Tool for analysing BPU perfomance

Danil Slinchuk
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
ORCID: 0009-0002-8440-0433

Aleksei Efremov
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
ORCID: 0009-0004-5510-5923

Vladimir Kutuev
YADRO Lab
St. Petersburg State University
St. Petersburg, Russia
v.kutuev@spbu.ru

***Abstract*—When designing processors, one needs to compare their individual components, such as the BPU (branch prediction unit), with solutions on the market. This leads to the problem of finding or generating specific test cases that perform differently on various BPUs. These test cases serve as a starting point for further analysis. To solve this problem, one can employ fuzzing, testing method based on randomly generation input data. This paper describes a tool that allows you to test and compare various BPUs in practice. The tool generates a set of random tests, runs them, collects counters from the CPU, and provides data for analysis. Additionally, we have obtained results for several CPUs and compiled the tests into a benchmark, which demonstrates the differences between BPUs. All technical intricacies are described and justified.**

***Index Terms*—branch prediction unit, fuzzing, code generation, microarchitecture analysis, CPU analysis, performance counters, BPU comparison, Gem5, RISC-V.**

I. Introduction

Since the emergence of computing technology, there has always been a desire to increase the performance of program execution. One of the important ways to speed up the processor is the method of instruction pipelining [1], which allows processing several instructions simultaneously, dividing their execution into a series of sequential steps. However, its efficiency is not always constant. For example, conditional and unconditional branches can cause control hazards [2], which leads to idling of processor components and reduces the efficiency of the pipelining method to zero. A special module, the branch prediction unit (BPU), is used to partially solve this problem. BPU attempts branch prediction to reduce the processor downtime during jumps.

BPU is a part of the processor's microarchitecture. The intricacies of its implementation and factors influencing its efficiency in most processor families of companies like Intel and AMD are unknown to us, as their architecture implementations are proprietary. At the same time, the open-source RISC-V Instruction Set Architecture (ISA) is actively developing and gaining popularity. So identifying scenarios where the BPU in one processor shows significantly worse results than the BPU in common modern CPUs can assist processor developers in improving the BPU itself and developing the RISC-V architecture through a detailed examination of such situations [3]–[5]. To address this task, the development of a tool was initiated.

Fuzzing is used to find scenarios that reveal differences in the efficiency of different BPUs. The method is based on repeatedly executing randomly generated code to find blocks of instructions that reveal weaknesses of BPUs. It's worth mentioning that fuzzing is used specifically because there are many different types of BPU with different internal devices and that use different algorithms. Simple tests, for example, to determine the size of the BTB (Branch Target Buffer) may not be sufficient [6], [7], as modern BPU devices can be very complex [8]. Fuzzing can also help reveal incorrect behavior of BPU and bugs in its implementation, we implemented the tool based on these ideas. Its execution proceeds in 3 main stages:

- *generate* — random test generation in the C language;
- *analyse* — building, running tests and collecting data from CPU counters containing information about BPU's operation;
- *summarize* — filtering and analysing the results, graphing and saving the data.

The tool is available on GitHub [9].

The rest of the paper is organized as follows. Subsection 'Architecture' demonstrates how the tool works and outlines its main architecture. Subsection 'Test generation' describes code generation and the possible problems associated with it. Subsection 'Data collection' describes the methods used to collect data from CPU counters, methods to improve reproducibility, and some of the optimizations used. The Subsection 'Summarize results' outlines the comparison metrics for the BPU. Section 'Evaluation' presents the results of comparing the BPUs of several processors for each of the architectures: RISC-V, x86, ARM64 obtained using our tool, along with the conclusions drawn from these results. In Section 'Future work', we discuss ways to develop the tool and its possible improvements.

II. Background

There are several well-known types of BPUs that differ in their operating principles and efficiency [10]. All BPUs can be divided into two groups: static and dynamic. Static branch prediction is the simplest technique as it only uses information about the branch instruction and does not rely on the dynamic code execution history. However, it has been widely used in practice: the early implementations of MIPS and SPARC (the first commercial RISC architectures) employed this technique.

Dynamic branch prediction utilizes information about taken or untaken branches collected during execution to predict branch outcomes. There are quite a few subtypes of this technique and they all demonstrate varying effectiveness. Examples include:

- 1-bit saturating counter;
- 2-bit saturating counter;
- correlating branch prediction buffer;
- tournament branch predictor;
- branch target buffer;
- return address predictors;
- integrated instruction fetch units;
- next-line prediction;
- local branch prediction;
- global branch prediction.

It is known that the major processor manufacturers (Intel and AMD) have used some well-known branch prediction techniques. For example, the non-MMX Intel Pentium processor used a saturating counter; The Intel Pentium MMX, Pentium II, and Pentium III had local branch predictors; Pentium M, Core, Core 2, and Silvermont-based Atom processors employed global branch prediction. However, when it comes to modern flagship processors we do not know the specific BPU implementations, so comparing them can only be done through thorough testing.

It is worth noting that the BPU is a microarchitectural component that affects not only the system's overall performance but also its security. The widely known Spectre [11] security vulnerability is based on branch prediction and the use of the BPU. Speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers. The most recent vulnerability of this type, Spectre-HD, was discovered in February 2023, also known as "Spectre v6" or "Spectre SRV".

III. Related works

Different BPU implementations demonstrate different effectiveness [3]–[6] as there are numerous types and implementations. Due to the inability to directly compare several BPUs, the fuzzing method is utilized in the tool as it has been widely used for testing other hardware components [12]. A search was conducted for tools capable of comparing BPUs and below are several projects addressing similar tasks. However, no direct alternatives to our tool were found.

A. Overview of alternatives

AFL-fuzz-compiler [13] is a specialized AFL version designed for fuzzing compilers. However, since it was not originally intended as a means to optimize system performance, its results may be uninformative for the task at hand. For the same reason, it is unable to conduct measurements and provide conclusions reflecting the actual system performance.

Coremark [14] is a portable test suite developed to measure processor performance. After compilation and execution, Coremark provides an estimate of the processor performance. However, it is important to consider that the tests included in Coremark are aimed at evaluating the entire processor microarchitecture, so it is not possible to extract results that solely apply to the BPU. It is also worth noting that the tests are pre-created, and their quantity is limited.

Championship Branch Prediction [15], [16] (CBP) competitions for developing BPU models were last held in 2016. To evaluate the models presented by competition participants, a special test suite was used, consisting of pre-recorded execution traces of various programs, primarily consisting of branch instruction sequences. These traces have a unique format, which complicates their application outside of the testing system, including attempts to assess real processors' performance with their help.

B. Overview of ways of obtaining counters

Analysis of methods for obtaining data from the BPU was also carried out since there are several different options.

Perf [17] is a kernel-based profiler for Linux that collects data on program performance by reading hardware and software counters or tracing it during execution. Hardware counter readings are obtained through a specific kernel module.

OProfile [18] is a performance analysis tool for Linux that utilizes built-in performance monitoring capabilities provided by the processor to gather information on kernel operation, executed modules, and running programs.

Cachegrind [19] is a utility included in the performance profiling toolset Valgrind. It simulates program operation with the processor's cache memory and branch prediction, providing approximate values of various program performance-related parameters.

Gem5 [20] is a processor simulator where you can prototype your processor, including its individual components such as the BPU. You can then run and analyse this model within Gem5 before creating a physical copy of the processor. As it is a simulator, it provides a wide range of diverse and reproducible data and traces, which are typically not guaranteed on an actual processor.

C. Review results

Initially, the option of using Cachegrind to study BPU performance was considered. However, it was found that Cachegrind is not the optimal choice as it can only simulate BPU operation and provide data based on simulation. Therefore, the metrics obtained using Cachegrind may significantly differ from real data. Even the creators of this tool acknowledge this fact and suggest using Perf for more accurate results.

During the review of the toolset, an analysis was also conducted on the possibility of using Perf and OProfiler. Both tools were found to be very similar to each other and for the tasks at hand they use the same kernel module. As these tools only allow collecting processor counters from the entire process execution and require installation, the decision was made to directly use kernel system calls that they utilize instead.

Finally, to enable data retrieval from the simulated processor model, it was decided to implement support for the powerful processor simulator — Gem5. This tool provides the ability

to prototype the BPU and study its operation, allowing for in-depth analysis and development of this processor component before its physical implementation.

As a result, the decision was made to choose the Linux kernel system calls and Gem5 for BPU metric measurements based on the considerations outlined above.

IV. Implementation

In this section, we describe the capabilities of the tool as of April 1.

A. Architecture

The tool consists of three main stages:

- *generate* — test generation;
- *analyse* — running the tests and collecting CPU counters;
- *summarize* — creating graphs and summary tables.

To use the tool, a CLI was developed, allowing each stage to be executed separately, as well as an *aggregate* driver that connects all stages for sequential execution. Additionally, the driver allows the stage *analyse* to be executed multiple times to obtain data from different BPUs on the same set of tests. Thus, users can access each component separately to achieve their goals, for example running custom tests, and the tool is easy to extend.

B. Test generation

In the *generate* stage tests are created. The generation starts with the randomized construction of an abstract syntax tree (AST) that includes constructs creating a load on the BPU: loops and branching, as well as various calculations to modify branching conditions. The constructed AST is then transformed into C code to be compiled for a wide range of architectures and written to the test file. Test generation should be randomized, yet controlled. For example, to prevent the generation of infinite loops (which make tests uninformative), the use of variables in loop declaration and within its body can be limited. Additionally, in the current generation process, it is possible to adjust the probability of the appearance of particular constructs in a test, the maximum nesting depth, and the number of constructs at each level.

C. Data collection

In the *analyse* stage, the tool runs tests and collects data on the BPU's performance. The tool can do this in two different ways: using the Linux kernel and the Gem5 processor simulator. Let's delve deeper into the former method. Before running the test, the tool wraps it with special functions. These functions utilize system calls and isolate the area under test. Various methods are employed in the analysis to enhance the accuracy and repeatability of testing results. To minimize the probability of test execution being preempted, the tool binds the test process to a core, increases its priority, and uses a FIFO scheduler. To ensure more reliable data, the tool gathers information only from the section of the code under test, excluding the code related to process initialization, runs tests multiple times, and subtracts the values spent on setting up the test environment from the final test results. This approach allows the user to avoid the necessity of manually configuring the system environment and conducting numerous measurements to obtain reliable data.

In addition to obtaining data from a real processor, the tool also supports collecting information from the Gem5 processor simulator. In Gem5 the user can model their processor, including its individual components such as the BPU. Thus, a processor developer may not need to create a new physical processor but can analyse and test a new BPU model for their central processor using the simulation model in Gem5. This provides developers with a convenient and efficient way to verify and refine models of new processor components before their implementation.

The tool can also run tests on remote setups using SSH with machines operating in parallel to reduce testing time. It can be useful when testing multiple processors or individual setups.

D. Summarize results

The metric chosen to compare BPUs is the percentage of mispredicted branches. The lower this percentage, the better the BPU performed on that test. After collecting data from all tests the tool calculates this metric for each test and its average for each tested BPU. Then a graph is plotted representing the BPU's performance across all tests which helps easily identify interesting and anomalous tests and draw conclusions on which of the BPUs, on average, shows better results. At the end of the *summarize* stage the tool saves all the calculated data for each BPU.

Thus, after using the tool the user receives a set of generated tests, data on the BPU's performance for each test and a graph for quick analysis.

V. Evaluation

In this section we will discuss the results obtained using the tool in practice. In total, 10 devices with 3 different architectures were tested.

- x86:
  - AMD Ryzen 7 4700;
  - AMD Ryzen 9 7900X;
  - AMD Ryzen Threadripper 2970WX;
  - 11th Gen Intel Core i5-1135G7;
  - Intel Core i7-6700;
  - Intel Core i5-8250U.
- ARM64:
  - Raspberry Pi: Broadcom BCM2711 Cortex-A72 (ARM v8) 64-bit;
  - Odroid: Samsung Exynos5 Octa ARM Cortex-A15 2GHz and Cortex-A7 1.3GHz.
- RISC-V:
  - VisionFive 2: StarFive RISC-V JH7110;
  - LicheePi 4A: TH1520 RISC-V C910.

The devices were divided into two groups: low-end and high-end. The first group included all devices with ARM and RISC-V architectures while the second group included x86. Several

hundred tests were generated for each architecture and indicative and anomalous tests within each group were combined into benchmarks. The results for each group are presented in *Fig.* 1 and *Fig.* 2. Assessing the accuracy of the measurement method is problematic, as it can vary greatly from processor to processor. Thus, after 10 runs of our benchmark with the following parameters: "timeout": 10, "max_test_launches": 50. For AMD Ryzen 7 4700 and StarFive processors, the average value of the standard error for all tests was 0.04% and 0.21%, respectively, and the maximum value of the standard error reached 1.6% and 7.6%, respectively, which shows a difference in errors of about 5 times. This is justified by the fact that different processors contain different BPU performance counters: there may be no counter for the total number of branches, but instead there will be a counter for correctly predicted branches, and accordingly, each counter will have its own error. It is worth noting that only indicative tests are represented on the graphs allowing for a comprehensive view. The X-axis shows the generated tests and the Y-axis represents the misprediction rate for a specific BPU. All tests are sorted by increasing maximum of misprediction rate. Let's take a closer look at each of the graphs.

A. low-end

The *Fig.* 1 shows that on tests with the lowest misprediction rates the BPUs show fairly similar results. However, further on LicheePi and Odroid become outsiders and show worse results. There are also anomalous tests such as `test_71` and `test_63` where Raspberry's BPU shows a high misprediction rate.

B. high-end

As can be seen from the *Fig.* 2 the average misprediction rate compared to low-end has significantly decreased, with a maximum misprediction rate of about 16 while for low-end this value is around 35. It is also worth noting that for tests with misprediction rates less than two the outsiders are Intel Core i5-1135G7 and AMD Ryzen Threadripper 2970WX.

C. Results for Gem5

In addition to comparing BPUs on real processors, an analysis of various BPUs configured in gem5 was conducted:

- BiModeBP;
- LTAGE;
- LocalBP;
- MultiperspectivePerceptron64KB;
- MultiperspectivePerceptron8KB;
- MultiperspectivePerceptronTAGE64KB;
- MultiperspectivePerceptronTAGE8KB;
- TAGE;
- TAGE_SC_L_64KB;
- TournamentBP.

The test results are presented in *Fig.* 3, which shows that BiModeBP and MultiperspectivePerceptronTAGE64KB perform poorly in most tests, although this is not always the case. It is worth noting that a detailed analysis of each test can provide valuable information about the BPU during the development of a new processor and contribute to improving its architecture. The presented benchmark and test results are available in the repository [21] for open analysis by anyone interested.

VI. Future work

The further development of the project includes plans to extend test generation, making it more targeted on BPU and dependent on test results for better generation of informative tests. Also there are plans to support Android devices, possibly using adb to expand the statistics. Testing results for a greater number of processors are also planned.

VII. Conclusion

A tool using the fuzzing method for comparing BPUs has been implemented. It generates tests that utilize BPUs and collects CPU counters about their operation for analysis. With this tool results were obtained for multiple architectures and a benchmark of generated tests was created. This benchmark contains tests that we consider relevant for comparing BPUs, as they show similar results for several BPUs in our testing. The benchmark is publicly available and can provide valuable insights for CPUs developers and researchers worldwide.

References

[1] K. Murakami, N. Irie, and S. Tomita, "Simp (single instruction stream/multiple instruction pipelining): A novel high-speed single-processor architecture," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 3, pp. 78–85, 1989.

[2] I.-J. Huang and A. M. Despain, "Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE, 1993, pp. 594–599.

[3] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th annual international symposium on computer architecture*, 1993, pp. 257–266.

[4] A. Youssif, N. Ismail, and F. Torkey, "Comparison of branch prediction schemes for superscalar processors iceec 2004," in *Electrical, Electronic and Computer Engineering, 2004. ICEEC'04. 2004 International Conference on*, 2004, pp. 257–260.

[5] S. A. I. Quadri and M. Z. Jahangir, "Design, implementation and performance comparison of different branch predictors on pipelined-cpu," in *2017 International Conference on Computer, Electrical & Communication Engineering (ICCECE)*. IEEE, 2017, pp. 1–7.

[6] Branch predictor: How many "if"s are too many? including x86 and m1 benchmarks! [Online]. Available: https://blog.cloudflare.com/branch-predictor

[7] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Halfhalf: Demystifying intel's directional branch predictors for fast, secure partitioned execution," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1220–1237.

[8] V. Uzelac, "Microbenchmarks and mechanisms for reverse engineering of modern branch predictor units," *A Masters Thesis submitted to the University of Alabama*, 2008.

[9] Control hazard analyzer. [Online]. Available: https://github.com/osogi/control-hazard-analyzer

[10] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[12] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: Cpu fuzzing via intricate program generation."

[13] afl-compiler-fuzzer. [Online]. Available: https://github.com/agroce/afl-compiler-fuzzer

[14] Coremark. [Online]. Available: https://www.eembc.org/coremark/
[15] Championship branch prediction 2004. [Online]. Available: https://jilp.org/cbp/
[16] Championship branch prediction 2016. [Online]. Available: https://jilp.org/cbp2016/
[17] Perf. [Online]. Available: https://perf.wiki.kernel.org/index.php
[18] Oprofile. [Online]. Available: https://oprofile.sourceforge.io/about/
[19] Cachegrind. [Online]. Available: https://valgrind.org/docs/manual/cg-manual.html
[20] Gem5. [Online]. Available: https://www.gem5.org/
[21] Collected benchmark. [Online]. Available: https://github.com/osogi/control-hazard-analyzer/tree/syrcose/tests-benchmark-example

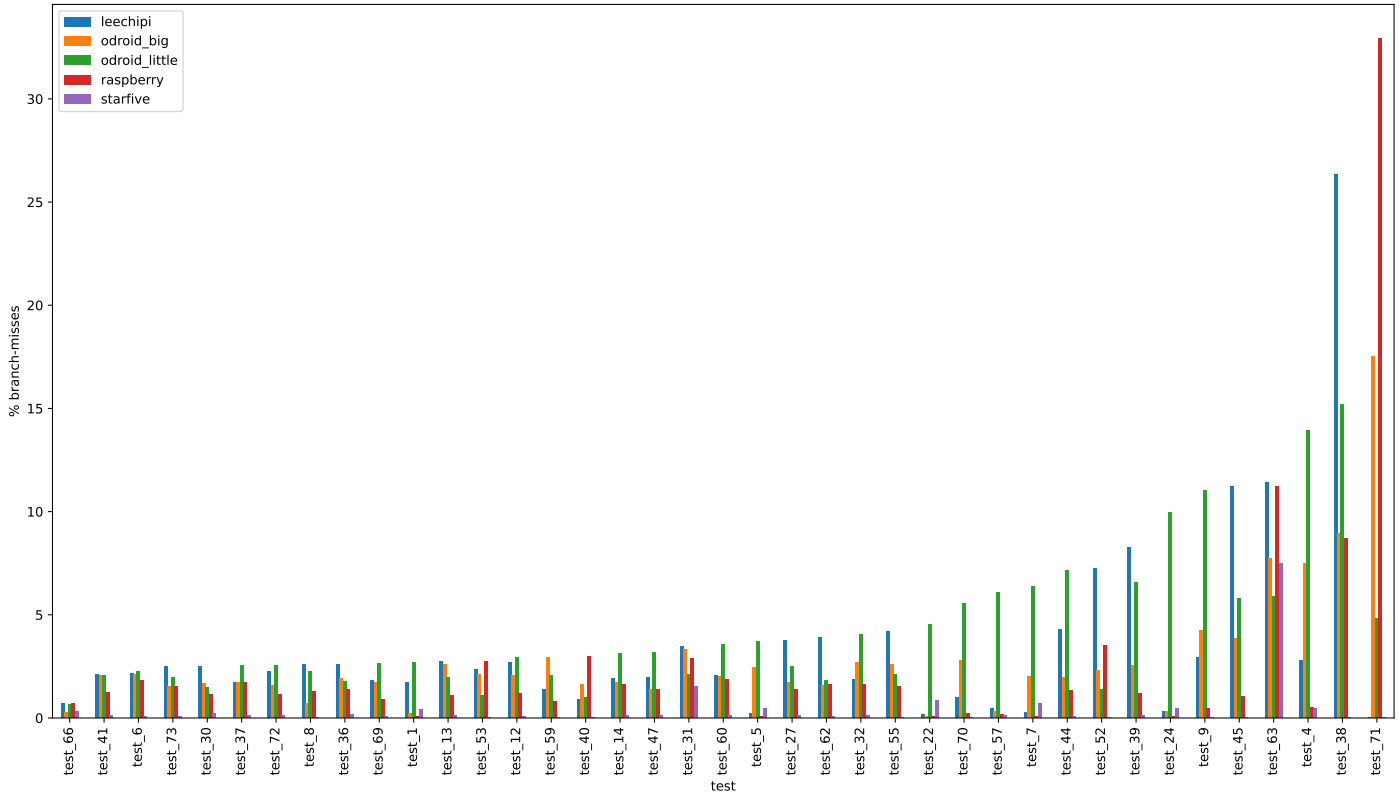

Fig. 1. Results for low-end

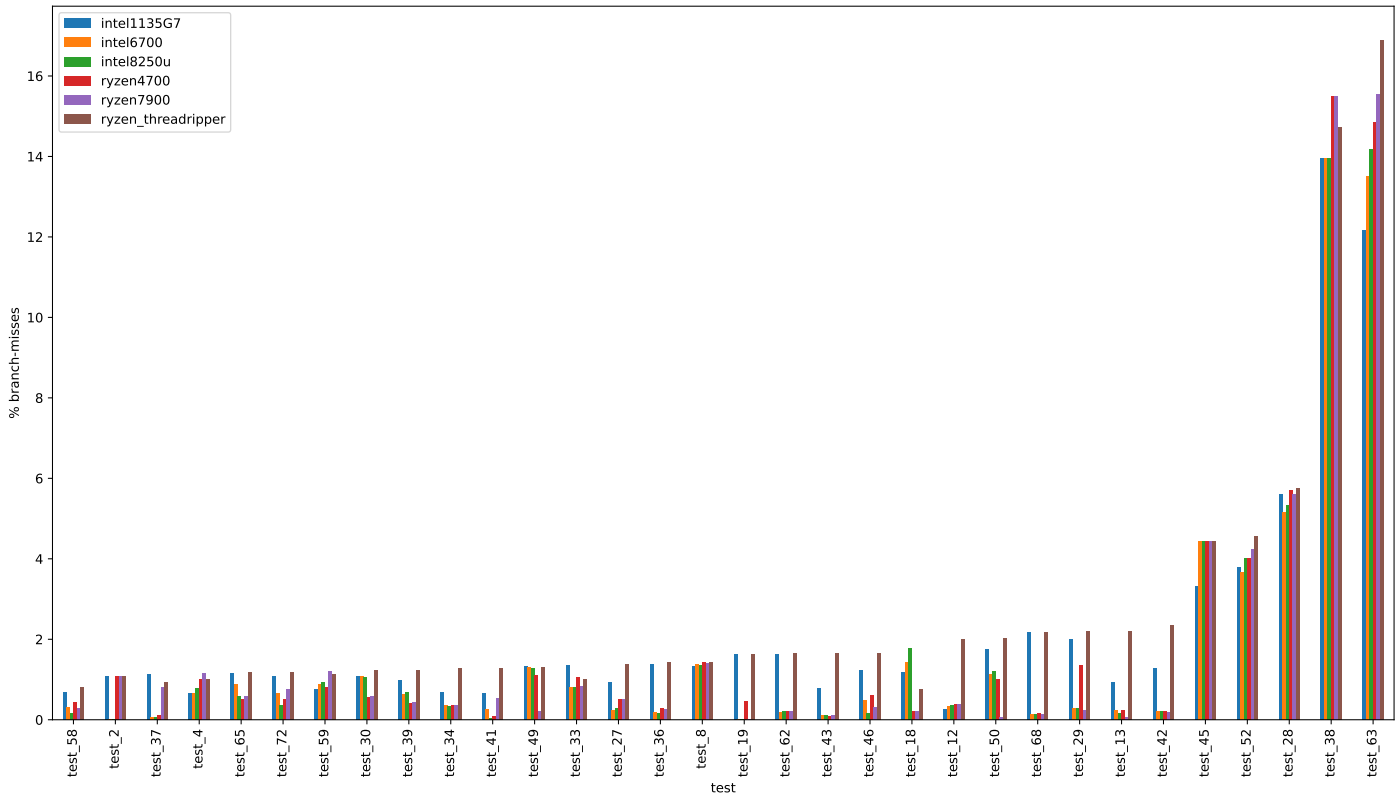

Fig. 2. Results for high-end

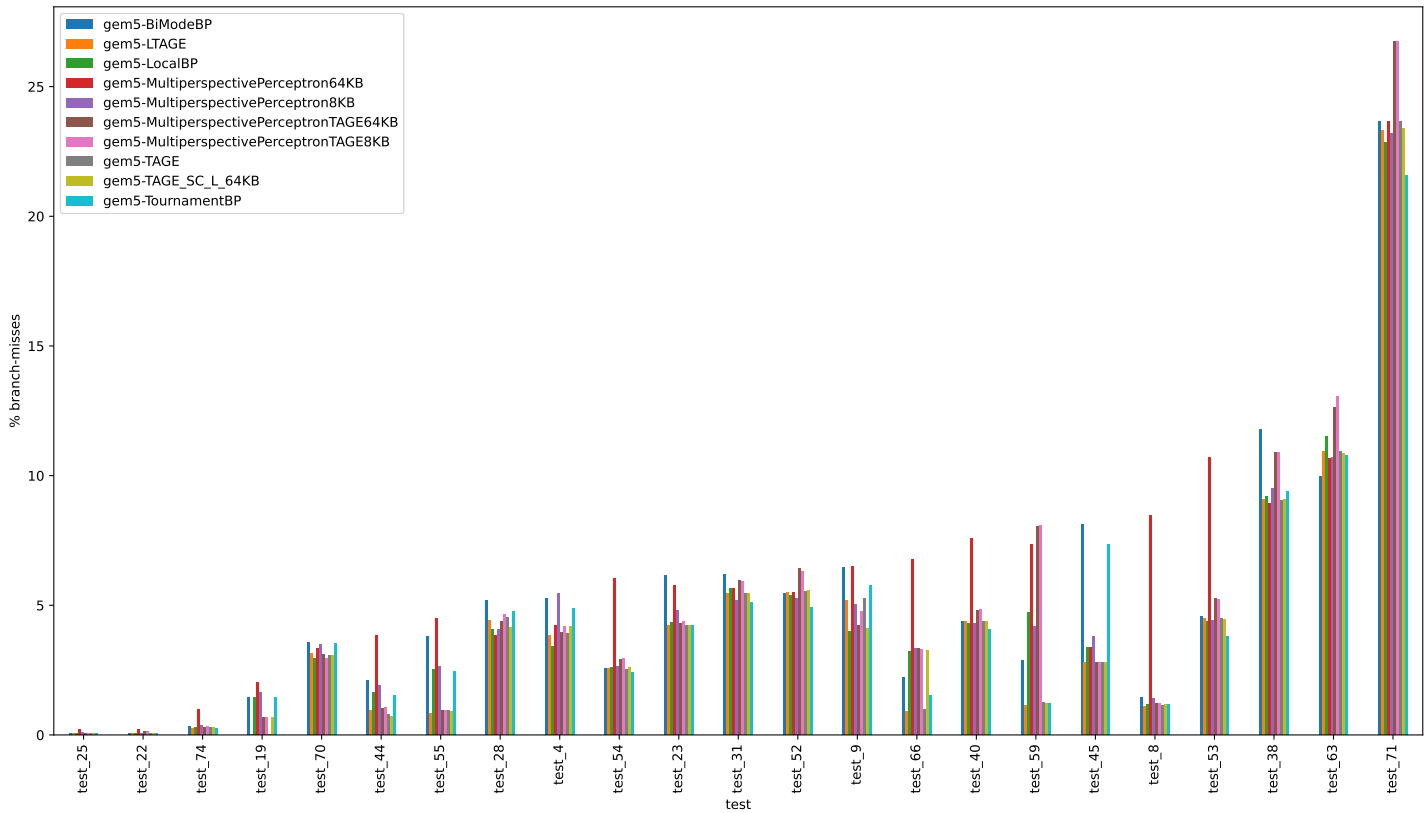

Fig. 3. Results for gem5