The Simulation-based Gold-Standard framework for verifying HDL branch predictors

Katy Thackray [1] Karl Mose [1]

[1]University of Cambridge

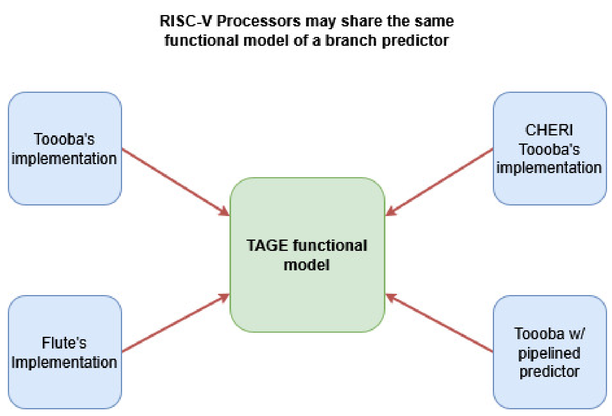
Branch Predictors: Software vs Hardware implementations

Software implementations: State of the art branch predictors are published in the form of a high level programming language implementation. These languages provide don't worry about the low level implementation details and are straightforward to test and alter throughout development.

Hardware optimizations: Software implementations do not take advantage of **hardware optimizations** relevant in real designs.

- They may be pipelined, particularly for cascaded schemes.
- They may use tree structures to increase parallelism in the prediction function
- Other implementations such as ahead-pipelining or folded history

Relationship: The relationship between HDL implementations and functional specifications is many-to-one and a software implementation can easily define the functional specification.



Challenge: The translation of these software implementations to hardware is non-trivial!

The importance and challenges in verifying HDL predictors

1. An incorrect branch predictor won't cause a processor to behave erroneously, it will just worsen IPC. This makes bugs hard to spot.
2. SoA art branch predictors are incredibly context sensitive and low impact issues in the programming could be disastrous for certain benchmarks
3. in real SoA hardware implementations there are far too many cases to reliably fully cover using a set of test benches

Challenge: Debugging HDL predictors can be like finding a needle in a hay stack

The purpose of the SGV framework

The Simulation-based Gold-Standard framework (or SGV framework) is a more principled approach to developing branch predictorsa and aims to solve these challenges

Two branch predictor models

As input to the SGV framework, two branch predictors are provided:

- A branch predictor written in a high level language acts as the gold-standard functional model. In the SGV framework this predictor is written in C++
- A branch predictor written in a HDL, Bluespec SystemVerilog (BSV), is the hardware model being tested against the functional model.

Gold-standard testing

- These two models are run simultaneously in lockstep on billions of instructions from a set of traces you can choose. The framework halts instantly when the output between the two models differ.
- After a halt the source of the divergence can be traced using the logs generated by the framework.

Through sheer brute force even miniscule bugs can be found with this method.

The SGV framework

- SGV is implemented on top of Champsim, a trace-based simulator
- With SGV, Champsim simulates and compares a C++ branch predictor and a predictor written in Bluespec SystemVerilog (BSV)
- To bridge a HDL such as BSV to Champsim, which is written in C++, the framework uses OS FIFOs to communicate with a separate process running a **Bluesim** test-bench that passes predictions to and from the BSV predictor model.

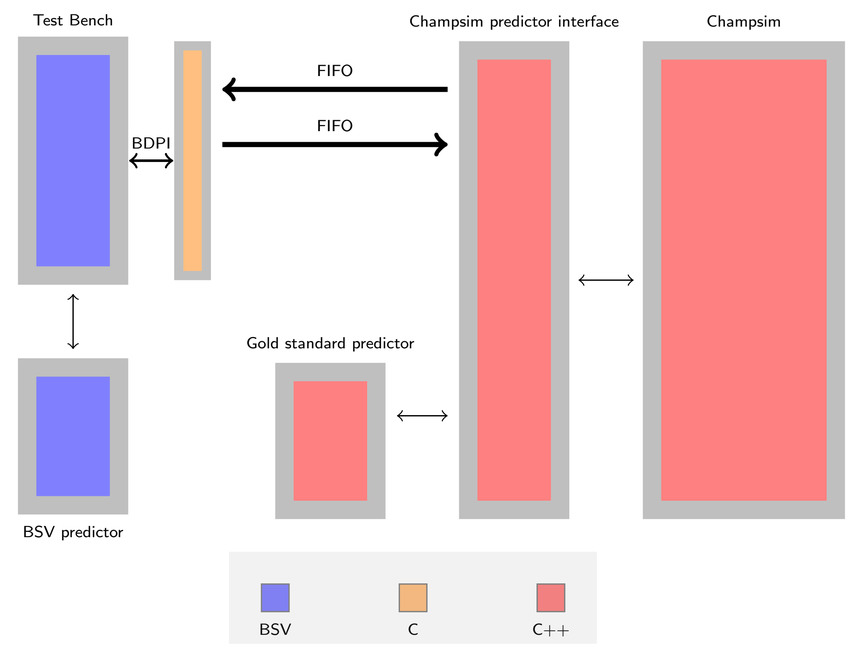


Figure 1. Architecture of the gold standard model

- The Bluespec SystemVerilog library **BDPI** is to call C functions from a BSV simulation
- The Champsim simulator simulates branch prediction updates to be immediate and in-order.
- The fact that Champsim is **trace-based** is exploited to send predictions across the FIFO ahead of time. This allows performance is of the SGV framework to be comparable to running Champsim without the SGV framework.

Using the SGV framework

Our investigation

In our investigation we investigate using the SGV framework to debug a **TAGE** based predictor written in BSV for the Toooba processor. The interface used in the SGV framework was made to match the processor's branch prediction interface. No changes to the HDL code were required.

Hardware modules written for Software predictors

To use the SGV framework it was necessary to implement a C++ based version of TAGE that's more faithful to a hardware-based implementation than software-based predictors usually are. The HDL predictor uses an LFSR and for the two models to perfectly converge this was necessary. For the models to have the same outputs it was necessary to implement the LFSR in C++ rather than use a random number generator.

Usage

The SGV framework is effective in the later stages of development after some testing has already taken place directly in the HDL. The output streams of both predictor models are stored in logs throughout the simulation used to trace the source of bugs after the framework halts.

Faults found with the SGV framework

Using SGV we were able to quickly find divergence in behaviour between the software and hardware based predictor.

1. Bit manipulations: A few thousand predictions into the simulation the framework detected a fault which incorrectly left-shifted the PC passed into the training data.
2. Incorrect logic: Tracing the cause of another halt from the simulator revealed that the hardware implementation, when choosing the first table as the provider, would incorrectly also mark the first table as the alternative table.
3. Table allocation: 85,000 branch instructions into the simulation. The root cause was the fact that the BSV TAGE predictor was decrementing the useful counter of all entries above *and including* the provider entry, when it should only be the entries above the provider entry.

By searching for specific indices in the logs we could trace back allocations and state updates to find the source of each halt in the testing framework. Particularly for the 3rd bug it's highly likely such a discrepancy would have remained undetected.

Conclusion of findings and future work

Effectiveness: The SGV framework successfully found multiple bugs in the predictor, even after testing was done on individual components in the predictor. This was found to be particularly effective for validating iterations of the same predictor throughout development.

Development methodology: This framework particularly fits a development process that begins from an inefficient HDL predictor and iterating increasingly optimised versions. The new branch predictor produced from this process improved MKPI in Toooba by 44%.

Future work: Future work could involve testing state of the art predictors such as Tage-SC-L. The framework could further be improved to allow for more flexible logging and halting on differing internal state rather than the prediction. The framework could also be extended to allow out of order branch prediction updates to increase the testing coverage.

km7816@cam.ac.uk

kt526@cam.ac.uk