Harper Grace Challinor

One predictor to rule them all: A parameterisable predictor for Toooba

Computer Science Tripos – Part II

Trinity Hall

2025

UNIVERSITY OF
CAMBRIDGE
DEPARTMENT OF
COMPUTER SCIENCE
AND TECHNOLOGY

Acknowledgements

My sincere gratitude and appreciation to Dr. Jonathan Woodruff for supervising me, to Dr. Jack Hughes for his useful comments as my director of studies, to Dr. Rishiyur Nikhil for his support with Bluespec SystemVerilog and for reviewing the pull request for my bug-fix, to Karl Mose for his assistance to my very specific technical questions and for running the computer architecture paper reading group, and to Prof. Jamie Vicary for his kind words after the "how to write dissertation" session when I thought I didn't have it in me.

I was immensely supported throughout this project by my loving fiancée Sophia[1], my wonderful friends, and my caring family. Thank you all.

[1] Check out her animated undergraduate final project "Creature City" on YouTube!

Declaration of originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 2441F, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. The project required the approved use of ChatGPT[2] and such use is acknowledged in the text at the relevant sections. I am content for my report to be made available to the students and staff of the University.

I utilised work from another student with Blind Grading Number 2373A from before the projects began – see section 2.2.

Date: 18th May 2025

[2]This was for a script that did not tackle any of the intrinsic complexity of the project – see section 2.8.

Proforma

Candidate number:	2441F
Project title:	One predictor to rule them all: A parameterisable predictor for Toooba
Examination:	Computer Science Tripos – Part II
Year:	2025
Dissertation word count:	10,085[3]
Project code line count:	3,012[4]
Project originator:	The supervisor
Project supervisor:	Dr. Jonathan Woodruff

Original aims of the project

The original goals were to: implement a parameterisable prediction module in Bluespec SystemVerilog for Toooba using the organisation of TAGE; include unit tests; instantiate the predictor for branch direction prediction; instantiate the predictor for branch target prediction; and evaluate the differences in performance on benchmarks.

Work completed

I implemented several predictors for branch direction prediction and next-address prediction. The organisation was changed from TAGE to Gselect for implementation feasibility. The original goals are otherwise met and a positive result is reported.

Special difficulties

None.

[3] According to the Overleaf editor.

[4] Counted manually, including number of insertions from `git diff --shortstat` for fork submodules.

Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
- 1.2 Branch prediction . . . 8
  - 1.2.1 Branch direction prediction . . . 8
  - 1.2.2 Branch target prediction and next-address prediction . . . 9
- 1.3 An example of parameterisability: saturating counters . . . 9
- 1.4 Success criteria . . . 10
- 1.5 Contributions . . . 11
- 1.6 A note on software licenses . . . 11

2 Preparation **12**
- 2.1 The Gselect branch direction predictor . . . 12
- 2.2 ChampSim . . . 13
- 2.3 Bluespec SystemVerilog . . . 13
- 2.4 Toooba . . . 14
  - 2.4.1 Front-end . . . 14
  - 2.4.2 Branch direction prediction interface . . . 15
  - 2.4.3 Next-address prediction interface . . . 16
  - 2.4.4 Interface differences . . . 16
- 2.5 Benchmarks . . . 17
- 2.6 Starting point . . . 17
- 2.7 Requirements analysis . . . 18
- 2.8 Engineering techniques . . . 19
  - 2.8.1 Best practices . . . 19
  - 2.8.2 Composable modular design . . . 19
  - 2.8.3 Project management . . . 20

3 Implementation **21**
- 3.1 A security bug-fix for Toooba . . . 21
- 3.2 Repository overview . . . 22
- 3.3 From the dumbest predictor to Gselect . . . 23
  - 3.3.1 Always taken . . . 23
  - 3.3.2 Local one-level . . . 23
  - 3.3.3 Gselect . . . 24
- 3.4 `ValueWithConfidence` . . . 24
- 3.5 Designing for the interface differences . . . 25
- 3.6 Dealing with the interface differences . . . 26
  - 3.6.1 Alternate BDP interface . . . 27
  - 3.6.2 Alternate NAP interface . . . 27
  - 3.6.3 Modifying Toooba . . . 28
  - 3.6.4 Modifying my Gselect . . . 28
- 3.7 `TRegFile`: Transactional register file . . . 29
- 3.8 Gselect next-address predictor . . . 30
- 3.9 Emergent design . . . 30
- 3.10 The parameterisable predictor . . . 31
  - 3.10.1 Instantiations . . . 32

3.11 Hash-tagged Gselect 33

4 Evaluation **35**
4.1 Predictor monitoring 35
4.2 Performance and accuracies 36
4.3 Success criteria and project requirements 38
4.4 Investigation into next-address prediction instantiation 39
4.5 Hash-tagged Gselect 39

5 Conclusions **41**
5.1 Contributions 41
5.2 Future work 41
5.3 Personal and professional development 42

Bibliography

Appendix A: Toooba TLB micro-cache bug-fix

Glossary

Project proposal

Chapter 1

Introduction

1.1 Motivation

Modern processor microarchitectures execute instructions speculatively to avoid pipeline bubbles and increase performance [1, chapter 3]. Without doing so, instructions that change the program counter (PC) or have other effects must finish execution before the next instructions are fetched, meaning that each clock cycle is not fully utilised – there is a "bubble".

Speculative execution is informed by several modes of prediction, including branch direction prediction (BDP)[5] – a Boolean prediction for whether a branch is taken; branch target prediction – which 64-bit address is next; memory dependence prediction; and prefetching. Prediction accounts for a significant portion of transistor and area budgets – the resources available for building a microprocessor – as well as development and verification time. This significance can be seen from the paper "Evolution of the Samsung Exynos CPU Microarchitecture" [2] dedicating four of its 11 chapters to prediction[6], as well as the large area dedicated to branch prediction (direction and target) in AMD's Zen 2 core, shown in figure 1.1.

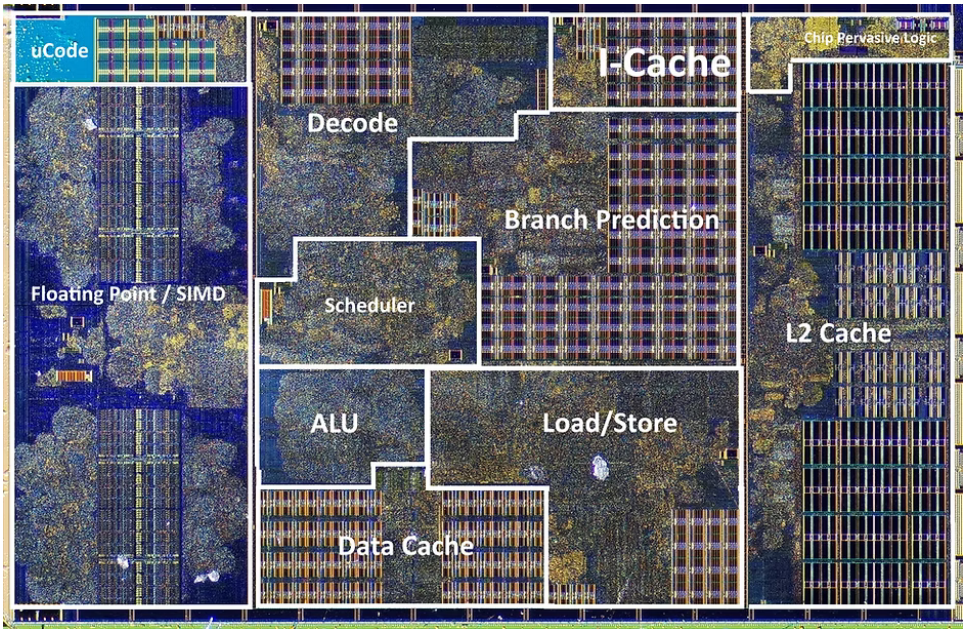

Figure 1.1: An annotated die shot of a Zen 2 core.
Image credit: die shot by Fritzchens Fritz [3], annotated by Ashley Webb [4] based on an AMD slide deck [5] presented at ISSCC 2020, accompanying a paper on the microarchitecture [6].

[5]BDP is not a widely used acronym in literature; it is more commonly called branch prediction but I find that too vague. I overload it to refer to both branch direction prediction and branch direction predictors – hardware that performs these predictions.

[6]One of these also concerns cache management.

The ideas and organisation of a predictor for one mode can be reused for another mode. For example, TAGE, the state-of-the-art BDP, was introduced with a variant for branch target prediction [7] and has been shown to be effective for memory dependence prediction in simulation [8].

In this dissertation, I have investigated the possibility and effectiveness of using one parameterisable predictor module for two modes of prediction – BDP and next-address prediction (NAP)[7] – effectively reusing source code between them. After reviewing the literature, I believe this approach to be novel.[8] The predictor is implemented for Toooba, an open-source RISC-V CPU written in Bluespec SystemVerilog (BSV).

Benefits of a parameterisable predictor One predictor that can be instantiated for multiple modes of prediction carries the following benefits.

- Less time in total is required for development and verification. This allows more time to optimise the design.
- Speculative execution has caused vulnerabilities in commercial microprocessors. In particular, Spectre variant two involves an attacker training the branch target buffer (BTB) [10] – hardware that performs branch target prediction. BDP has also been successfully attacked, with BranchScope [11] for one-level[9] predictors and Bluethunder [13] for two-level predictors. Reusing source code between prediction modes decreases the attack surface for such vulnerabilities.
- Advanced techniques and predictor organisations may be easily shared between modes of prediction; e.g., there are numerous techniques for BDP [12] that other modes may benefit from.

1.2 Branch prediction

Control hazards to pipelined processors occur in the presence of instructions that change the PC – we do not know which instructions are next until these instructions finish execution. However, an educated guess can be made, although it bares the cost of the hardware overhead to make these predictions and deal with incorrect predictions (mispredictions/misspeculation). Pipeline bubbles can be made smaller or avoided entirely when the prediction is correct, and high-accuracy predictors can make this the common case.

This brief introduction will be limited to the branch and jump instructions in the RISC-V instruction set architecture [14, section 2.5].

1.2.1 Branch direction prediction

Branch instructions (branches) have a condition and an offset. The branch is taken when the condition is met; otherwise, it is not taken. The PC following a branch either points to the next instruction in program-order when the branch is not taken, or becomes offset by an amount statically specified in the instruction when the branch is taken.

[7] NAP is an alternative to branch target prediction – see § 1.2.2. I overload NAP in the same manner as BDP to refer to both next-address prediction and next-address predictors.

[8] This approach should not be confused with omnipredictors, which share a unified memory across modes of prediction [9].

[9] The number of levels refers to how many prediction tables the predictor has [12].

A prediction for a branch is binary – either 'taken' or 'not taken' – and can be represented with a Boolean value; this is referred to as the direction of the branch and is a dynamic property. There are only two possible next instructions after a branch.

1.2.2 Branch target prediction and next-address prediction

Jump instructions (jumps) are unconditional and set the PC to an offset from a base of either the jump's address (the current PC) or the value in an architectural register. When a register value is used as the base, the next instruction could be located anywhere in the address space; therefore, the target cannot be statically known.

A prediction for a register-based jump is the target address of the jump, a dynamic property. There are many possible next instructions after a jump.

Branch target buffers BTBs are a common form of branch target prediction hardware. They are a cache of branch addresses to their targets, usually with tagged entries [15].

Next-address prediction "Next-address prediction", though not used in literature, is a narrower term that I prefer to "branch target prediction" to mean the use of a BTB on every instruction's address, regardless of the instruction type. I use this term because it describes the way Tooba uses its BTB more aptly.

During the instruction fetch stage, the next address or addresses to fetch from are retrieved from the BTB without knowing if the current instruction or instructions are jumps or branches yet, since this is before the instruction decode stage.

Typically, a BTB does not store an entry for instructions that cannot modify the PC; therefore, a miss is interpreted as PC + 4 (the next instruction in program-order). NAP is also performed for jumps that offset the new PC from the address of the jump (the PC), for which the target can be statically known; a BTB should only get this case wrong once – a compulsory miss – unless aliasing/eviction occurs.

When using the compressed extension, as Tooba does, NAP is performed for each instruction fragment – two bytes that may be the entirety or half of an instruction. BTB misses are instead interpreted as PC + 2 and targets are only given for the second half of uncompressed branches and jumps – the first half should produce a miss, pointing to the second half.

Relation between next-address prediction and branch direction prediction NAP is performed on all instructions, so it overlaps with the role of BDP. Since NAP is performed earlier in the pipeline than BDP, the NAP can be initially followed and then overruled by a BDP if the instruction is a branch. The combination of both techniques can further reduce the penalty for branches to zero cycles in the best case, completely avoiding a pipeline bubble; whereas BDP alone would still have a small bubble.

In the next section, we will see how hardware may be adapted to work for both of these modes.

1.3 An example of parameterisability: saturating counters

Two-bit saturating counters – a form of finite-state machine[10] that count up and down but cannot go above or below boundary values – are a commonly used element in a

[10] No start state is specified.

variety of predictors [16, 17] because they add hysteresis when the same result occurs multiple times. When used in a one-level BDP, a branch's counter is incremented when the branch is taken and decremented when it is not taken, as shown in figure 1.2a. The next prediction is the most significant bit of the counter.

This is isomorphic to a Boolean "remembered value" with a one-bit saturating counter of 'confidence', shown in figure 1.2b. The next prediction is the remembered value.

If we swap the Boolean for another type, the predictor can be used for another mode. For example, a NAP will need an address to be remembered. This idea forms the basis for the parameterisable predictor. We could also parameterise the number of bits for the confidence. Section 3.4 describes my implementation of this parameterisable counter.

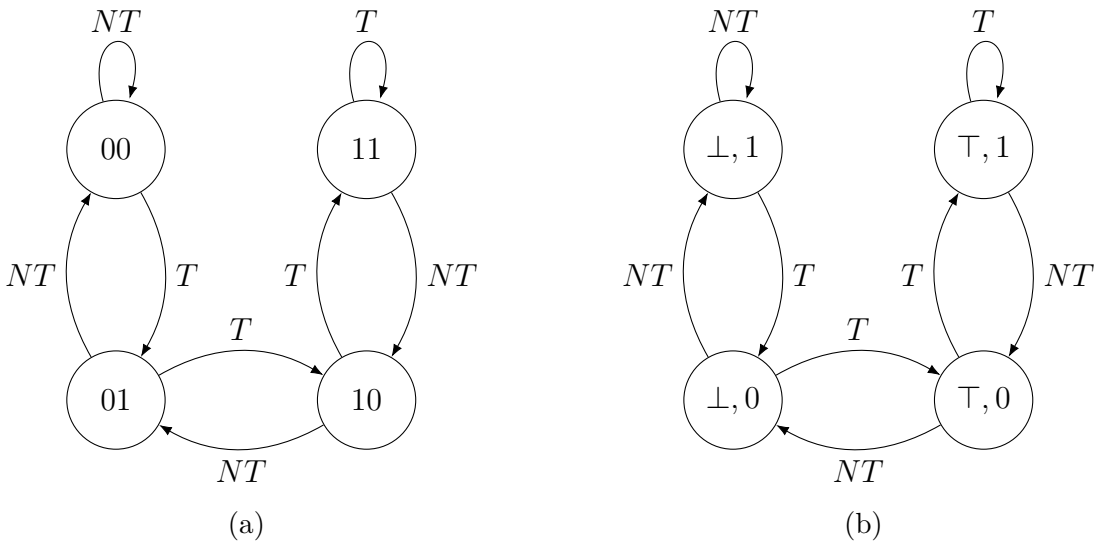

Figure 1.2: Isomorphism between a two-bit saturating counter (a) and a Boolean remembered value with one-bit of confidence (b).
NB: T and NT mean taken and not-taken branches respectively.

1.4 Success criteria

The success criteria presented here differ from the criteria presented in my project proposal: the base predictor organisation is Gselect instead of TAGE because it is more feasible to implement; the structure of the criteria is condensed; and the focus on tests has been removed because they demonstrate a professional approach to software engineering (§ 2.8) rather than project success.

1. Implement a Gselect predictor in BSV that is parameterisable in the data type it predicts and the global history it keeps.

2. Instantiate the predictor for BDP in Toooba and obtain similar performance to Toooba's current Gselect BDP.

3. Instantiate the predictor as a BTB in Toooba for NAP and evaluate it against Toooba's BTB.

This criteria is expanded upon when we analyse the project requirements in section 2.7.

1.5 Contributions

This dissertation and the underlying project (open source on GitHub with appropriate licensing) make several contributions:

- a description of Toooba's front-end (§ 2.4.1), which serves as a good reference for anyone working on Toooba (including future Part II students – I certainly would have benefitted from this!);
- an accepted bug-fix for a failing test in Toooba (§ 3.1) with security implications, which also affects CHERI-RISC-V;
- a well-tested BSV datatype and function for a remembered value with confidence, as described in section 1.3 (`ValueWithConfidence` – § 3.4);
- a well-tested BSV module for register files that support multiple writes per cycle (`TRegFile` – § 3.7);
- a modification to Toooba to pass "prediction tokens" rather than "training information" through the pipeline, supporting the use of alternate BDP and NAP interfaces and giving power benefits (§ 3.5);
- a parameterisable Gselect predictor, which my review of prediction literature suggests to be novel, for Toooba (§ 3.10), carrying benefits of less time for development and verfication, a decreased attack surface for attacks like Spectre, and allowing techniques and organisations to be shared between modes of prediction;
- an investigation into the performance of instantiations of the parameterisable predictor for BDP and NAP (§ 4); and
- a parameterisable predictor following a custom organisation – hash-tagged Gselect (§ 3.11) – and an investigation into the performance of its instantiations for BDP and NAP (§ 4.5).

1.6 A note on software licenses

This project uses ChampSim and Toooba. They are within git submodules containing forked (modified) versions of both software. ChampSim is licensed under the Apache 2.0 license which I have respected by retaining the license and noting where modifications or additions have been made. Toooba is licensed under a mixture of the MIT license and the Apache 2.0 licenses, which I have respected by retaining the licenses and noting where modifications of additions have been made. See the "README" in the source-code submission for clarifications.

Chapter 2

Preparation

This section covers the Gselect BDP (§ 2.1), which is the organisation used for the parameterisable predictor; the ChampSim trace simulator (§ 2.2), which served as an initial development environment; the programming language BSV (§ 2.3), which Tooba and the parameterisable predictor are written in; and the open-source CPU Tooba, including its interfaces for BDP and NAP and their differences (§ 2.4) – included because the parameterisable predictor is instantiated for BDP and NAP within Tooba and has to account for the interface differences. We then cover the benchmarks (§ 2.5) used for evaluation and for guiding the implementation; the project's starting point (§ 2.6); the project requirements (§ 2.7), which expand on the success criteria; and the engineering techniques used throughout (§ 2.8).

2.1 The Gselect branch direction predictor

The parameterisable predictor uses an underlying organisation of Gselect.

The Gselect BDP, for which Tooba has an implementation, combines local and global history. As shown in figure 2.1, Gselect predicts the direction of a branch by indexing the prediction table (PT) of saturating counters; the branch is predicted to be taken iff the most significant bit of the selected counter is 1. The index is a concatenation of select bits from the branch's address and the global history – bits representing the most recent branch directions across all branches, stored in a shift register called the global history register (GR). The inclusion of global history allows correlation between branches to be exploited when execution follows common paths.

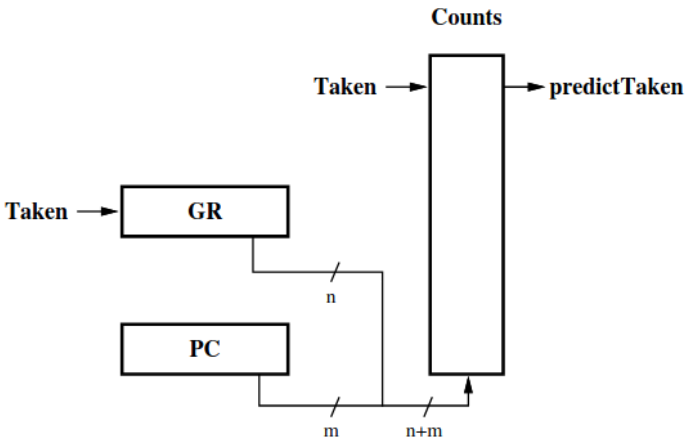

Figure 2.1: The organisation of the Gselect BDP,
using m bits from the PC and n recent global branch directions.
Image credit: Scott McFarling [18].

Gshare is another BDP which is extremely similar to Gselect, but combines the branch address and GR via exclusive-OR rather than concatenation [18]. It can be considered a "sister" predictor to Gselect.

BDPs that employ global history (with particular mention of Gshare) benefit from immediate, albeit speculative, updates to the GR [19]. This technique is used in Toooba's implementation of Gselect. It requires a rollback mechanism to remove incorrect history.

Gselect was chosen for this dissertation because it strikes a good balance between being simple enough to implement in a reasonable amount of time (c.f. TAGE) and complex enough (dynamic with a combination of local and global history) to be worth parameterising, with the type used for global history items being an important parameter. Gselect as described by Scott McFarling [18] requires only one PT[11], unlike its predecessor which had multiple PTs, indexed by bits from the address of the branch only, and then one of these results is chosen by the GR [20]. Gshare would have been another good option.

I implemented Gselect myself rather than using Toooba's to ensure I understand it fully and am able to extend it. It was implemented with parameterisability in mind from the beginning.

2.2 ChampSim

I explored different approaches for evaluating predictors. One option was ChampSim, an "open-source [trace-based] simulator" [21], "[designed to have] low [start-up] time, broad applicability, and high [configurability]" [22].

A fellow Part II candidate, whose blind grading number is 2373A, made modifications to allow branch predictors written in BSV and conforming to Toooba's BDP interface to be simulated. These modifications were made before we began our Part II projects, so our projects were in no way collaborative. They generously allowed me to use and modify their version, which I used to get started because it offered a minimal environment for a BDP written in BSV to be compiled and evaluated.

I removed some of their features that are irrelevant to this project and partially refactored the rest. The motivation for this refactoring was to gain an understanding of the code. In hindsight, this was not necessary.

Ultimately, I decided against using ChampSim for the rest of the project because of three key limitations: as implemented, BSV BDPs are not utilised in a superscalar manner, meaning the accuracy and correctness of a BDP for Toooba is not fully investigated; further modifications would be required after I modified Toooba's BDP interface (§ 3.6); and only traces from x86 programs are supported – a different instruction set from RISC-V.

I include this section because ChampSim served as an initial environment for my implementation while I was still gaining familiarity with BSV, Toooba, and BDPs. It also developed my understanding of trace simulation and developed my ability to understand an existing codebase.

2.3 Bluespec SystemVerilog

BSV is an open-source hardware description language (HDL), which Toooba and the parameterisable predictor are written in.

It "uses SystemVerilog's model of interfaces and interface instances" and "types and type system" [23]. BSV consists of interfaces which modules may implement. Modules consist of methods and rules, and may call another module's methods via an interface. Rules fire each cycle if their conditions are met.

[11] This makes Gselect a one-level predictor, despite using a mixture of local and global history.

It is an excellent language for this dissertation because "[BSV's] features – robustness in behavior due to cross-modular rules; strong static checking; and powerful elaboration from orthogonality and *type parameterization* – combine to improve the ability to *reuse code*" [23, emphasis added]. In particular, the parameterisation of hardware modules with types is crucial to this project – as described in section 1.3, a prediction hardware can be parameterised with a Boolean prediction to be a BDP or with an address to be a NAP.

Unfortunately, working in BSV has been difficult in comparison to mainstream HDLs. Its lack of popularity means that online resources are limited; there are 13 questions about it on `stackoverflow.com` and 201 topics on `groups.io` as of 15th April 2025. I used the Cambridge BSV web-tutor [24], which consists of tutorials and exercises, to develop my understanding initially, but was mostly reliant on the reference guides [25].

BSV also gives extremely verbose error messages – a single error that I encountered was 250,000 lines long!

Notwithstanding the difficulties faced, I am glad to have worked in this language.

2.4 Toooba

Toooba is an open-source RISC-V CPU from Bluespec, Inc. that is superscalar and out-of-order. The parameterisable predictor is designed for instantiation as a BDP and a NAP within Toooba.

Toooba is based on MIT's RiscyOO [26], which was built under a framework of composable modular design (CMD)[12] [27]. Notable additions from RiscyOO to Toooba include support for compressed instructions and supervisor and user privilege modes.

Toooba was chosen for this project because it is open-source; written in BSV, which was the perfect HDL for this project as described in the previous section; and is "intended as a high-end application processor" [26], meaning the BDP and NAP predictors are instantiated in a realistic setting, adding credibility to the investigation into their accuracies and performance, benefitting industry and academia.

2.4.1 Front-end

Documentation for Toooba does not exist, so this section contributes a description of the front-end, with emphasis on prediction. This description is useful for understanding BDP and NAP within Toooba and for understanding the bug that I fix in section 3.1. There is documentation for RiscyOO [28], which informs much of this section.

Figure 2.2 shows the pipeline stages Fetch 1, Fetch 2, and Decode. Fetch 1 uses a NAP to choose the next PC to fetch from. Decode also chooses the next PC with a combination of a BDP and a return-address stack (RAS) that takes priority. If the PC predicted by Decode is different than Fetch 1 predicted, it redirects the pipeline (sets PC for next fetch) and trains the NAP. The execution stages can also redirect the pipeline and train both the NAP and BDP.

In addition to NAP, Fetch 1 requests instruction fragments from the instruction cache (using the physical addresses – translated from virtual addresses via the translation look-aside buffer (TLB)) and may also perform memory-mapped input/output operations. Fetch 2 arranges instruction fragments to be decoded. In addition to BDP, Decode combines instruction fragments into instructions where appropriate and decodes them, passing the result to the Rename stage.

[12] More detail in § 2.8.2.

Toooba has one fewer fetch stage than RiscyOO due to caching TLB entries for instruction addresses in a micro-cache that can be accessed without a cycle between requesting and receiving a response. Because of this it recovers from pipeline flushes faster when it already has a translation for the virtual page in the micro-cache.

Misspeculation is dealt with via numbered epochs; a mismatch in expected epoch indicates misspeculation.

The front-end is always active, even during stalls, so it is possible for the same training of predictors to be repeated despite not progressing through the program.

For both BDP and NAP training, there is only one update per cycle.

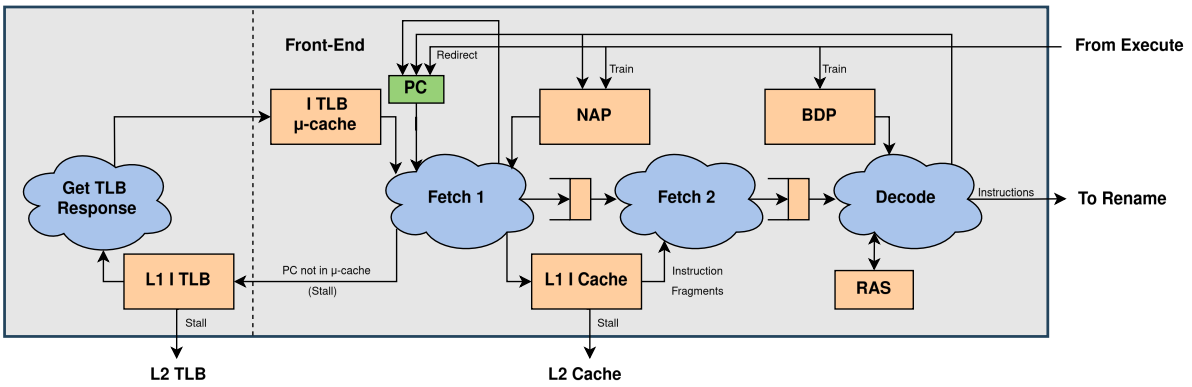

Figure 2.2: Toooba's front-end.
N.B. "I" means "instruction" for TLB and cache.

We will now cover the BDP and NAP interfaces in detail to inform the requirements and design of the parameterisable predictor.

2.4.2 Branch direction prediction interface

Toooba uses the interface shown in listing 2.1 for BDP.

```
1  typedef struct {
2      Bool taken;
3      trainInfoT train;
4  } DirPredResult#(type trainInfoT) deriving(Bits, Eq, FShow);
5
6  interface DirPred#(type trainInfoT);
7      method ActionValue#(DirPredResult#(trainInfoT)) pred;
8  endinterface
9
10 interface DirPredictor#(type trainInfoT);
11     method Action nextPc(Addr nextPc);
12     interface Vector#(SupSize, DirPred#(trainInfoT)) pred;
13     method Action update(Bool taken, trainInfoT train, Bool
           mispred);
14     method Action flush;
15     method Bool flush_done;
16 endinterface
```

Listing 2.1: Toooba's BDP interface.

Because Toooba is superscalar, multiple predictions (`pred`) can be made each cycle. `SupSize` denotes the "superscalar size" or "width" – how many instructions can be in each stage of the pipeline each cycle – and is set to 2; thus zero, one, or two predictions may be made each cycle.

`nextPc` is used to tell the BDP the "base" address to perform predictions from and is the address of the first instruction in the superscalar "batch". NB: `Addr` is the type used for 64-bit addresses. The BDP assumes the second instruction to have an address of `nextPc` + 4, though this is not always the case if the first instruction is compressed or the NAP predicted a taken branch or jump for the first instruction.

The `update` method is used to train the predictor based on whether a branch was actually taken when it was executed. The training information (`trainInfoT`) – information to identify the branch/PT entry and anything else the BDP relies on to update itself – that was returned alongside the prediction for the now-executed branch is passed back to the BDP, after having been through the pipeline alongside the branch.

The methods `flush` and `flush_done` are only used (and given a non-trivial definition) in secure contexts – beyond the scope of this dissertation.

2.4.3 Next-address prediction interface

Toooba uses the interface shown in listing 2.2 for NAP.

```
1 interface NextAddrPred#(numeric type hashSz);
2     method Action put_pc(Addr pc);
3     interface Vector#(SupSizeX2, Maybe#(Addr)) pred;
4     method Action update(Addr pc, Addr brTarget, Bool taken);
5     method Action flush;
6     method Bool flush_done;
7 endinterface
```

Listing 2.2: Toooba's NAP interface.

`put_pc`, `pred`, `update`, `flush`, and `flush_done` serve the same function as the methods and interfaces in the same order of the BDP interface. However, there are some crucial differences for `pred` and `update` described in the following section. NB: `hashSz` is specific to the BTB implementation and is not discussed.

2.4.4 Differences between the branch direction prediction and next-address prediction interfaces

Toooba's NAP interface has several differences from its BDP interface – enumerated below – which had to be considered in the requirements and design of the parameterisable predictor.

1. The maximum number of predictions are made each cycle and they are all called and returned at once rather than individually, so `pred` returns a vector of predictions rather than a vector of interfaces with a single `pred` method.[13]

2. The NAP operates on each instruction fragment rather than each instruction. Naturally, the PC used for each prediction in the batch increments by 2 rather than 4 and there are `SupSize` × 2 predictions (`SupSizeX2`).

[13] My supervisor and I believe that it is a mistake for `pred` to be an `interface` rather than a `method`; it is a language quirk that this does not affect compilation.

3. The NAP can miss and return `Invalid` rather than a `Valid` address, thus the return type for each prediction is a `Maybe#(Addr)`. `Invalid` is used to represent the next address being PC + 2 rather than `Valid`(PC + 2). The ability to miss arises from using tags for PT entries.

   > While we are on the topic of the `Maybe#(Addr)` type, there is also an important distinction between returning `Valid`(PC + 2) and `Invalid`. While these are somewhat equivalent, Toooba interprets `Valid(addr)` as a redirect to `addr`, reducing pipeline throughput by ending the current fetch batch, and `Invalid` as the common case that the next half-word is contiguous to the current half-word.

4. The NAP does not return any training information to be passed back to it.

5. The NAP is not informed via `update` if it correctly predicted `Invalid`.

6. `update` does not have a misprediction (`mispred`) parameter.

7. 64 bits of a possible address and one validity bit is a much longer result type than the single bit used in the global history of a BDP.

2.5 Benchmarks

The primary benchmark used throughout the implementation of BDPs and the parameterisable predictor was CoreMark® [29]. We can establish a change in the clock cycles per instruction (CPI) as a proxy for a change in predictor accuracy when nothing besides the predictor in question changes. This was used as a "sanity check" to guide the implementation, verifying that each step was in the right direction. The magnitude of a CPI change does not mean as much since it will be affected differently between NAP and BDP. Due to the small length of CoreMark® (189 K instructions) it is likely weighted towards the start-up phase of a predictor's life rather than the most accurate steady-state; therefore, it does not count as a full evaluation.

For the evaluation proper, I used a subset of the MiBench suite [30] of embedded benchmarks, prepared by Jonathan Woodruff. The subset consists of 11 programs across automotive and industrial control, consumer devices, networking, security, and telecommunications workloads. Instruction counts range from 106 K – 3,733 K, with a median of 2,370 K and a total of 22 M.

2.6 Starting point

The starting point for this project consists of Toooba (§ 2.4), the organisation of the Gselect BDP (§ 2.1), the CoreMark® and MiBench benchmarks (§ 2.5), and the modified version of ChampSim (§ 2.2).

I had completed roughly one third of the Cambridge BSV web-tutor (§ 2.3) before beginning the project.

2.7 Requirements analysis

I use the MoSCoW (must/should/could/won't have) framework to prioritise the requirements of this project because it emphasies key requirements and prevents scope creep. The success criteria (§ 1.4) inform the requirements and the requirements can be seen as an extension of them. Each requirement is traceable and worthy of documentation; actionable; and either testable or measurable, as recommended in [31]. Testability may be a yes/no judgement and measureability is from the benchmarks described in section 2.5.

The parameterisable predictor must

M1 be instantiated by itself as a BDP;

M2 be instantiated by itself as a NAP;

M3 use an organisation originating directly from the ideas of Gselect; and

M4 respect the interface differences between BDP and NAP in Toooba (§ 2.4.4) or rework them.

The parameterisable predictor should

S1 achieve a comparable accuracy and performance as Toooba's Gselect BDP; and

S2 make minimal modifications to Toooba.

The parameterisable predictor could

C1 achieve comparable or better accuracy and performance to Toooba's BTB for NAP.

The parameterisable predictor won't

W1 be instantiated for modes of prediction besides BDP and NAP.

As Gselect is an organisation for BDPs, the accuracy and performance should closely match Toooba's own Gselect implementation for the same predictor size. An exploratory approach is more appropriate for the NAP instantiation, putting it in the "could" category, although it would be possible for it to outperform Toooba's BTB at a similar size.

2.8 Engineering techniques

In order to meet the success criteria and requirements, I took a professional approach to this project, with the following engineering techniques and an iterative methodology for project management.

2.8.1 Best practices

- I began using Linux which was not easy but was beneficial to the project.
- I used Git and GitHub for version control and back-up. I also used OneDrive for back-ups (a pain for Linux!).
- I used ChatGPT to generate a script to ignore previously seen warnings when building Toooba so that I would not glance over warnings that I introduce. I do not claim credit for the script and it is clearly marked within the submitted repository. ChatGPT was an appropriate tool here because of its speed and the fact that the script does not tackle the intrinsic complexity of the project. The script did contain several errors which I fixed but it was still faster than writing it myself.
- I also used BSV scheduling attributes that can turn warnings into errors, e.g., `fire_when_enabled` specifies that a rule that must fire in any cycle where its explicit conditions are met (it cannot be implicitly blocked by conditions of methods it calls).
- When modifying Toooba to support custom BDP and NAP interfaces, I introduced compilation flags to control which interfaces are used, avoiding version control issues when evaluating.

2.8.2 Composable modular design

RiscyOO – and by extension, Toooba – were developed under a framework of CMD, described in [27]:

> “In CMD, a design is a collection of modules, which interact with each other according to a set of atomic rules such that,
>
> - The interface methods of modules provide combinational access and perform atomic updates to the state elements inside the module;
> - Every interface method is guarded, i.e., it cannot be applied unless it is ready; and
> - Modules are composed together by rules or “transactions” which call interface methods of different modules. A rule either successfully updates the states of all the called modules or it does nothing.”

I have used CMD for the development of the parameterisable predictor, which gives the benefit of “composability when selected modules are refined selectively” – the inner workings of modules are not seen other than through their interfaces. CMD was easy to achieve from working in BSV which “provides type checking and enforces a guarded interface protocol and rule atomicity”. The seminal paper [27] also mentions that “CMD designs are often highly parameterized” which my project’s designs certainly are!

2.8.3 Project management

The first step was to analyse the requirements (§ 2.7); I found them to be clear and unchanging, making project management smooth.

Because I lacked familiarity with BSV and Toooba, I initially used a process of iterative development to make increasingly sophisticated BDPs, working up to Gselect.

After this initial development phase, I analysed the differences between BDP and NAP so that I could design alternate interfaces for them and design the parameterisable Gselect predictor.

I then used emergent design (§ 3.9) to develop the parameterisable predictor out of two mostly similar source files for BDP and NAP, refactoring the commonalities out into the parameterisable predictor. The predictor was instantiated for both BDP and NAP.

Testing was performed throughout and took three forms:

- Being able to run programs such as CoreMark® demonstrates that the predictors are working and the results can be used to "sanity check" predictor accuracy. A subsection of the MiBench suite is used for the full evaluation.
- I ensured that modified Toooba still passed the suite of ISA tests, although unmodified Toooba was already failing one test for which I delivered a bug-fix (§ 3.1).
- I created unit tests for `ValueWithConfidence` and `TRegFile`.

Chapter 3

Implementation

After describing my fix for a bug causing Toooba to fail an ISA test (§ 3.1) and giving a repository overview (§ 3.2), we work towards a parameterisable predictor following the organisation of Gselect that can be instantiated for BDP and NAP within Toooba.

First, I describe my implementation of BDPs for Toooba's interface, working up to Gselect (§ 3.3). I then describe my `ValueWithConfidence` datatype (§ 3.4) which my Gselect implementation uses. Next, we cover the design process for dealing with the interface differences between BDP and NAP (§ 3.5) and modify Toooba and my Gselect implementation (§ 3.6), which requires the `TRegFile` module (§ 3.7). We then traverse the path to using a modified copy of the same source code for NAP (§ 3.8). This allows for the commonalities to be refactored out of both designs via emergent design (§ 3.9) and the parameterisable predictor is born (§ 3.10); it is instantiated for both modes of prediction. Finally, after being unsatisfied with the results of the evaluation and investigating the cause of this poor performance in chapter 4, I perform one last round of iterative development to create the parameterisable hash-tagged Gselect predictor (§ 3.11) in order to meet the final unmet requirement (C1), giving a satisfying and successful end to this project.

This project is entirely simulation based, using Bluesim, which simulates BSV code.

3.1 A security bug-fix for Toooba

Toooba, without any modifications and cloned from the master branch, was failing an ISA test. The test is `RV64MI-access` and involves jumping to an illegal virtual address [32]. Being able to jump to illegal addresses violates the Sv39 virtual memory system [33, section 12.4]. This bug also affects CHERI-RISC-V in their fork of Toooba.

The reason for the failure was that the TLB micro-cache (described in § 2.4.1) was reusing recently translated physical page numbers without checking if the current virtual address is valid under Sv39. This meant that illegal addresses could be executed without exception.

I fixed the bug by indexing the micro-cache with full addresses rather than virtual page number only.

- If an address is valid, we consider a match if the virtual page numbers match, allowing a reuse of the cached physical page number as before.
- If an address is invalid, we consider a match only if the full virtual address matches. When there is no match, a TLB response is requested and cached in the next cycle; it will have an error code. We then use this error response and delete it from the micro-cache so that normal operation may resume.

My pull request for this fix on the GitHub repository for Toooba has been merged into the master branch; I am very proud to have contributed a bug-fix with security implications to an open-source CPU. The fix can be seen in full in appendix A.

3.2 Repository overview

Below is a representation of the file tree for my repository. Not all files are shown.

```
/
├── evaluation
│   ├── executables/ ..... Saved executables
│   ├── logs/ ..... Saved logs
│   └── evaluation.py ..... Script to interpret logs
├── parameterisable_predictors
│   ├── GSelect/ ..... The parameterisable Gselect predictor
│   └── PTGS/ ..... The parameterisable hash-tagged Gselect predictor
├── branch_predictors ..... BDPs
│   ├── AlwaysTaken/
│   ├── SaturationCounter/ ..... Local one-level
│   ├── GSelectBase/ ..... Gselect predictor for the original BDP interface
│   ├── GSelect/ ..... Gselect predictor for the modified BDP interface
│   ├── ParamGSelectBdp/ ..... Instantiation of the parameterisable predictor
│   └── PTGS_BDP/ . Instantiation of the parameterisable hash-tagged Gselect predictor
├── next_address_predictors
│   ├── GSelectBtb/
│   ├── ParamGSelectNap/ ..... Instantiation of the parameterisable Gselect predictor
│   └── PTGS_NAP/ . Instantiation of the parameterisable hash-tagged Gselect predictor
├── utils ..... Utilities
│   ├── ValueWithConfidence.bsv
│   └── TRegFile.bsv
├── tests ..... Unit tests
│   ├── Makefile
│   └── ...
├── TooobaWrapper
│   ├── Toooba/ ..... Submodule
│   ├── try_coremark.sh ..... Script to compile Toooba and run a benchmark
│   └── filter_known_warnings.py ..... ChatGPT generated script
└── ChampSimWrapper
    ├── ChampSim ..... Submodule
    ├── try_champsim_bsv_branch.sh
    ├── config.json ..... Configuration file
    └── traces/
```

I split the predictors into BDPs, NAPs, and parameterisable predictors. Each predictor has its own directory in case it consists of multiple files, though this did not occur. Some BDPs and NAPs instantiate parameterisable predictors as "inner" predictors.

This structure supports modular experimentation with predictors, allowing results to be easily collected over benchmarks (contained within Toooba).

The utilities are separate because they are used in all predictor categories. There is a separation of concerns for utilities and their tests.

I put submodules in wrappers to organise supplementary scripts around them. I modified both Toooba and ChampSim but the majority of the code is not my own. I added directories for my predictors to Toooba's `Makefiles`.

3.3 From the dumbest predictor to Gselect

Starting in the modified version of ChampSim (§ 2.2) but then moving onto Toooba instead, I began implementing BDPs of increasing complexity as I got to grips with BSV.

3.3.1 Always taken

`AlwaysTaken.bsv`

The simplest BDP organisation is to always predict branches to be taken. This predictor does not need to update its state, so its `update` method is defined to be `noAction`. The interface requires some training information to be returned through `pred[i].pred` and passed back in `update`, so `Bit#(0)` (an empty type) is used for `trainInfoT`.

3.3.2 Local one-level

A.k.a. bimodal

`SaturationCounter.bsv`

I next made a one-level BDP that uses local history only. As shown in figure 3.1, only lower bits from the PC are used to index a PT of two-bit saturating counters, i.e., predictions for a branch are only dependent on the past behaviour of that branch (or branches that alias to it if there is contension in the PT). The bits used exclude the lowest two bits since a memory word will usually only contain a single branch.[14] The address for the second prediction in a batch is assumed to be the address of the next word, though this may not actually be the case due to the presence of compressed instructions. The saturating counters in the PT are two-bit unsigned integers.

The training information consists of the PC bits used as the PT index and the counter for that entry.

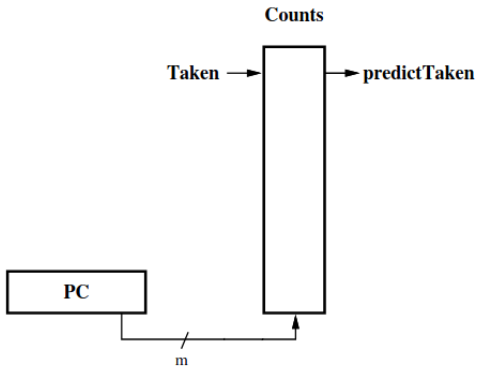

Figure 3.1: A local one-level BDP, using m bits from the PC.
Image credit: edited version of figure from Scott McFarling [18].

[14] The lowest two bits index bytes within a memory word.

3.3.3 Gselect

GSelectBase.bsv

Gselect is similar to a local one-level predictor (§ 3.3.2) but uses global history as well as bits from the PC to index the PT. A full description of Gselect can be found in section 2.1.

Using global history results from previous predictions in the same cycle presented great difficulty but was solved by retaining predictions in a cycle in UnsafeRWires – RWires "allow data transfer between methods and rules without the cycle latency of a register" [25] and "unsafe" means that wires can be written to multiple times in the same rule/method.[15]

I replaced the two-bit saturating counters with remembered Bools each with a single bit of confidence – ValueWithConfidence, described in the next section.

The update method gets a fresh copy of the counter from the PT before updating it, unlike for the local predictor (§ 3.3.2) which uses an outdated copy via the training information.

On misprediction, the global history is reverted to what it was before the prediction was made, along with the actual result.

The training information consists of the PT index and the GR, which is actually redundant as it is included in the index.

3.4 ValueWithConfidence

ValueWithConfidence.bsv

I implemented the datatype ValueWithConfidence – parameterisable in the type of the value remembered and the number of bits of confidence. I used an instantiation of it in my Gselect predictor with type Bool and one bit of confidence, which is isomorphic to a two-bit saturating counter, as described in section 1.3.

ValueWithConfidence is implemented as a struct of the remembered value (of some given type) and an unsigned integer (of some given number of bits). I have defined a function updateValueWithConfidence which takes a ValueWithConfidence and a value of its underlying type. If the new value is the same as the remembered value, the confidence increases by 1, up to the maximum confidence; otherwise, the confidence decreases and if it would go below zero, the new value is remembered instead with a confidence of zero.

I wrote unit tests for ValueWithConfidence, instantiating it for both a BDP and with two bits of confidence for an enumerated type of my favourite musicians. Both tests involved incrementing confidence (by giving it the same value again) once it was at the maximum confidence and remembering a new value once confidence would drop below zero. It passes the tests.

[15] My Gselect implementation does not actually do this and I believe it to be a compiler bug that I had to use UnsafeRWire instead of RWire. This issue is also the source of the 250,000-line error message mentioned in § 2.3.

3.5 Designing for the interface differences

This section makes up a large part of the design phase of my project management methodology (§ 2.8.3). It is informed by BDP and NAP (§ 1.2.1, § 1.2.2); Toooba's front-end (§ 2.4.1), its prediction interfaces (§ 2.4.2, § 2.4.3) and **their differences (§ 2.4.4 – repeated partially)**; and the organisation and implementation of Gselect (§ 2.1, § 3.3.3).

I decided that, due to the numerous differences, I would design modified interfaces for both BDP and NAP in Toooba. The differences are minimal to adhere to requirement S2.

	Interface difference for NAP from BDP	Design constraint/decision
1.	The maximum number of predictions are made each cycle and they are all called and returned at once rather than individually, so `pred` returns a vector of predictions rather than a vector of interfaces with a single `pred` method.	Alternate NAP interface will follow BDP interface; the NAP caller can call all `pred`s together
2.	The NAP operates on each instruction fragment rather than each instruction.	The parameterisable predictor should have a parameter for the number of predictions that can be made each cycle and a parameter for which bits of the PC to use.
3.	The NAP can miss and return `Invalid` rather than a `Valid` address, thus the return type for each prediction is a `Maybe#(Addr)`.	∗
4.	The NAP does not return any training information to be passed back to it.	∗
5.	The NAP is not informed via `update` if it correctly predicted `Invalid`.	∗
6.	`update` does not have a misprediction (`mispred`) parameter.	Alternate BDP interface will not have the `mispred` parameter; it is easy to recalculate.
7.	64 bits of a possible address and one validity bit is a much longer result type than the single bit used in the global history of a BDP.	The parameterisable predictor will have a mechanism for condensing results (of predictions or explicit updates) into "history items".

∗ Rather than storing `Invalid`, Toooba's original BTB may miss (or be unconfident in its result) and return `Invalid`. In this case – when it was correct to predict `Invalid` because the next instruction fragment was at the next contiguous half-word – the BTB is not informed of its correctness via an update.

The current scheme contradicts the organisation of Gselect which does not support missing – instead, PT entries alias. I decided to stay closer to Gselect rather than creating

another predictor organisation.[16] This means that `Invalid` must be stored (with a bit of confidence) in the PT.

To solve the problem of not getting all the required updates, I decided that old predictions may be assumed to be correct. I decided that rather than training information (which only the BDP interface used), the predictors would pass "prediction tokens" which are used to identify the internally-stored training information that is relevant for this update, assuming this old prediction to be correct. The generated prediction tokens are eventually re-used so if we have not heard back for some training information that is about to be overwritten, we use it for an update. This creates a need for multiple updates per cycle, since there are multiple predictions per cycle that may overwrite old training information as well as the explicit `update` method. Training information should be deleted after an update uses it.

Moving information through the pipeline – meaning more pipeline registers – is a big power cost in computer architecture [34, chapter 7.7]; therefore, a fixed size prediction token is more efficient for arbitrarily complex training information because the training information remains as internal state of the predictor.

My design decisions concluded that

- Toooba would be modified to use prediction tokens rather than training information for BDP;
- Toooba would be modified to pass prediction tokens for NAP;
- multiple updates may occur each cycle, up to `SupSize`+1 for a BDP or `SupSizeX2`+1 for a NAP;
- each prediction would be an interface with a method rather than a single method returning a vector of predictions;
- the parameterisable predictor would have a parameter for the maximum number of predictions each cycle;
- the parameterisable predictor would have a parameter for which bits of the PC to use;
- neither interface would have a misprediction parameter in their `update` method; and
- the parameterisable predictor would have a mechanism for condensing past predictions and results into history items.

3.6 Dealing with the interface differences

From these design decisions, I had to modify Toooba and my Gselect implementation. I ensured that any modifications to Toooba could be disabled with compilation flags to avoid version control issues during evaluation.

[16] After the evaluation, I created the parameterisable hash-tagged Gselect predictor, which is a custom organisation (§ 3.11).

3.6.1 Alternate BDP interface

`BrPred.bsv` – inside Toooba

Listing 3.1 shows my interface for BDP. It passes prediction tokens rather than training information and has different arguments to `update`.

```
1  typedef struct {
2      Bool taken;
3      dirPredTokenT token;
4  } DirPredResult#(type dirPredTokenT) deriving(Bits, Eq, FShow
       );
5
6  interface DirPred#(type dirPredTokenT);
7      method ActionValue#(DirPredResult#(dirPredTokenT)) pred;
8  endinterface
9
10 interface DirPredictor#(type dirPredTokenT);
11     method Action nextPc(Addr nextPc);
12     interface Vector#(SupSize, DirPred#(dirPredTokenT)) pred;
13     method Action update(dirPredTokenT token, Bool taken);
14     method Action flush;
15     method Bool flush_done;
16 endinterface
```

Listing 3.1: Alternate BDP interface.

3.6.2 Alternate NAP interface

`BtbIfc.bsv` – new file inside Toooba

Listing 3.2 shows my interface for NAP. It follows the same structure as my BDP interface. `put_pc` is a differently named method that is otherwise identical to `nextPc`; it uses a different name for consistency with the original NAP interface.

```
1  typedef struct {
2      Maybe#(Addr) maybeAddr;
3      napTokenT token;
4  } NapPredResult#(type napTokenT) deriving(Bits, Eq, FShow);
5
6  interface NapPred#(type napTokenT);
7      method ActionValue#(NapPredResult#(napTokenT)) pred;
8  endinterface
9
10 interface NextAddrPred#(type napTokenT);
11     method Action put_pc(Addr pc);
12     interface Vector#(SupSizeX2, NapPred#(napTokenT)) pred;
13     method Action update(napTokenT token, Maybe#(Addr)
           brTarget);
14     method Action flush;
15     method Bool flush_done;
```

```
endinterface
```

Listing 3.2: Alternate NAP interface.

3.6.3 Modifying Toooba

Listing 3.3 shows one case of how I modified Toooba to support my interfaces. Notice that compiler directives for conditional compilation are used to determine which interfaces to compile with, based on the `ALTERNATE_IFC_BDP` and `ALTERNATE_IFC_NAP` compilation flags. Most fields of the `struct` are omitted.

```
let out = FromFetchStage{
    pc: pc,
    //...
    `ifndef ALTERNATE_IFC_BDP
    dpTrain: dir_pred.train,
    `else
    dpToken: dir_pred.token,
    `endif
    //...
    `ifdef ALTERNATE_IFC_NAP
    , napToken: in.napToken,
    hiNapToken: in.hiNapToken
    `endif
};
```

Listing 3.3: The output of Toooba's front-end (NB: `FromDecode` would be a better name for the struct).

3.6.4 Modifying my Gselect

`GSelect.bsv`

This section describes how I adapted my implementation of the Gselect BDP (§ 3.3.3) to work for the alternate BDP interface.

- Rules are used to apply the up-to `SupSize` + 1 updates each cycle. The `update` method signals to perform an update in one of these rules.
- In the update rules, the predictor accesses training information via a register file indexed by the supplied prediction tokens.
- In the update rules, the removed `mispred` (misprediction) argument is computed via comparing the predicted direction – obtained via the training information – to `taken`.
- Training information is signalled to be deleted after an update concerning it is performed. The deletion doesn't take place if we instead store new information when predicting.

The multiple updates were made possible by using my TRegFile module, described in the next section, for the PT and stored training information. NB: these extra updates don't actually occur for BDP since it receives updates on every prediction and this is faster than the same prediction token being reused.

3.7 TRegFile: Transactional register file

TRegFile.bsv

Register files in BSV may only be written to once per cycle, making multiple updates per cycle – as the parameterisable predictor does – difficult. To overcome this limitation, I developed the transactional register file, or TRegFile.

The TRegFile, as shown in figure 3.2, supports multiple writes per cycle, takes intitial values, and can be reset via clear to those values. It uses the highest-indexed write port if there are conflicting writes for an entry. There are also read ports for the sake of symmetry with writing.

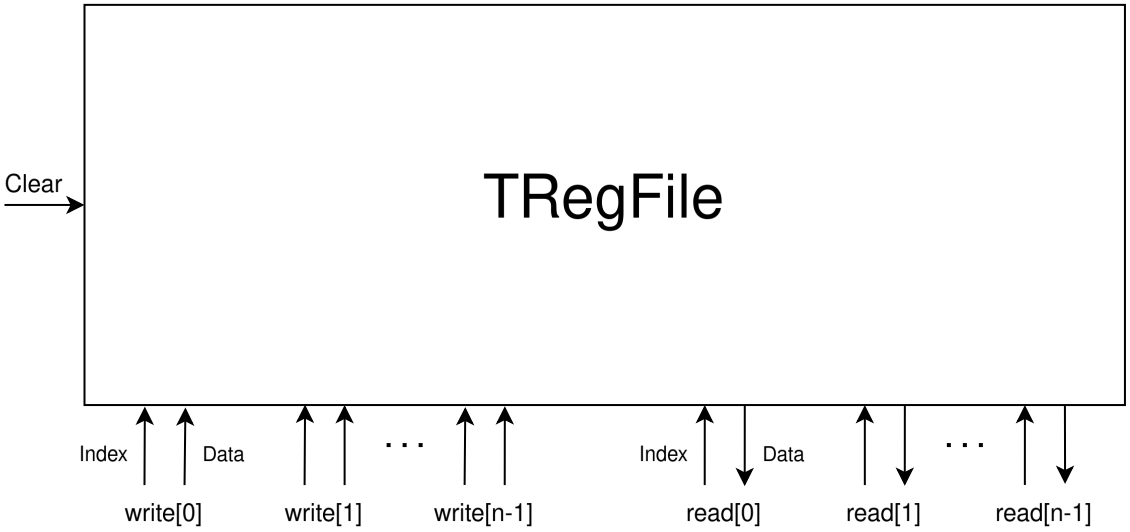

Figure 3.2: The TRegFile interface.

I wrote unit tests for TRegFile: I tested it used as a normal register file; with multiple reads and writes per cycle; and using its clear method (also tested in the same cycle as a write). I found a bug in the order of write ports due to testing and then fixed it. All tests pass.

In synthesis, TRegFile would not map well to real hardware found in, e.g., FPGAs; however, this project only concerns simulation. A more realistic module for the same interface could use a write-queue to a register file.

While working on this module, I conversed with one of the creators of BSV – Dr. Rishiyur Nikhil – via forum posts.

3.8 Gselect next-address predictor

`GSelectBtb.bsv`

Copy, paste, tweak!

The next task was to use a modified copy of my Gselect alternate-interface BDP for NAP. Due to the alternate interfaces, we have less differences to worry about, however some still prevail. These are the intrinsic complexity of this dissertation – the differences between BDP and NAP. These remaining differences are listed below.

- There are up to `SupSizeX2` predictions per cycle rather than `SupSize`.
- There are up to `SupSizeX2` + 1 updates per cycle rather than `SupSize` + 1.
- Different bits of the PC should be used in the PT index. The bit range for NAP starts one bit lower (only one least-significant bit is excluded).
- The type for `ValueWithConfidence` is `Maybe#(Addr)` instead of `Bool`.
- The "default prediction" – the initial remembered value in each PT entry – is `Invalid` instead of `False`.
- Only the validity bit out of the `Maybe#(Addr)` result type is used as a global history item, meaning the global history is whether or not each instruction was a jump/branch or not. Another option would have been to hash the full results together but this would be noisy and closer to Gshare than Gselect.

3.9 Emergent design

Emergent design involves "refactoring out" commonalities between two units of code [35]. I used it to obtain the parameterisable predictor from the Gselect BDP and Gselect NAP. I performed all tests before and after this refactoring, as advised in [35].

3.10 One predictor to rule them all: the parameterisable predictor

`ParamGSelect.bsv`

One predictor to rule them all, one predictor to build them, One predictor to predict them all and in the darkness guide them.

The parameterisable predictor was created using emergent design from my implementations of the Gselect BDP and the Gselect NAP. Any type definitions used within were replaced with type parameters. The parameterisable predictor is instantiated as an "inner predictor" for a module that acts as a BDP wrapper and a module that acts as a NAP wrapper.

The complete interface is shown in listing 3.4.

```
1  interface Predict#(type tokenT, type resultT);
2      method ActionValue#(PredictResult#(tokenT, resultT))
           predict;
3  endinterface
4
5  interface ParamGSelect#(
6      // type variables omitted but shown in mkParamGSelect
7  );
8      method Action nextPc(Addr nextPc);
9      interface Vector#(numPreds, Predict#(tokenT, resultT))
           predict;
10     method Action update(tokenT token, resultT actual);
11     method Action flush;
12     method Bool flush_done;
13 endinterface
14
15 module mkParamGSelect#(
16     Addr pcBitMask,  // Must have numPcBits set bits.
17     resultT defaultPrediction,
18     function globalHistoryItemT makeGlobalHistoryItem(resultT
           result)
19 ) (ParamGSelect#(resultT, tokenT, numPreds, numPcBits,
       numGlobalHistoryItems, globalHistoryItemT,
       numConfidenceBits))
20     provisos(
21         // omitted
22     );
```

Listing 3.4: The interfaces and module signature for the parameterisable predictor.

The bits to use from the PC when generating PT indexes are controlled via a parameter, `numPcBits`, that says how many bits to use and a provided bit mask. Naturally, the number of set bits in the mask must match the parameter.

The function argument `makeGlobalHistoryItem` provides a function that takes a result and turns it into an item of global history which can be used in future to index the PT.

The size of each PT index, call it indexLen, is `numPcBits`+`numGlobalHistoryItems`×`SizeOf#(globalHistoryItemT)`, where `numGlobalHistoryItems` is the number of items of global history to consider for each prediction and `globalHistoryItemT` is the type of each item. The number of entries in the PT is 2^{indexLen}. Each entry is of size `SizeOf#(resultT)` + 1 because of the confidence bit. Therefore, the predictor size in bits is

$$(\texttt{SizeOf\#(resultT)} + 1) \times 2^{\texttt{numPcBits}+\texttt{numGlobalHistoryItems}\times\texttt{SizeOf\#(globalHistoryItemT)}}.$$

We do not consider the size of the internally held training information, similarly to how Toooba's current BDP sizes do not account for "holding" training information by passing it through the pipeline.

3.10.1 Instantiations

`ParamGSelectBdp.bsv`, `ParamGSelectNap.bsv`

The table below shows the parameterisation of the parameterisable predictor as an inner predictor for both BDP and NAP.

Parameter/argument	BDP	NAP
resultT	Bool	Maybe#(Addr)
tokenT	UInt#(8)	UInt#(8)
numPreds	SupSize	SupSizeX2
numPcBits	4	3
numGlobalHistoryItems	8	7
globalHistoryItemT	Bool	Bool
numConfidenceBits	1	1
pcBitMask	'b111100	'b1110
defaultPrediction	False	Invalid
makeGlobalHistoryItem	id	isValid

This gives a 8 Kb BDP and a 66 Kb NAP. The parameters used for BDP mimic those used in Toooba's implementation of Gselect (`GSelectPred.bsv`), giving a predictor of the same size. For NAP, I chose parameters that made a predictor of a size similar to Toooba's BTB – its size is 73.5 Kb – and used a similar ratio of PC bits to global history bits as for BDP.

The function `id` does nothing and is used because BDP uses the same global history type as its result type. `isValid` extracts the validity bit from a `Maybe#(Addr)` (whether it is a jump/branch).

These instantiations were evaluated in chapter 4 and found to have poor performance and accuracy for NAP, which lead me to create another parameterisable predictor with a custom organisation, which we cover in the next section.

3.11 Hash-tagged Gselect

`PTGS.bsv`, `PTGS_BDP.bsv`, `PTGS_NAP.bsv` – originally stood for partially-tagged Gselect

It's actually a precious gift to be dissatisfied, because that keeps you moving forward

—Porter Robinson

Unsatisfied with the results of the evaluation in chapter 4 but with days until the submission date, I decided upon one more iterative development cycle to create the parameterisable hash-tagged Gselect (HTGSEL) predictor – a version of Gselect with hashed tags for the PT entries. This solves the problem of the NAP storing `Invalid` most of the time, which effectively says "this isn't a branch/jump". I also suggest some future work in this section, adding to the work suggested in the conclusions in chapter 5.

Hashed tags are used within Toooba's BTB but my review of the literature has not found any predictors that utilise both global history and hashed tags and their combination may be a novel approach.

As shown in figure 3.3, PT entries in HTGSEL have a parameterisable (and therefore optional) tag to determine if an entry is a hit or a miss for a given PC. The tag comes from the address bits not selected for the index, hashed via XOR to the tag size.

Entries also have a validity bit in the form of a `Maybe` type so that entries can be "deleted" when `Invalid` should be predicted. The NAP instantiation is parameterised with address bits instead of a `Maybe#(Addr)` because the `Maybe` is added by the predictor. Entries may be replaced when their confidence is zero or they are invalid.

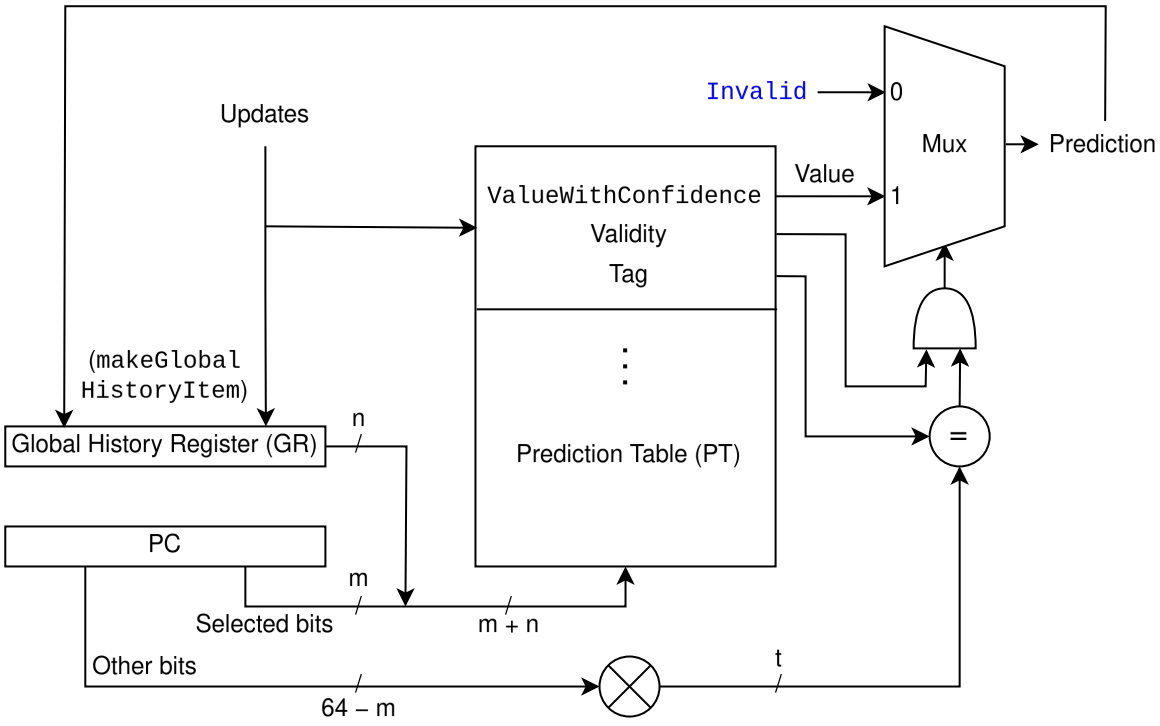

Figure 3.3: The parameterisable HTGSEL predictor, with n bits of global history, m selected bits from the PC, and t-bit tags.

Unfortunately, the enforced validity bit means that BDP entries have this bit unnecessarily, increasing predictor size by 50%. NB: zero bits are used for the tags in the BDP instantiation, but this validity bit remains. For comparison, Toooba's BTB acts as if it missed when it finds a PT entry with zero confidence but this isn't a viable alternative because it would affect BDP behaviour. Perhaps we have found the limit to parameterisable predictors, stemming from the intrinsic differences between the modes of prediction, or maybe there is room for optimism and other organisations should be tried. Perhaps it would be possible for the predictor to identify when the tag size is zero bits, as it is for BDP, and not include the validity bit, but this would require alternate control flow in the update logic, going against the spirit of reusing source code.

An optimisation for the NAP instantiation is storing partial addresses in the inner predictor; the surrounding NAP forms the predicted address by assuming the upper bits are not changed by the branch/jump and the least-significant bit is always zero (instruction fragments are always half-word aligned). Reducing the address bits in each entry allows for more entries at a similar size (and also accounts for the tag bits). Perhaps room could also be made for more confidence bits in the entries, which `ValueWithConfidence` supports and the predictor has been parameterised for, but this is not yet explored.

Finally, I tweaked the parameters for NAP, using 7 address bits, 4 bits for the GR, a 24-bit result type, and a 10-bit tag size. This made a 73.7 Kb PT which is an almost identical size as Toooba's BTB – 73.5 Kb.

Chapter 4

Evaluation

Firstly, we inspect how important metrics for evaluation are collected (§ 4.1). This is used to gather results over benchmarks for instantiations of the parameterisable Gselect predictor for BDP and NAP (§ 4.2), feeding into an evaluation of the entire project against the success criteria and requirements (§ 4.3); the project is found to be successful, meeting the whole success criteria and all of the "must" and "should" requirements. However, we see that the NAP instantiation is rather weak and perform an investigation into it (§ 4.4). This information is taken advantage of with one more iterative development cycle for the parameterisable HTGSEL predictor (§ 3.11), which we then evaluate the BDP and NAP instantiations of (§ 4.5), finding them to meet the "could" requirement and making the project a total success!

4.1 Predictor monitoring

To allow for evaluation, I added several logging points, with the `$display()` command in BSV. These are

- when the NAP is trained due to the decode stage (which involves the BDP) disagreeing with one of its predictions;
- when the BDP is trained from execute due to mispredicting; and
- every 1,000 cycles the `TRegFile`s output how many of their entries differ from their initial contents.

The NAP and BDP trainings allow us to find a misprediction rate, which we express in mispredictions per thousand instructions (MPKI) – a standard unit [12]. The NAP rate is compared to the decode stage's more accurate, yet still speculative execution path; NAP trainings from the execute stage are not considered because they may double count in the case where NAP was correct but BDP was incorrect. The `TRegFile` contents monitoring is for investigating how many entries in the NAP PT are "Invalid".

4.2 Performance and accuracies

For each of the 11 benchmarks, I found the

- clock cycles per instruction executed (CPI) – figure 4.1;
- BDP misprediction rate – figure 4.2; and
- NAP misprediction rate, using decode as the ground truth – figure 4.3

across the configurations (with text colour acting as a legend for the figures)

- **unmodified Tooba, using its implementation of the Gselect BDP and its BTB**;
- **Tooba with my BDP and its BTB**;
- **Tooba with its Gselect BDP and my NAP**; and
- **Tooba with my BDP and my NAP**.

"My BDP" and "my NAP" refer to instantiations of the parameterisable Gselect predictor. The PT sizes are equivalent or smaller than Tooba's Gselect and BTB, as described in section 3.10.1.

I also report means across benchmarks for each configuration, weighted by the instruction counts of the benchmarks and with error bars showing the weighted standard deviations using the formula

$$\sigma_{\mathrm{w}} = \sqrt{\sum_{i=1}^{N} w_i (x_i - \mu_{\mathrm{w}})^2}, \text{ where } w \text{ are normalised weights.}$$

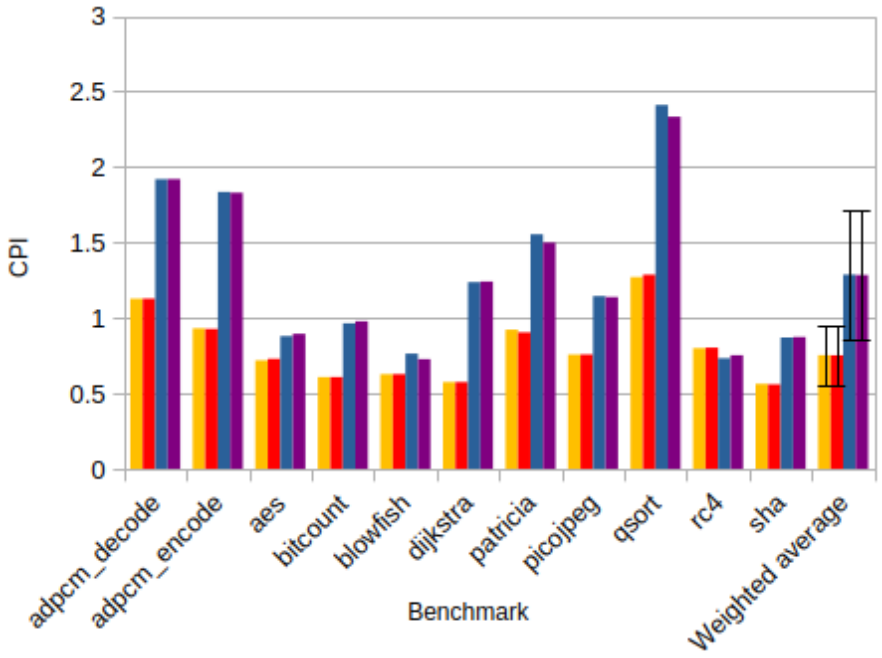

Figure 4.1: CPI for each benchmark across all configurations

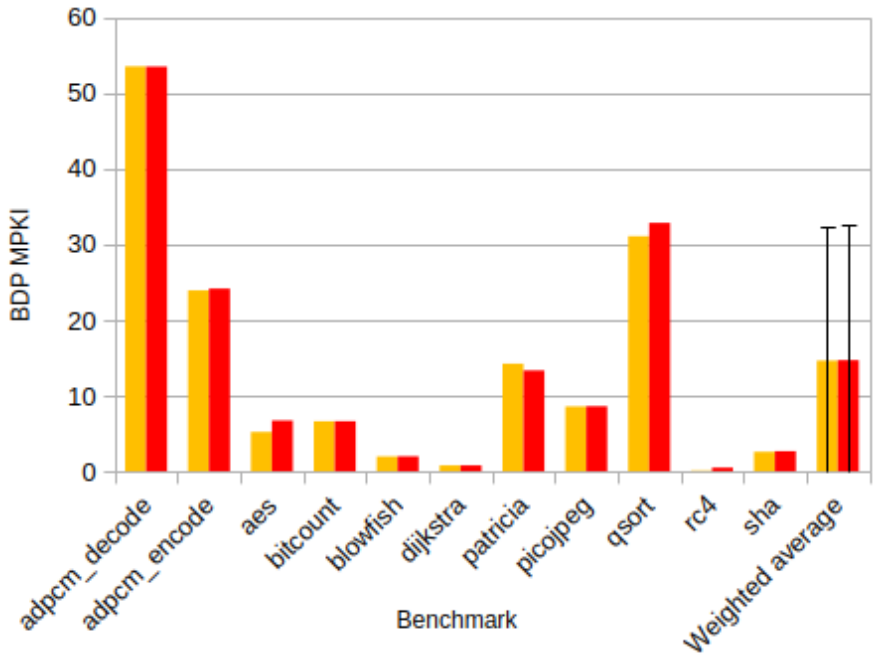

Figure 4.2: BDP MPKI for each benchmark without and with my BDP.
My BDP has near-identical accuracy to Toooba's Gselect BDP.

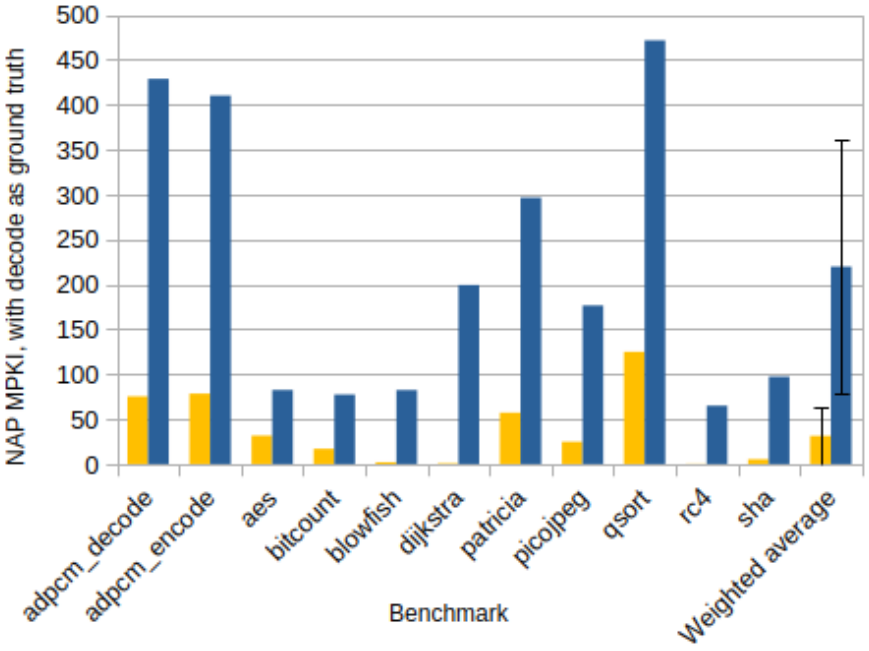

Figure 4.3: NAP MPKI for each benchmark without and with my NAP.
My NAP is significantly worse than Toooba's BTB.

The CPIs in figure 4.1 supplement the above results, showing that the significantly worse NAP accuracy has a big effect on performance while the near-identical BDP accuracy has very little effect on performance.

We now analyse these results in the context of the success criteria and project requirements in the following section.

4.3 Success criteria and project requirements

In this section, we perform an assessment of this project and results against the success criteria (§ 1.4) and project requirements (§ 2.7).

- I have implemented a Gselect predictor in BSV that is parameterisable in the data type it predicts and the global history it keeps, meeting success criteria 1 and requirement M3.
- I have instantiated the parameterisable predictor for BDP within Toooba and figures 4.1 and 4.2 show that it has accuracy and performance similar to Toooba's Gselect BDP; success criteria 2 and requirements M1 and S1 have been met.
- I have instantiated the parameterisable predictor for NAP within Toooba and figures 4.1 and 4.3 demonstrate an investigation into its accuracy and performance, meeting success criteria 3 and requirement M2; however, the accuracy and performance are much worse than Toooba's BTB so requirement C1 has not been met.
- Requirement M4 is met via the alternate interfaces (§ 3.6) which required modifications to Toooba but they were minimal and did not affect control flow, meeting requirement S2.
- The parameterisable predictor was not instantiated for prediction modes besides BDP and NAP, meaning requirement W1 was not met as planned.

The entire success criteria and the "must" and "should" project requirements were met, making this project a success! I also contribute a description of Toooba's front-end and fix a bug within, which I take great pride in. Notably, I did take a different approach to this project than originally planned, using the simpler Gselect predictor organisation instead of the state-of-the-art TAGE, as otherwise I would have had less time for the interesting parts of the project.

We will now investigate why the accuracy of the NAP instantiation of the parameterisable Gselect predictor is so poor.

4.4 Investigation into next-address prediction instantiation

Or, "why is my NAP so bad?"

The Gselect organisation uses a PT with no concept of missing; however, the common case for a BTB is to miss as most instructions aren't jumps/branches! Figure 4.4 displays this well – in each benchmark the PT was storing "this isn't a jump/branch" in over 90% of its entries at the halfway point.[17] Essentially, this unit of hardware specialised for branch/jump instructions is being bogged-down by the common case of instructions that do not modify the PC. The configuration used was Toooba's Gselect BDP with my NAP instantation of the parameterisable predictor. The PT is effectively less than 10% of the size it could be, leading to the dramatic reduction in performance and accuracy found in figures 4.1 and 4.3.

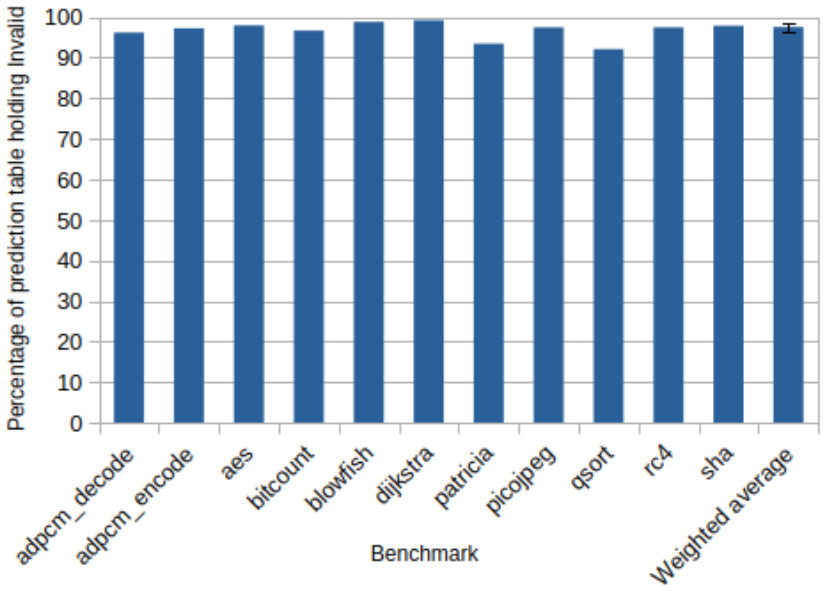

Figure 4.4: Percentage of the NAP PT holding `Invalid` halfway through each benchmark.

I suspect that this issue could be mitigated with the use of tags – some bits from the PC (or even global history!) would be stored in each PT entry to determine if the entry is relevant, allowing misses. I test this suspicion in the next section.

4.5 Hash-tagged Gselect

After performing this evaluation, I was unsatisfied and implemented the parameterisable HTGSEL predictor in section 3.11 to address the limitations found in the previous section. We now evaluate its performance relative to unmodified Toooba in figures 4.5 and 4.6. The BDP size is 50% bigger (without need nor effect – each PT entry has an unecessary validity bit) and the NAP size is almost identical to Toooba's BTB.

[17] The halfway point is used because it is representative of the workload within a benchmark rather than the start-up or wind-down phases

Legend:

- **Unmodified Toooba, using its implementation of the Gselect BDP and its BTB**
- **Toooba with instantiations of the parameterisable HTGSEL predictor for BDP and NAP**

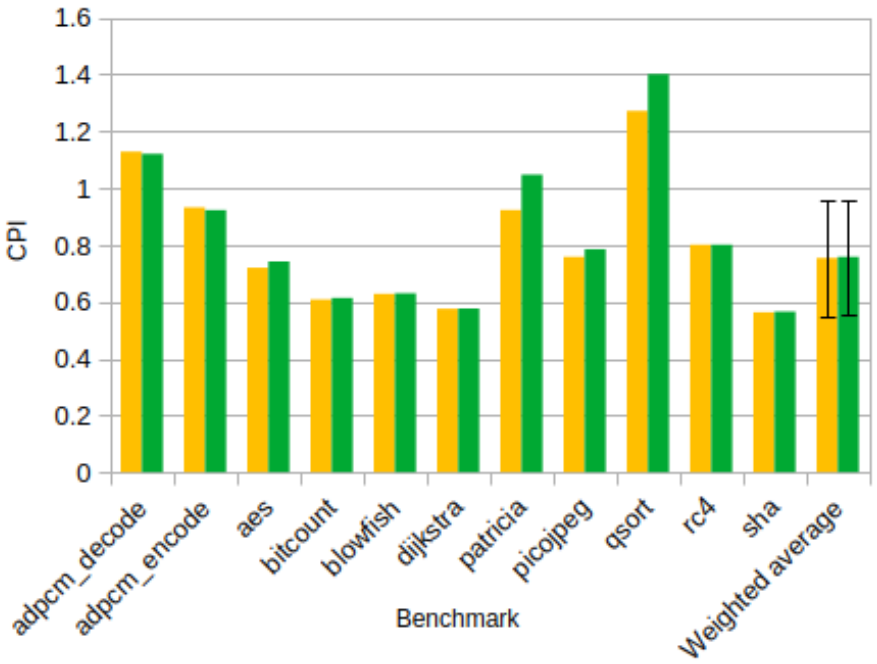

Figure 4.5: CPI for each benchmark.

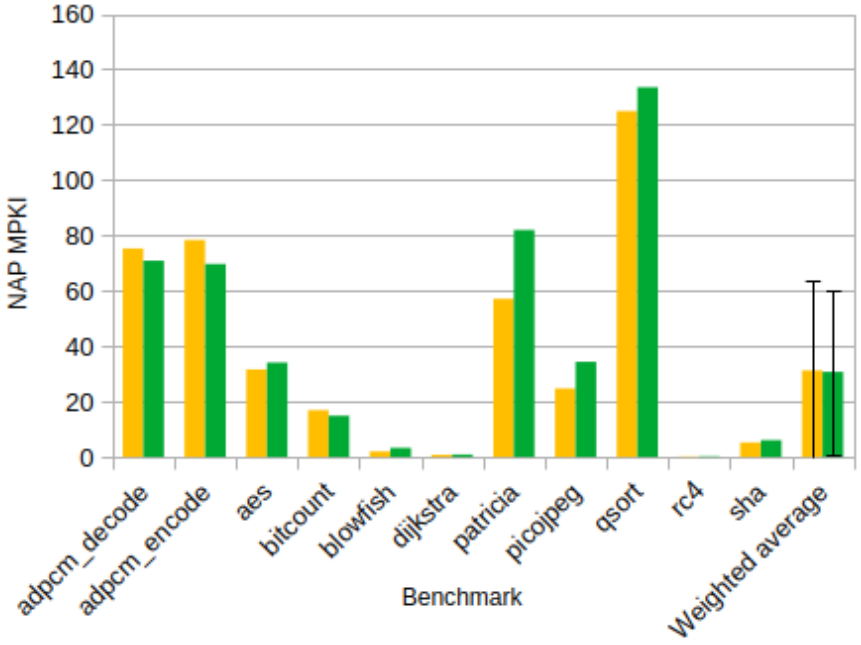

Figure 4.6: NAP MPKI for each benchmark.

We can see that the performance and accuracy of the HTGSEL NAP is equivalent to Toooba's BTB, meeting requirement C1 and making this project a resounding success!

Chapter 5

Conclusions

Parameterisable predictors provide the benefits of less total time for development and verification; a decrease in the attack surface for vulnerabilities like Spectre; and allowing advanced techniques and organisations to be shared between modes of prediction, making this work valuable to processor designers in academia and industry. This work also demonstrates the benefits and promotes the use of high-level HDLs like BSV – which allow source code to be neatly re-used through parameterisation and CMD – something that industry has been slow to adopt.

5.1 Contributions

In this dissertation, as described more thoroughly in section 1.5, I have contributed: a description of Tooba's front-end; an accepted fix for a bug in Tooba with security implications; the `ValueWithConfidence` datatype; the `TRegFile` module; modified BDP and NAP interfaces for Tooba with power benefits; the parameterisable Gselect predictor; the parameterisable HTGSEL predictor; and investigations into the accuracies of instantiations of the parameterisable predictors for BDP and NAP.

5.2 Future work

The contributions of this novel dissertation open many avenues for future work.

- Application to other modes of prediction, such as memory dependence predictors and prefetchers (requirement W1).
- Finding optimal parameters for instantiation of the parameterisable predictors, perhaps taking a gradient descent approach.
- Finding optimal organisations, for example,
  - TAGE – the state-of-the-art for BDPs – has a variant for branch target prediction [7] and has been shown to be effective for memory dependence prediction [8], and
  - Yasuo Ishii et al. [36] suggest forming target hashes from the instruction address and target, then XORing it into a GR that only partially shifts for BTBs. This approach would be interesting to explore and the parameterisable predictor could be extended to have another function parameter to allow combining global history in different ways.
- A version of the `TRegFile` that maps better to hardware, perhaps using a write queue.

I suggest some additional interesting future work in section 3.11.

5.3 Personal and professional development

This timely completion of this project required me to find the confidence to change direction when it was necessary, switching from TAGE to Gselect for the predictor organisation.

Throughout this project I learnt many best practices, including following a development methodology and using emergent design. To ensure the project went smoothly, I developed Linux knowledge and skills; used and designed tests; followed the framework of CMD; and used version control and back-ups. I learnt a new programming language and how to read code effectively. I solidified my knowledge of front-ends in microarchitectures, with specific focus on branch predictors and branch target buffers. Finally, I got to engage in the open-source community by contributing a bug-fix with security implications to Toooba which has been accepted.

Bibliography

[1] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] Brian Grayson et al. 'Evolution of the samsung exynos cpu microarchitecture'. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 40–51.

[3] Fritzchens Fritz. *Die Shot of AMD EPYC 7702 (Zen 2, Rome)*. 2019. URL: https://www.flickr.com/photos/130561288@N04/49045449908/ (visited on 12/04/2025).

[4] Ashley Webb. *High Detail 'Zen 2' CPU core layout*. 2020. URL: https://www.eridonia-archives.com/post/high-detail-zen-2-cpucore-layout-with-zen-1-for-reference (visited on 12/04/2025).

[5] Akira Fukuda. 'AMD's Zen 2 processor CPU core and chiplet technology announced at ISSCC'. In: *PC Watch* (2020). Title and author translated from Japanese. URL: https://pc.watch.impress.co.jp/docs/column/semicon/1236258.html (visited on 12/04/2025).

[6] Teja Singh et al. '2.1 zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core'. In: *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2020, pp. 42–44.

[7] André Seznec and Pierre Michaud. 'A case for (partially) tagged geometric history length branch prediction'. In: *The Journal of Instruction-Level Parallelism* 8 (2006), p. 23.

[8] Karl H Mose et al. 'MASCOT: Predicting Memory Dependencies and Opportunities for Speculative Memory Bypassing'. In: (2025).

[9] Arthur Perais and André Seznec. 'Cost effective speculation with the omnipredictor'. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–13.

[10] Paul Kocher et al. 'Spectre attacks: Exploiting speculative execution'. In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[11] Dmitry Evtyushkin et al. 'Branchscope: A new side-channel attack on directional branch predictor'. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 693–707.

[12] Sparsh Mittal. 'A survey of techniques for dynamic branch prediction'. In: *Concurrency and Computation: Practice and Experience* 31.1 (2019), e4666.

[13] Tianlin Huo et al. 'Bluethunder: A 2-level directional predictor based side-channel attack against sgx'. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 321–347.

[14] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft"*, RISC-V Foundation. 2019. URL: https://riscv.org/specifications/ratified/ (visited on 09/04/2025).

[15] C.H. Perleberg and A.J. Smith. 'Branch target buffer design and optimization'. In: *IEEE Transactions on Computers* 42.4 (1993), pp. 396–412. DOI: 10.1109/12.214687.

[16] Johnny KF Lee and Alan Jay Smith. 'Branch prediction strategies and branch target buffer design'. In: *Computer* 17.01 (1984), pp. 6–22.

[17] Andreas Moshovos. 'Memory dependence prediction'. PhD thesis. PhD thesis, University of Wisconsin-Madison, 1998.

[18] Scott McFarling. *Combining branch predictors*. Tech. rep. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[19] Kevin Skadron, Margaret Martonosi and D Clark. 'Speculative updates of local and global branch history: A quantitative analysis'. In: *Journal of Instruction-Level Parallelism* 2 (2000).

[20] Shien-Tai Pan, Kimming So and Joseph T Rahmeh. 'Improving the accuracy of dynamic branch prediction using branch correlation'. In: *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. 1992, pp. 76–84.

[21] Nathan Gober et al. *ChampSim*. Year is of most recent commit in adopted history. 2024. URL: https://github.com/ChampSim/ChampSim (visited on 13/04/2025).

[22] Nathan Gober et al. 'The championship simulator: Architectural simulation for education and competition'. In: *arXiv preprint arXiv:2210.14324* (2022).

[23] Rishiyur Nikhil. 'Bluespec System Verilog: efficient, correct RTL from high level specifications'. In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE. 2004, pp. 69–70.

[24] Paul Fox and Simon Moore. *Cambridge Bluespec Tutor*. 2014. URL: https://www-bluespec.cl.cam.ac.uk/ (visited on 15/04/2025).

[25] *BSV Reference Guides*. 2025. URL: https://github.com/B-Lang-org/bsc/releases/tag/2025.01.1 (visited on 15/04/2025).

[26] *Toooba: RISC-V Core; superscalar, out-of-order, multi-core capable; based on RISCY-OOO from MIT*. Bluespec Inc., 2019. URL: https://github.com/bluespec/Toooba (visited on 15/04/2025).

[27] Sizhuo Zhang et al. 'Composable building blocks to open up processor design'. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 68–81.

[28] Sizhuo Zhang. *RiscyOO Design Document*. Requires compilation with pdflatex. MIT CSAIL's Computation Structures Group. URL: https://github.com/csail-csg/RiscyOO_design_doc (visited on 20/04/2025).

[29] EEMBC. *CoreMark® Benchmark*. 2009. URL: https://www.eembc.org/coremark/ (visited on 21/04/2025).

[30] M.R. Guthaus et al. 'MiBench: A free, commercially representative embedded benchmark suite'. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[31] John McGuire. *Requirements Analysis for Software Development (Guide)*. Pulsion. URL: https://www.pulsion.co.uk/blog/requirements-analysis-for-software-development/ (visited on 21/04/2025).

[32] RISC-V Foundation. *RISC-V Tests*. URL: https://github.com/riscv-software-src/riscv-tests/.

[33] Andrew Waterman et al., eds. *RISC-V privileged specification version 1.9.1*. RISC-V Foundation. URL: https://riscv.org/specifications/ratified/ (visited on 12/05/2025).

[34] Sarah Harris and David Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[35] Agile Sherpa. *Agile Emergent Design*. URL: https://web.archive.org/web/20110501130212/http://www.agilesherpa.org/agile_coach/engineering_practices/emergent_design/ (visited on 04/05/2025).

[36] Yasuo Ishii et al. 'Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective'. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 172–182. DOI: 10.1109/ISPASS51385.2021.00034.

Appendix A: Toooba TLB micro-cache bug-fix

src_Core/RISCY_OOO/procs/RV64G_OOO/FetchStage.bsv +23 -5

```
@@ -344,58 +344,76 @@
344 344         nextAddrPred.put_pc(pc_reg[pc_final_port]);
345 345     endrule
346 346
347     -   Reg#(Vector#(PageBuffSize,Maybe#(Vpn))) buffered_translation_virt_pc <- mkReg(replicate(Invalid));
    347 +   Reg#(Vector#(PageBuffSize,Maybe#(Addr))) buffered_translation_virt_pc <- mkReg(replicate(Invalid));
348 348     Reg#(Vector#(PageBuffSize,TlbResp)) buffered_translation_tlb_resp <- mkRegU;
349 349     Reg#(Bit#(TLog#(PageBuffSize))) buffered_translation_count <- mkRegU;
350 350
    351 +   function Maybe#(UInt#(TLog#(PageBuffSize))) matchingVpnOrInvalidAddress(Addr pc);
    352 +       // Maybe return an entry with the same VPN or same full PC if its an invalid address.
    353 +       if (validVirtualAddress(pc)) begin
    354 +           // Compare VPNs.
    355 +           function Maybe#(Vpn) getVpnIfPossible(Maybe#(Addr) maybePc);
    356 +               if (maybePc matches tagged Valid .pc) getVpnIfPossible = Valid(getVpn(pc));
    357 +               else getVpnIfPossible = Invalid;
    358 +           endfunction
    359 +           matchingVpnOrInvalidAddress = findElem(Valid(getVpn(pc)), map(getVpnIfPossible, buffered_translation_virt_pc));
    360 +       end else
    361 +           // Compare entire PCs.
    362 +           matchingVpnOrInvalidAddress = findElem(Valid(pc), buffered_translation_virt_pc);
    363 +   endfunction
    364 +
351 365     rule invalidate_buffered_translation(!iTlb.flush_done);
352 366         buffered_translation_virt_pc <= replicate(Invalid);
353 367     endrule
354 368
355 369     // getTlbResp catches a iTLB translation and writes it into translation
356 370     // buffer. If there is an active iTlb flush, clear the buffer.
357 371     rule getTlbResp;
358 372         // Get TLB response
359 373         TlbResp tr <- tlb_server.response.get;
360 374         translateAddress.deq;
361 375         if (iTlb.flush_done) begin
362 376             // Check if, because of pipelining, we already have this vpn.
363     -           Bool found = elem(Valid(getVpn(translateAddress.first)), buffered_translation_virt_pc);
364     -           if (!found) begin
365     -               buffered_translation_virt_pc[buffered_translation_count] <= Valid(getVpn(translateAddress.first));
    377 +           let pc = translateAddress.first;
    378 +           if (!isValid(matchingVpnOrInvalidAddress(pc))) begin
    379 +               // If we don't have this VPN or the PC is invalid and doesn't match in full:
    380 +               buffered_translation_virt_pc[buffered_translation_count] <= Valid(pc);
366 381                 buffered_translation_tlb_resp[buffered_translation_count] <= tr;
367 382                 buffered_translation_count <= buffered_translation_count + 1;
368 383             end
369 384         end else buffered_translation_virt_pc <= replicate(Invalid);
370 385         if (verbosity >= 2) $display ("%d Fetch Translate: pc: %x, ", cur_cycle, translateAddress.first, fshow (tr));
371 386     endrule
372 387
373 388     // doFetch1 pulls a prediction out of the BTB and attempts to translate it
374 389     // from a small buffer (~2) of recent TLB translations.
375 390     // If the necessary translation is not in the buffer, doFetch1 submits a TLB
376 391     // lookup request and then retrys until getTlbResp has populated the buffer
377 392     // and the lookup succeeds.
378 393     rule doFetch1(started && !waitForRedirect[0] && !waitForFlush[0]);
379 394         let pc = pc_reg[pc_fetch1_port];
380 395
381 396         // Grab a chain of predictions from the BTB, which predicts targets for the next
382 397         // set of addresses based on the current PC.
383 398         Vector#(SupSizeX2, Maybe#(Addr)) pred_future_pc = nextAddrPred.pred;
384 399
385 400         // Next pc is the first nextPc that breaks the chain of pc+4 or
386 401         // that is at the end of a cacheline.
387 402         Vector#(SupSizeX2,Integer) indexes = genVector;
388 403         function Bool findNextPc(Addr in_pc, Integer i);
389 404             Bool notLastInst = getLineInstOffset(in_pc + fromInteger(2*i)) != maxBound;
390 405             Bool noJump = !isValid(pred_future_pc[i]);
391 406             return (!(notLastInst && noJump));
392 407         endfunction
393 408         Bit#(TLog#(SupSizeX2)) posLastSupX2 = fromInteger(fromMaybe(valueof(SupSizeX2) - 1, find(findNextPc(pc), indexes)));
394 409         Maybe#(Addr) pred_next_pc = pred_future_pc[posLastSupX2];
395 410
396 411         // Search the last few translations to look for a match.
397     -       Maybe#(UInt#(TLog#(PageBuffSize))) m_buff_match_idx = findElem(Valid(getVpn(pc)), buffered_translation_virt_pc);
    412 +       Maybe#(UInt#(TLog#(PageBuffSize))) m_buff_match_idx = matchingVpnOrInvalidAddress(pc);
398 413         if (m_buff_match_idx matches tagged Valid .buff_match_idx) begin
    414 +           // Invalidate the buffered TLB response if it was for an invalid virtual address.
    415 +           if (!validVirtualAddress(pc))
    416 +               buffered_translation_virt_pc[buff_match_idx] <= Invalid;
399 417             let next_fetch_pc = fromMaybe(pc + (2 * (zeroExtend(posLastSupX2) + 1)), pred_next_pc);
400 418             let pc_idxs <- pcBlocks.insertAndReserve(truncateLSB(pc), truncateLSB(next_fetch_pc));
401 419             PcIdx pc_idx = pc_idxs.inserted;
```

Glossary

`SupSizeX2` twice the value of `SupSize` 16, 26, 30, 32

`SupSize` the superscalar width of Toooba – how many instructions there can be in each pipeline stage in each cycle , 16, 26, 28, 30, 32

`TRegFile` transactional register file (§ 3.7) 11, 20, 21, 29, 35, 41

`ValueWithConfidence` remembered value with n-bits of confidence (§ 1.3, § 3.4) 11, 20, 21, 24, 30, 34, 41

batch instructions in the same stage of a superscalar pipeline 16, 17, 23

ChampSim Championship Simulator (§ 2.2) 11–13, 17, 22, 23

compressed an extension of RISC-V where instructions may be two bytes (compressed) instead of four bytes only , 9, 14, 23

CoreMark® a benchmark 17, 20

emergent design refactoring the commonalities out of similar components (§ 3.9) 20, 21, 31, 42

global history the results of the most recent prior predictions on any address (§ 2.1) 10, 12, 13, 17, 24, 25, 30–33, 39, 41

Gselect a branch direction predictor (§ 2.1) , 10–13, 17, 18, 20–22, 24–26, 28, 30–33, 35–39, 41, 42

Gshare similar to Gselect 12, 13, 30

instruction fragment two bytes that may represent a whole compressed instruction or half of a non-compressed instruction 9, 14, 16, 25, 34

MiBench an embedded benchmark suite 17, 20

misspeculation incorrect speculation 8, 15

prediction token a value returned by a predictor when making predictions and passed back to it to identify which training information to use, used in modified Toooba 11, 26–29

RiscyOO an open-source RISC-V CPU which Toooba is based on 14, 15, 19

saturating counter a finite-state machine that cannot go below or above boundary values 9, 10, 23, 24

Spectre a speculative execution vulnerability 8, 11, 41

superscalar a pipeline that processes multiple instructions per stage per cycle , 14, 16

TAGE state-of-the-art BDP – this project was originally going to implement it 8, 10, 38, 41, 42

Toooba an open-source RISC-V CPU (§ 2.4) , 8–23, 25–28, 32–34, 36–42

training information information returned by a predictor when making predictions and passed back to it for training, used in unmodified Toooba and used differently in modified Toooba , 11, 16, 17, 23–29

BDP branch direction prediction or branch direction predictor (§ 1.2.1) , 7–32, 34–39, 41

BSV Bluespec SystemVerilog (§ 2.3) 8, 10–14, 17, 19–21, 23, 29, 35, 38, 41

BTB branch target buffer (§ 1.2.2) 8–10, 16, 18, 25, 32–34, 36–41

CMD composable modular design (§ 2.8.2) 14, 19, 41, 42

CPI clock cycles per instruction 17, 36, 37, 40

GR global history register 12, 13, 24, 34, 41

HDL hardware description language 13, 14, 41

HTGSEL hash-tagged Gselect 11, 21, 33, 35, 39–41

MPKI mispredictions per thousand instructions 35, 37, 40

NAP next-address prediction or next-address predictor (§ 1.2.2) 8–12, 14–22, 25–28, 30–41

PC program counter 7–9, 12, 14, 16, 17, 23–26, 30–33, 39

PT prediction table 8, 12, 13, 16, 17, 23–26, 29–36, 39

TLB translation look-aside buffer 14, 15, 21

Project proposal

Part II Project Proposal: Parameterised TAGE Prediction Module for Toooba

[redacted]

October 20, 2024

1 Introduction

Toooba [1] is a superscalar out-of-order RISC-V core developed by Bluespec in Bluespec SystemVerilog (BSV) and based on MIT's RiscyOO. Toooba supports a small suite of branch predictors, including GSelect, GShare, and a tournament predictor. It does not, however, support the state-of-the-art TAGE predictor.

TAGE (TAgged GEometric) [2] is a conditional branch predictor that uses multiple global history lengths hashed with the branch address to index into predictor tables. The prediction hit matched by the longest history is used. A global history is a binary sequence of taken/not taken branches. The lengths of the histories used form a geometric series. TAGE has also been applied to indirect branches where the target depends on register state, as ITTAGE [2], and combined wth TAGE to form COTTAGE [2].

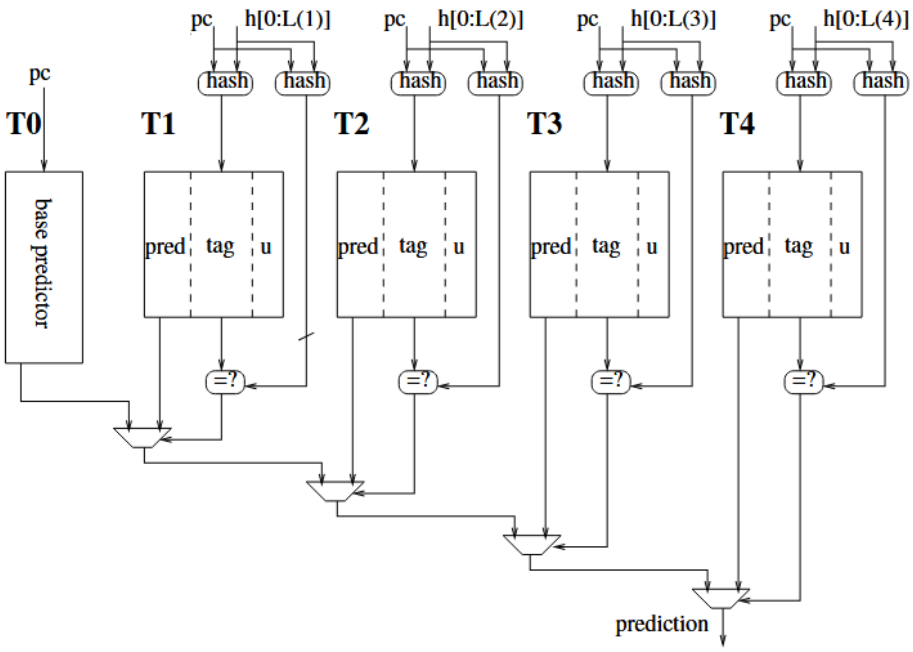

Figure 1: A 5-component TAGE predictor [2]

2 Description of Work

The start of this project will be to implement a TAGE predictor for Toooba that is parameterisable in what datatypes it predicts on and the type and lengths of history used. The predictor will be parameterised for direction prediction as well as target prediction (indirect branches) and its accuracies will be evaluated in ChampSim on the same benchmarks as in [2]. These benchmarks are from the Championship Branch Prediction [3]. Toooba's overall simulated performance in MiBench/CoreMark will also be used as a metric.

Extensions

Documented instances of exact application of TAGE methods to other modes of prediction in computer architecture have not been found so this could represent space for novel use-cases, such as memory dependence prediction, instruction/data cache miss prediction, and prefetching. Implementing the predictor for these modes and evaluating against Toooba's overall baseline performance will form extensions to the project.

Synthesis can form another extension to the project, with opportunity to evaluate timing and area characteristics and performance on benchmarks.

3 Project Plan

Dr Jonathan Woodruff has agreed to supervise this project in writing. Dr Robert Mullins has agreed to be a UTO supervisor for all of Dr Woodruff's students.

I will use my own laptop for this project using Git and GitHub for revision history and file backup. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. The Bluespec compiler, simulator, and IDE will be required. (Academic) licenses are needed.

Success Criteria

1. Evaluate Toooba's performance on

   (a) its baseline performance in MiBench/CoreMark;

   (b) its branch prediction accuracy with GSelect and GShare using the CBP benchmarks in ChampSim.

2. Design unit tests for a parameterised TAGE prediction module.
3. Design implementation tests for the parameterised TAGE prediction module with Toooba.
4. Implement the parameterised TAGE prediction module.
5. Instantiate and integrate the parameterised TAGE prediction module for branch direction prediction in Toooba.
6. Evaluate the performance of the parameterised TAGE prediction module for branch direction prediction in Toooba.
7. Instantiate and integrate the parameterised TAGE prediction module for branch target prediction in Toooba.
8. Evaluate the performance of the parameterised TAGE prediction module for branch target prediction in Toooba.
9. Evaluate Toooba's new performance in MiBench/CoreMark.

Starting Point

I have done background reading on TAGE, namely the foundational paper [2]. I have completed the first 3 exercises (of 8) of the Cambridge Bluespec Web-Tutor. I have inspected the source code for Toooba [1] to see which pipeline stage branch predictors are updated/used.

Work Plan

The following plan is broken into 10 work packages, each taking approximately 2 weeks, and beginning in the middle of week 2 of Michaelmas term (2024). There is a 1 week break at the end of each term to catch up on supervision work and a 2 week break for Christmas/New Year's.

1. 21st Oct – 1st Nov
   Learn Bluespec SystemVerilog via the CL's web tutor. Study prior work on TAGE.

2. 4th Nov – 15th Nov
   Set-up GitHub repository. Evaluate Toooba's baseline performance with MiBench and/or CoreMark.

3. 18th Nov – 29th Nov
   Evaluate Toooba's branch prediction accuracy with its implemented branch predictors. This will require compiling (to C++ with Verilator) and adapting BSV source code to C++ compatible with ChampSim.

4. 9th Dec – 20th Dec
   Plan the implementation of the parameterised TAGE module.

5. 6th Jan – 17th Jan
   Unit tests and integration tests for the parameterised TAGE module.

6. 20th Jan – 31st Jan
   Begin to implement the parameterised TAGE module. Progress Report.

7. 3rd Feb – 14th Feb
   Implement the parameterised TAGE module (cont.).

8. 17th Feb – 28th Feb
   Instantiate and integrate the parameterised TAGE module for branch direction prediction in Toooba. Evaluate accuracy in ChampSim.

9. 3rd Mar – 14th Mar
   Instantiate and integrate the parameterised TAGE module for branch target prediction in Toooba. Evaluate accuracy in ChampSim.

10. 24th Mar – 4th Apr
    Evaluate Toooba's new performance with MiBench and/or CoreMark.

There are 6 weeks remaining after package 10 which will be used for write-up and extensions.

References

[1] Bluespec, Inc., "Github - bluespec/toooba: Risc-v core; superscalar, out-of-order, multi-core capable; based on riscy-ooo from mit," 2024, last accessed 17 October 2024. [Online]. Available: https://github.com/bluespec/Toooba

[2] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *J. Instr. Level Parallelism*, vol. 8, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:6159849

[3] J. Instr. Level Parallelism, "The 1st JILP Championship Branch Prediction Competition (CBP-1)," 2004, last accessed 17 October 2024. [Online]. Available: https://jilp.org/cbp/