Санкт-Петербургский государственный университет

Кафедра Системного программирования

Группа 22.Б11-мм

Анализ BPU: генерация тестов и сбор статистики

Слинчук Данил Алексеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ст. преподаватель кафедры ИАС К. К. Смирнов

Консультант:
Инженер-исследователь, Лаборатория технологий программирования инфраструктурных решений СПбГУ, В. А. Кутуев

Санкт-Петербург
2024

Оглавление

Введение

BPU (branch prediction unit) – это критически важный компонент современных процессоров, отвечающий за предсказание результатов условных и безусловных переходов при исполнении кода. Предсказание результатов ветвлений необходимо для увеличения производительности процессоров, поскольку ветвления провоцируют сброс конвейера, что существенно снижает эффективность ЦПУ. BPU является частью микроархитектуры процессора, поэтому остаются неизвестными тонкости его реализации и факторы, влияющие на его эффективность в большинстве семейств процессоров компаний Intel [11] и AMD [1], поскольку реализация их архитектур закрыта. Выявление ситуаций, когда BPU в одном процессоре показывает аномально низкие результаты, по сравнению с BPU в распространенных современных CPU, может способствовать усовершенствованию самой архитектуры BPU путем тщательного изучения таких ситуаций [9, 10, 8].

Для нахождения сценариев, выявляющих разницу в эффективности различных BPU, был предложен метод, рассматривающий его, как «черный ящик». Суть этого метода заключается в многократном запуске различных блоков кода, чтобы найти блоки инструкций, показывающие слабые места BPU. Для реализации данного метода была начата работа по созданию инструмента, включающего в себя:

- генерацию тестов на языке C;
- сборку, запуск тестов и сбор данных со счетчиков процессора, содержащих информацию о работе BPU;
- подведение и обработка статистики по запущенным тестам с демонстрацией результатов на графике.

В данной работе были реализованы: генератор тестов на C; улучшения, связанные с презентабельностью результатов в последнем этапе инструмента; а также сбор данных с реальных процессоров и формирование бенчмарка. Данная работа выполняется совместно с работой

студента второго курса направления «Программная инженерия» Ефремова Алексея, в рамках которой была сделана часть инструмента по сбору статистики BPU.

1. Постановка задачи

Целью работы является расширение функциональности проекта [4]. Для выполнения цели были поставлены следующие задачи:

1. провести обзор существующих генераторов кода;
2. реализовать этап генерации тестов;
3. настроить фильтрацию, а также сортировку результатов;
4. повысить презентабельность графика, строящегося на последнем этапе инструмента;
5. сформировать бенчмарк, содержащий сгенерированные тесты, а также собрать результаты с процессоров различных архитектур для получения статистики.

2. Обзор

2.1. Сбор статистики BPU

В работе Ефремова Алексея, соучастника проекта, основной задачей является получение данных о функционировании BPU со счетчиков процессора, таких как: количество условных и безусловных переходов, число выполненных инструкций, информацию о работе BTB (branch prediction buffer) и т.д. При сравнении двух BPU решающим фактором является процент некорректных переходов, однако разработанный инструмент сохраняет все данные о работе, которые могут пригодиться для более тщательного изучения архитектуры и пройденного теста.

На данный момент используется два основных способа получения значений для сравнения BPU: с помощью системного вызова `perf_event_open` и симулятора Gem5 [6]. Каждый из них собирает разный набор данных, но в обоих случаях он содержит необходимую информацию.

2.2. Обзор сравнения BPU

Перед началом работы был проведен анализ того, как сравниваются BPU в современном в мире. Одной из показательных работ является пост в блоге The CloudFlare Blog [2], где проводилось исследование различных архитектур, таких как AMD EPYC 7642, AMD EPYC 7713, Intel Xeon Gold 6262, Apple Silicon M1. Исследование показало, что производительность BPU в основном определяется размером BTB (Branch Target Buffer), а не размером исполняемого кода. Для демонстрации влияния размера BTB на производительность измерялись задержки в ЦП для разных типов переходов. Хотя в этой работе основное внимание уделялось размеру BTB, ее методы и выводы могут быть применимы и к настоящему проекту.

2.3. Обзор аналогов генераторов

2.3.1. MicroTESK

MicroTESK [12] — это реконфигурируемый (перенацеливаемый и расширяемый) генератор тестовых программ (TPG), основанный на модели, для микропроцессоров и других программируемых устройств (такие инструменты также называют генераторами потока инструкций или ISG). Генератор настраивается с помощью спецификаций архитектуры системы команд (ISA) и конфигурационных файлов, которые описывают параметры подсистем микропроцессора (конвейера, памяти и других).

2.3.2. LLVM-snippy

LLVM-snippy [7] — это кроссплатформенный генератор случайного кода. Генерация может быть основана на модели или работать в неуправляемом режиме. LLVM-snippy поддерживает генерацию циклов, вызовов функций, шаблонов памяти и многого другого. Ключевое отличие этого генератора в том, что он основан на компиляторе LLVM и имеет отдельный интерфейс для подключения моделей. Поэтому состав, кодировку и семантику инструкций можно переиспользовать.

2.3.3. Csmith

Csmith [5] — это случайный генератор программ на языке C. Его основная цель — находить ошибки компилятора в случайных программах с использованием дифференциального тестирования. Конфигурация генератора происходит с помощью аргументов командной строки, а результатом генерации является файл с программой на языке C уже готовый к компиляции.

2.3.4. Выводы

MicroTEST и LLVM-snippy очень большие проекты, которые могут использоваться для большого спектра задач, однако они требуют тон-

кой конфигурации для получения желаемого результата. Также генерацию кода будет необходимо настраивать для каждой архитектуры отдельно, что существенно усложняет использование данных генераторов. Csmith наоборот довольно прост в использовании и конфигурации, однако этот инструмент создавался в первую очередь для фаззинга компиляторов, а также Csmith не получал обновлений в течение нескольких лет. С учетом всего выше сказанного было принято решение реализовать собственный генератор кода на C, так как в этой работе нужно учитывать специфику самих тестов. Также использование сторонних генераторов добавляет сложную зависимость в проект, и при дальнейшей разработке модификация такого решения может оказаться очень трудоемкой или вовсе невозможной.

3. Генерация кода

Конечным артефактом генерации был выбран файл программы на языке C, так как в таком случае его можно скомпилировать необходимым образом под любую архитектуру. Также такой формат нагляднее для дальнейшего анализа теста разработчикам BPU, чем текст на языке ассемблера или двоичный файл, так как сохраняет структуру условных переходов.

3.1. Генерация AST

Генерация кода начинается с рандомизированного построения AST (abstract syntax tree). Такое представление было выбрано, поскольку затем его довольно легко транслировать в код программы, а также можно точечно влиять на структуру порождаемого кода. На данный момент можно генерировать следующие синтаксические конструкции:

- условные выражения;
- циклы for и while;
- различные арифметические вычисления.

Генератор можно конфигурировать, сейчас доступны следующие параметры:

- максимальная вложенность условных выражений и циклов;
- количество блоков на каждом уровне;
- веса для каждого типа блоков, от которых будет зависеть шанс появления этого блока в AST;
- максимальные значения для генерируемых литералов.

Также можно контролировать генерацию новых переменных и их использование в различных блоках, есть возможность динамически вычислять каждый параметр генерации на основе каких-то данных. Элемент рандомизации присутствует при выборе следующего блока для

генерации, используемых переменных в блоке и генерируемых литералов и констант.

3.2. Трансляция в код программы

После построения AST транслируется в текст программы путем его обхода с помощью шаблона проектирования «посетитель» (visitor). На выходе получается текст программы на языке C, который можно использовать для тестирования.

4. Улучшение графиков с результатами анализа

Одной из задач работы является повышение презентабельности строящихся графиков для анализа тестирования. В предыдущей реализации был ряд проблем, затрудняющих быстрый анализ, а также влияющих на визуальное качество графиков 1:

- результаты никак не фильтровались, из-за чего могли появиться данные с отрицательным или большим 100 процентом неправильно предсказанных переходов;
- тесты никак не сортировались, из-за чего график был «размазанным»;
- отсутствовала легенда и подписи для каждой оси;
- график отражал только процент неверно предсказанных ветвлений, однако для анализа также важно общее количество ветвлений в тесте;
- некоторые тесты не завершались полностью за установленное время выполнения, однако график не отражал информацию об этом.

В результате работы эти недостатки были исправлены: добавлена фильтрация и сортировка результатов, легенда к графику, а также подсказка при наведении на тест, содержащая общее количество переходов в нем. Не полностью прошедшие тесты на графике выделялись более тусклым цветом. Пример графика после доработки изображен на рисунке 2.

Таким образом проделанные улучшения позволяют проводить более быстрый анализ большого количества тестов, что критически важно для тестирования с большим числом тестов.

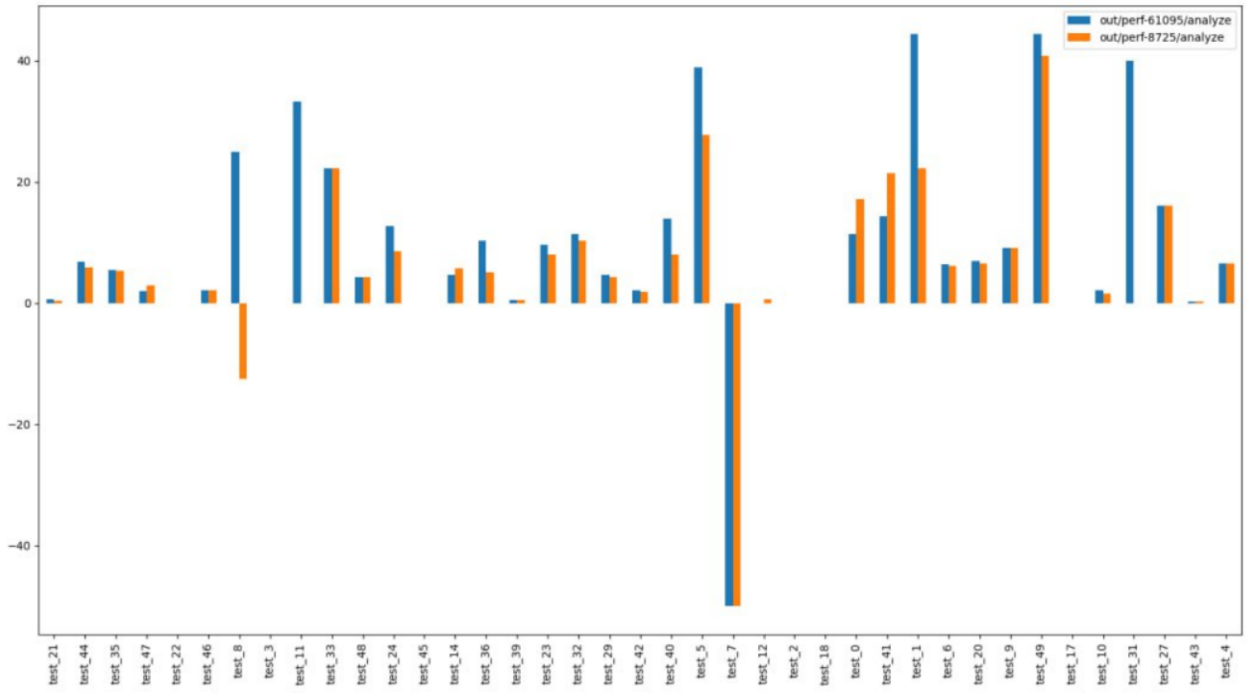

Рис. 1: Пример прошлого графика

5. Получение результатов

Одной из задач работы являлось получение статистики о работе BPU для процессоров разных архитектур. Для этого было необходимо сформировать набор тестов, по которым будет происходить сравнение, а также правильно сформировать тестовое окружение для каждого процессора.

5.1. Формирование бенчмарка

Для получения статистических данных о работе BPU в сравнении с другими необходим набор тестов (бенчмарк). Для этого была проведена селекция тестов: было выбрано два BPU разных типов, затем генерировалось около сотни тестов, и из них отбирались самые показательные. Основными критериями для отбора послужили такие метрики как процент неверно предсказанных переходов и общее количество ветвлений в тесте. В общей сложности было сгенерировано несколько сотен тестов, из которых 75 было сформировано в бенчмарк. Далее тесты из этого

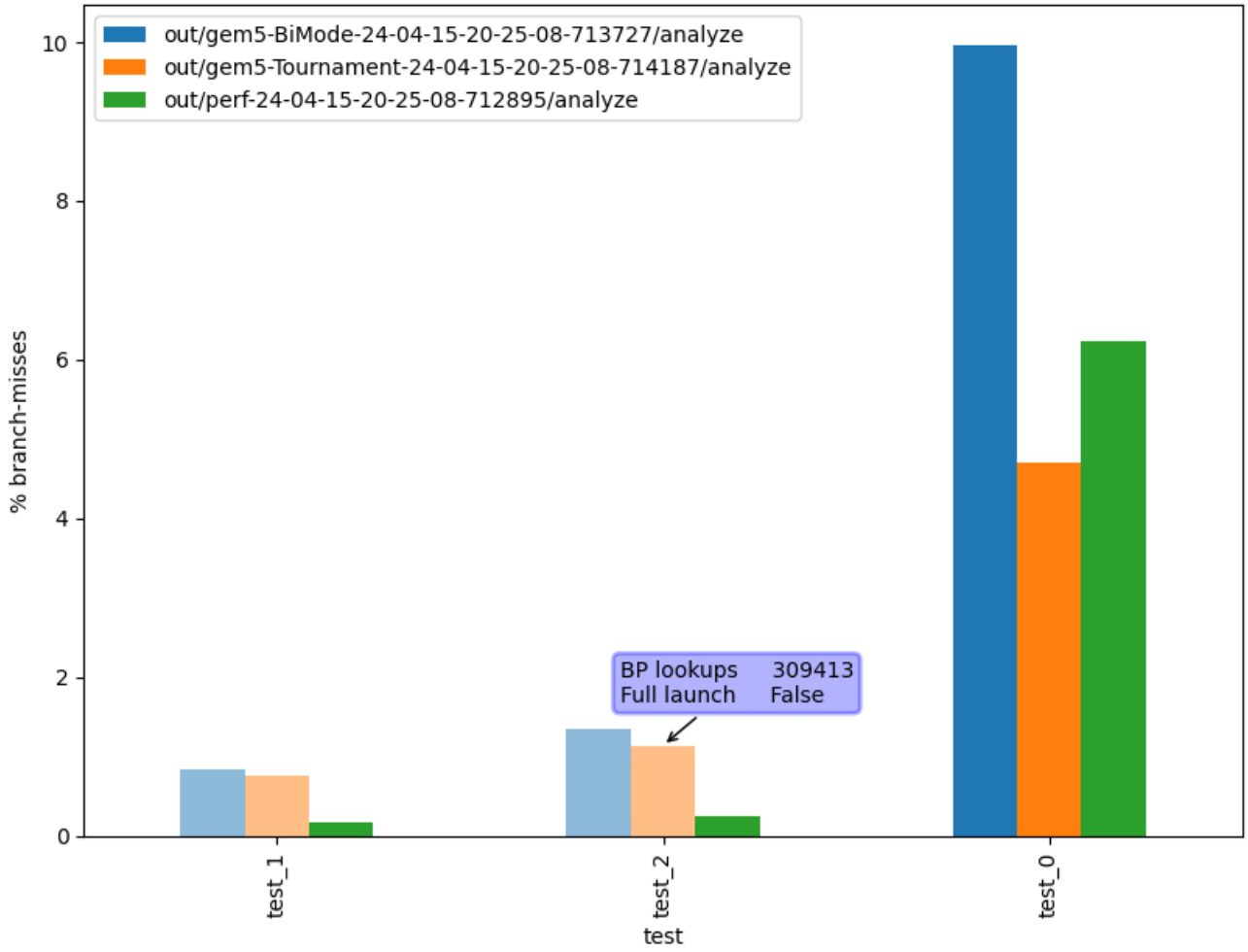

Рис. 2: Пример графика после исправлений

бенчмарка запускались на других процессорах для подведения статистики.

5.2. Подготовка тестового окружения

Перед проведением тестирования было необходимо правильно настроить тестируемые компьютеры для получения достоверных результатов, так как каждый процессор может содержать уникальные счетчики.

5.2.1. Микрокомпьютер ODROID-XU4

Дистрибутив, который изначально был на данном компьютере, содержал ядро Linux, не поддерживающее сбор данных с маломощных

процессоров. Если процесс запускался на маломощном ядре, то все счётчики по его завершении равнялись нулю. Поэтому был установлен другой дистрибутив с ядром, поддерживающим сбор счётчиков с обоих видов ядер. Также наш инструмент поддерживает запуск теста на конкретном ядре, что очень полезно, так как в процессоре ODROID-XU4 используется два типа ядер.

5.2.2. Sipeed Lichee RV-86 Panel

Изначально тесты так же планировалось запускать на одноплатном компьютере плате Lichee RV-86 Panel с процессором RISC-V. Для подключения к нему использовался UART с утилитой minicom, была настроена тестовая среда. Однако в ходе настройки выяснилось, что у рассматриваемого процессора всего одно ядро, на котором будет запущена ОС, ssh сервер и остальные приложения, из-за чего процесс с тестом будет часто вытесняться из контекста и его данные нельзя будет рассматривать как достоверные.

5.2.3. Raspberry Pi 4

Установить окружение для тестов для Raspberry было не сложно, но в ходе тестирования выяснилось, что процессор ARM Cortex-A72 установленный в этой модели не имеет счётчика для подсчёта общего числа ветвлений, а только неугаданных. Но было обнаружено, что утилита perf на Raspberry способна собирать число угаданных переходов, что обычно не поддерживают другие процессоры. После этого был изучен исходный код этой утилиты, а конкретно как она конфигурируется для разных процессоров, и был найден способ как в рамках разрабатываемого инструмента тоже считать подобные процессорозависимые события.

5.3. Результаты тестирования

Всего было протестировано 10 устройств с 3 архитектурами.

- 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-XU4: Samsung Exynos5 Octa ARM Cortex-A15 2GHz and Cortex-A7 1.3GHz.
- RISC-V:
  - VisionFive 2: StarFive RISC-V JH7110;
  - Sipeed LicheePi 4A: TH1520 RISC-V C910.

Для сравнения BPU устройства были разделены на две группы: low-end и high-end. В первую группу вошли все устройства с архитектурой ARM и RISC-V, во вторую - x86. Результаты по каждой группе представлены на графиках 3 и 4. Оценка точности метода измерения является проблематичной, поскольку она может сильно варьироваться от процессора к процессору. Таким образом, после 10 прогонов нашего эталонного теста со следующими параметрами: ”timeout”: 10, ”max_test_launches”: 50. Для процессоров AMD Ryzen 7 4700 и StarFive среднее значение стандартной ошибки для всех тестов составило 0,04% и 0,21% соответственно, а максимальное значение стандартной ошибки достигло 1,6% и 7,6% соответственно, что демонстрирует разницу в ошибках примерно в 5 раз. Это объясняется тем, что в разных процессорах имеются разные счетчики производительности BPU: может отсутствовать счетчик общего числа ветвлений, но вместо него будет счетчик

правильно предсказанных ветвлений, и, соответственно, у каждого счетчика будет своя собственная ошибка. Стоит отметить, что на графиках отражены только показательные тесты, то есть те, на которых соотношение производительности разных BPU было нестандартным. По оси X расположены сгенерированные тесты, а по оси Y процент промаха для конкретного BPU. Все тесты сортированы по максимальному значению этого процента. Рассмотрим подробнее каждый из графиков.

5.3.1. low-end

На графике 4 видно, что на тестах с самым маленьким процентом промахов, BPU показывают довольно схожие результаты. Однако дальше LicheePI и Odroid становятся аутсайдерами и показывают худшие результаты. Также присутствуют аномальные тесты, такие как 71 и 63, где Raspberry's BPU выдает большой процент.

5.3.2. high-end

Как можно заметить из графика 3, кол-во тестов, где процент промахов очень маленький, сильно увеличилось по сравнению с low-end. Также для тестов с процентами < 2 аутсайдерами являются Intel Core i5-1135G7 и AMD Ryzen Threadripper 2970WX. Стоит также отметить, что средний процент промахов по сравнению с low-end стал сильно меньше, максимальный процент составляет около 16, в то время как у low-end это значение порядка 35.

5.3.3. Gem5

Помимо сравнения BPU на реальных процессорах был проведен анализ различных BPU, сконфигурированных в gem5:

- BiModeBP;
- LTAGE;
- LocalBP;

- MultiperspectivePerceptron64KB;
- MultiperspectivePerceptron8KB;
- MultiperspectivePerceptronTAGE64KB;
- MultiperspectivePerceptronTAGE8KB;
- TAGE;
- TAGE_SC_L_64KB;
- TAGE_SC_L_8KB;
- TournamentBP.

Результаты тестов представлены на графике 5 из которого видно, что BiModeBP и MultiperspectivePerceptronTAGE64KB показывают наихудшие результаты в большинстве тестов, однако в ряде случаев это не так.

Стоит отметить, что детальный анализ каждого теста может дать ценную информацию о BPU во время разработки нового процессора и способствовать улучшению его архитектуры. Представленный бенчмарк, а также результаты тестов представлены в репозитории [3] для открытого анализа.

Заключение

В ходе данной работы были выполнены все задачи и получены следующие результаты:

1. проведён обзор генераторов кода: MicroTESK, LLVM-Snippy, Csmith;
2. реализован генератор кода на C;
3. доработан третий этап инструмента по построению графиков;
4. сформирован бенчмарк из 75 сгенерированных тестов, с его помощью проанализированы 10 реальных процессоров и 10 симуляций в Gem5 с различными BPU.

Результаты совместной работы с Алексеем Ефремовым были представлены на:

- конференции «Современные технологии в теории и практике программирования», опубликованы в сборнике входящем в РИНЦ[1];
- конференции SYRCoSE, сборник пока не опубликован.

Репозиторий инструмента на GitHub (пользователь: RoketFlame): https://github.com/osogi/control-hazard-analyzer (дата обращения: 2 июня 2024 г.).

Дальнейшее развитие инструмента

- Добавить генерацию с обратной связью, при которой результаты выполнения теста будут влиять на последующую генерацию. К примеру, мутационный фаззинг.
- Добавить генерацию других конструкций, например, вызова функций.
- Рассмотреть генерацию другого представления, например, графа потока управления или ассемблерного кода.

[1]Ссылка на сборник: https://www.elibrary.ru/item.asp?id=66237522 (дата обращения: 2 июня 2024 г.

Список литературы

[1] AMD. — официальный сайт. — URL: https://www.amd.com/ (дата обращения: 6 апреля 2024 г.).

[2] Branch predictor: How many ”if”s are too many? Including x86 and M1 benchmarks! — CloudFlare. — URL: https://blog.cloudflare.com/branch-predictor (дата обращения: 18 декабря 2023 г.).

[3] Collected benchmark. — URL: https://github.com/osogi/control-hazard-analyzer/tree/syrcose/tests-benchmark-example (дата обращения: 6 апреля 2024 г.).

[4] Control hazard analyzer. — GitHub. — URL: https://github.com/osogi/control-hazard-analyzer (дата обращения: 6 апреля 2024 г.).

[5] Csmith. — URL: https://github.com/csmith-project/csmith (дата обращения: 6 апреля 2024 г.).

[6] Gem5. — официальный сайт. — URL: https://www.gem5.org/ (дата обращения: 6 апреля 2024 г.).

[7] LLVM-snippy. — URL: https://github.com/syntacore/snippy (дата обращения: 6 апреля 2024 г.).

[8] Quadri Syed Ali Imran, Jahangir Mohd Ziauddin. Design, Implementation and Performance Comparison of Different Branch Predictors on Pipelined-CPU // 2017 International Conference on Computer, Electrical & Communication Engineering (ICCECE) / IEEE. — 2017. — P. 1–7.

[9] Yeh Tse-Yu, Patt Yale N. A comparison of dynamic branch predictors that use two levels of branch history // Proceedings of the 20th annual international symposium on computer architecture. — 1993. — P. 257–266.

[10] Youssif A, Ismail N, Torkey F. Comparison of branch prediction schemes for superscalar processors iceec 2004 // Electrical, Electronic and Computer Engineering, 2004. ICEEC'04. 2004 International Conference on. — 2004. — P. 257–260.

[11] intel. — официальный сайт. — URL: https://www.intel.com (дата обращения: 6 апреля 2024 г.).

[12] microTESK. — URL: http://www.microtesk.org/ (дата обращения: 6 апреля 2024 г.).

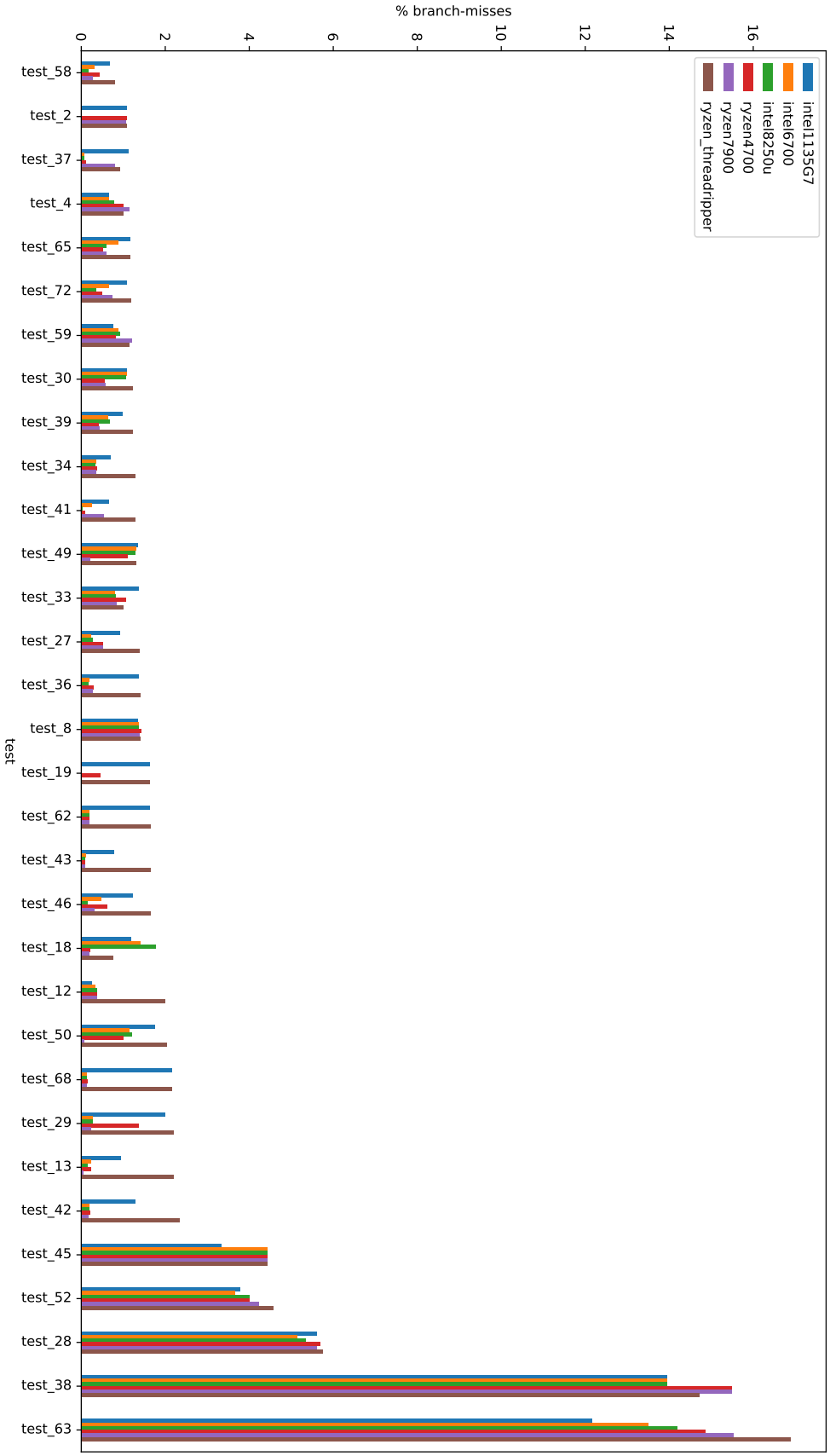

Рис. 3: Результаты для high-end

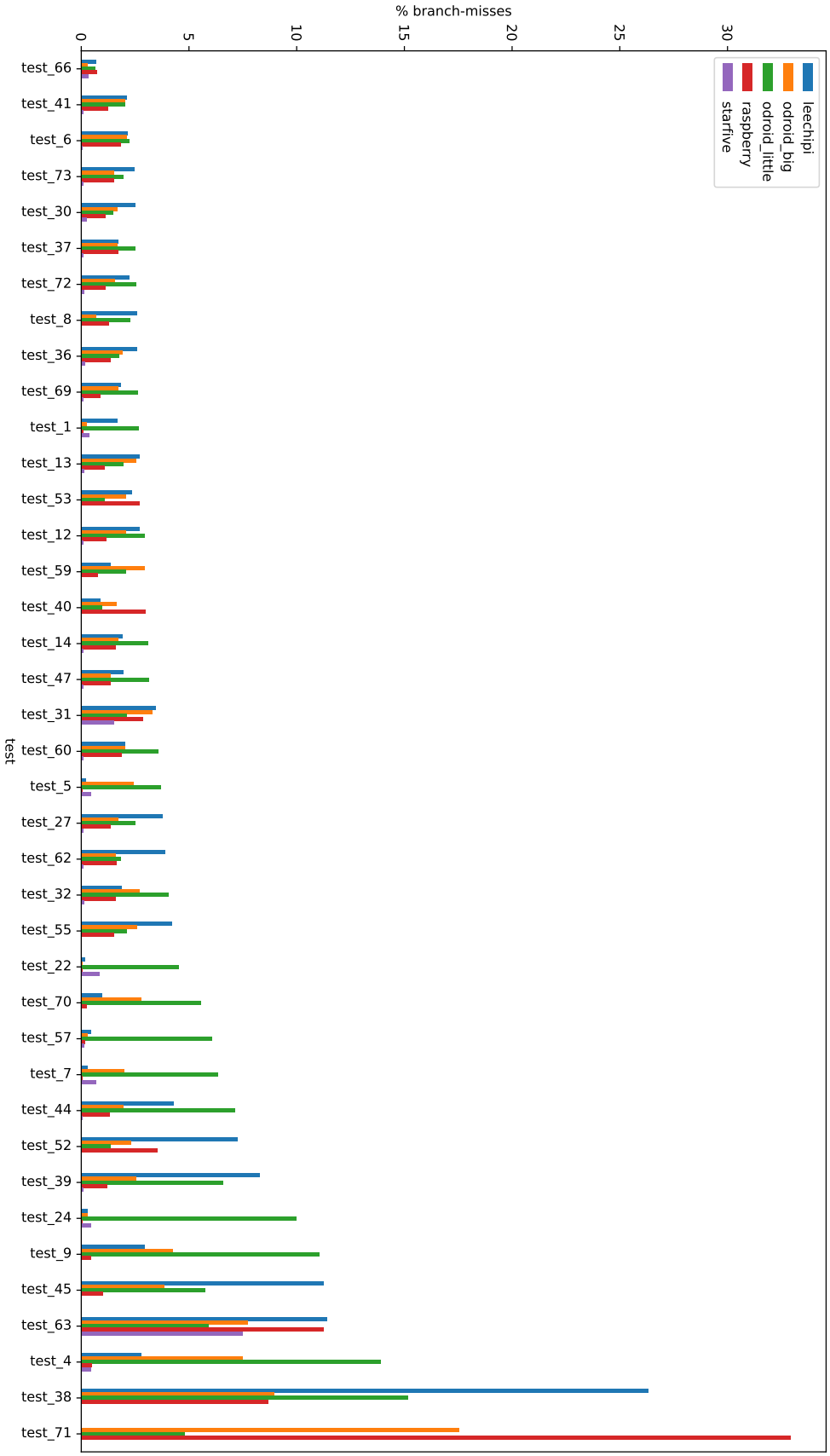

Рис. 4: Результаты для low-end

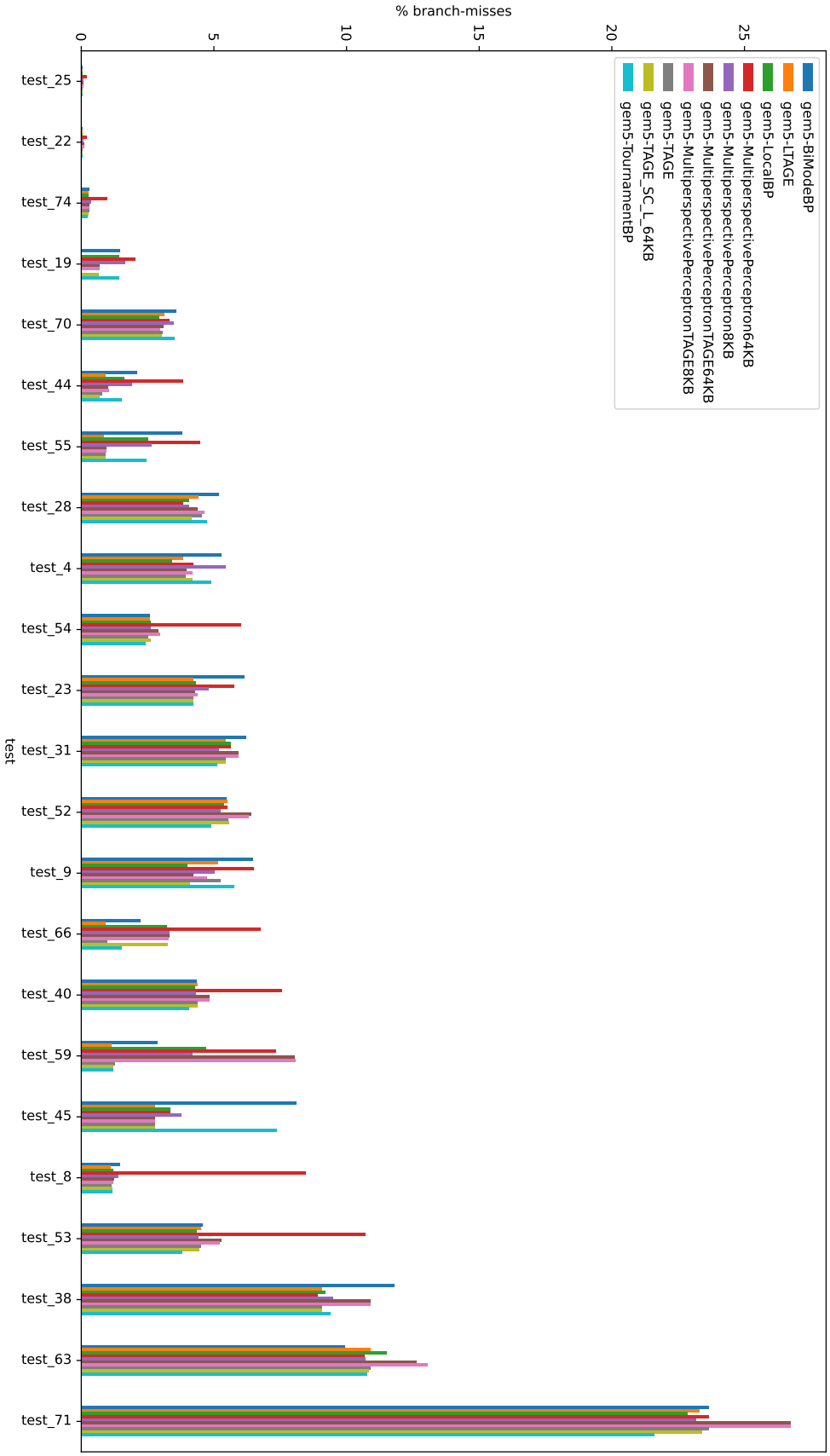

Рис. 5: Результаты для gem5