeted high frequency and high accuracy branch predictors for pipelined processors, and explored various branch predictor designs. These designs were combinations of a branch target buffer, a return address stack, commonly used simple and advanced direction predictors, pre-decoding, in-fetch instruction decoding, and target address calculation. Several FPGA-specific optimizations were proposed resulting in a branch predictor that is FPGA-friendly in that it offers high accuracy and high operating frequency.

In summary, this thesis makes the following contributions:

(1) This thesis explores the tradeoffs of various combinations of structures for branch target prediction on FPGAs. It has shown that eliminating the BTB and using the combination of FAC and RAS as the branch target predictor is the best design for both the minimalistic predictor and the more advanced O-TAGE-SC.

(2) This thesis studies the FPGA implementations of bimodal, gshare and gselect. It identifies the critical paths of these predictors. Accordingly, this thesis proposes a FPGA-friendly predictor gRselect that reverses the order of indexing to improve maximum operating frequency. It has shown that within the same hardware budget, gRselect together with FAC+RAS is the best performing branch predictor.

(3) This thesis shows that scaling bimodal, gshare and gRselect up to 32KB improves IPC marginally, but slows down the predictor significantly. The best performing bimodal, gshare and gRselect configurations all uses 1KB hardware budget.

(4) This thesis studies the FPGA implementations of the state-of-the-art Perceptron and TAGE branch direction predictors. Several FPGA implementation optimization techniques were proposed to achieve high operating frequency. It explored the designs tradeoffs of Perceptron and TAGE, and pro-

posed O-TAGE-SC, an overriding predictor that delivers 5.2% better instruction throughput over the 1KB gRselect.

Because an FPGA is a very different substrate than an ASIC, the design tradeoffs must be re-evaluated when designing branch predictors for soft processors. Not only the structures of the branch predictors must map well onto FPGAs, techniques that are impractical on an ASIC should also be considered.

Although O-TAGE-SC is ~3x more accurate than the 1KB gRselect, the IPS improvement is much smaller due to the processor's simple in-order pipeline. Considering more accurate branch predictors such as the ISL-TAGE for Nios II would only improve IPS marginally. This work recommends the 1KB FAC+RAS with gRselect for Nios II-f because the 5.2% improvement does not justify the 32x more storage used. Future work may consider investigating the benefits of implementing O-TAGE-SC for more elaborate soft processors, e.g., an Out-of-Order soft processor, which requires a more accurate branch predictor.

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] Kaveh Aasaraai and Amirali Baniasadi. A power-aware alternative for the perceptron branch predictor. In Lynn Choi, Yunheung Paek, and Sangyeun Cho, editors, *Advances in Computer Systems Architecture*, volume 4697, pages 198–208. 2007.

[4] Altera Corp. Nios II Processor Reference Handbook v9.0. 2009.

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

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

[7] Altera Corp. *Nios II Performance Benchmarks*, Nov. 2013.

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

[9] O. Cadenas, G. Megson, and D. Jones. A new organization for a perceptron-based branch predictor and its FPGA implementation. In *Proc. IEEE Computer Society Annual Symposium on VLSI.*, pages 305–306, May 2005.

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

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

[12] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *In Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[13] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 67–76. ACM, 2000.

[14] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 34–42, New York, NY, USA, 1991. ACM.

[15] D.C. Lee, P.J. Crowley, Jean-Loup Baer, T.E. Anderson, and B.K. Bershad. Execution characteristics of desktop applications on windows nt. In *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, pages 27–38, Jun 1998.

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

[17] André Seznec. A 64 kbytes isl-tage branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[18] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proc. 29th Annual International Symposium on Computer Architecture.*, pages 295–306. IEEE, 2002.

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

[20] C. S. Wallace. A suggestion for a fast multiplier. *Electronic Computers, IEEE Transactions on*, EC-13(1):14–17, Feb 1964.

[21] Di Wu, K. Aasaraai, and A. Moshovos. Low-cost, high-performance branch predictors for soft processors. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013.

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

[23] Xilinx Inc. *7 Series FPGAs Overview*, Feb. 2014.

[24] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 257–266, New York, NY, USA, 1993. ACM.

[25] 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.