УДК 004.23

Исследование влияния глубины стека вызовов на точность предсказания Return Address Stack в процессорах AMD ZEN 4

К. А. Безуглый, Р. В. Мальчева
Донецкий национальный технический университет
кафедра компьютерной инженерии
E-mail: qoopdata@gmail.com

Аннотация:
Исследована зависимость между глубиной рекурсивных вызовов и числом промахов предсказателя адресов возврата (RAS) в микроархитектуре AMD Zen 4. Разработан микробенчмарк на языке Zig, использующий аппаратные счётчики производительности через системный вызов perf_event_open для измерения событий ветвлений, промахов предсказателя ветвлений и промахов RAS. Экспериментально определена граница переполнения RAS-буфера между глубиной вызовов 24 и 64 на процессоре AMD Ryzen 7 7840HS. Представлен анализ дизассемблированного кода и механизма взаимодействия с подсистемой perf ядра Linux.

Постановка проблемы

Повысить производительность работы процессора с последовательным принципом обработки машинных команд возможно лишь за счет повышения тактовой частоты. Однако рост частоты, на которой работает процессор, обладает ограничением, вызванным задержками распространения сигналов при прохождении через логические элементы схемы и внутренние линии соединений. Поэтому неограниченно повышать частоту процессора невозможно [1].

Альтернативным способом повышения производительности процессорного элемента является использование подхода *параллельной обработки машинных команд*. В рамках этого направления реализованы и активно применяются на практике принципы *конвейерной* и *суперконвейерной* обработки, *суперскалярная* и *векторная* организация вычислений [1].

Современные суперскалярные процессоры исполняют инструкции спекулятивно – до того, как станет известен фактический результат условного или безусловного перехода. Это позволяет конвейеру оставаться загруженным и достигать пропускной способности в несколько инструкций за такт.

Однако при неверном предсказании направления перехода конвейер должен быть очищен, а спекулятивно исполненные инструкции – отброшены. Стоимость такого сброса в микроархитектуре AMD Zen 4 составляет порядка 11-18 тактов в зависимости от глубины конвейера и типа ветвления [2].

Особую категорию ветвлений представляют инструкции возврата из подпрограмм (*ret*). В отличие от условных переходов, адрес назначения которых фиксирован в коде и может быть предсказан по истории переходов, адрес возврата зависит от того, откуда была вызвана подпрограмма, – он динамический. Для предсказания адресов возврата процессоры используют специализированную аппаратную структуру – Return Address Stack (RAS), которая представляет собой стек фиксированного размера. При исполнении инструкции *call* адрес следующей инструкции помещается в RAS, а при исполнении *ret* - извлекается.

Проблема возникает, когда глубина вложенных вызовов превышает размер RAS-буфера: старые адреса вытесняются, и при возврате из глубоко вложенных вызовов предсказатель не может восстановить корректный адрес. Это приводит к промахам предсказания и, как следствие, к сбросам конвейера, снижающим производительность. Размер RAS-буфера является деталью реализации конкретной микроархитектуры и обычно не документируется в публичных спецификациях.

Настоящая работа посвящена экспериментальному определению границы переполнения RAS-буфера в микроархитектуре AMD Zen 4 (процессор Ryzen 7 7840HS) путём замера показаний аппаратных счётчиков производительности при варьировании глубины рекурсивных вызовов.

Предсказатель ветвлений AMD Zen 4

Предсказатель ветвлений (Branch Prediction Unit, BPU) в микроархитектуре AMD Zen 4 представляет собой многоуровневую систему, работающую на начальных стадиях конвейера – ещё до декодирования инструкций [2]. Он отвечает за предсказание трёх аспектов каждого ветвления: будет ли переход выполнен (taken/not-taken), какой адрес является целью

перехода и какой адрес является адресом возврата.

Для предсказания адресов возврата используется Return Address Stack – аппаратный стек, функционирующий по принципу LIFO (Last In, First Out). При декодировании инструкции call предсказатель помещает в RAS адрес инструкции, следующей за call (адрес возврата). При декодировании инструкции ret верхний элемент RAS извлекается и используется как предсказанный адрес возврата. Если предсказание корректно — конвейер продолжает работу без штрафа. Если нет — происходит сброс конвейера. RAS-буфер имеет фиксированный размер. В публичных руководствах AMD для Zen 4 точный размер не указан, однако экспериментальные данные различных исследователей указывают на глубину порядка 32 записей [3].

При глубине вложенных вызовов, превышающей этот размер, наиболее ранние адреса возврата вытесняются из стека и теряются. Когда управление начинает возвращаться из глубоко вложенных вызовов и глубина развёртки стека достигает области вытесненных записей, RAS не содержит корректного адреса, что приводит к промаху. Такой промах классифицируется процессором как специфическое событие RAS miss и регистрируется аппаратным счётчиком PMC с кодом 0xC9 (событие *retired_indirect_branch_instructions_mispredicted*, подтип RAS) [4].

Важно различать два типа событий:

- BRANCH_MISSES (общие промахи предсказателя ветвлений) включают промахи по всем типам ветвлений: условным, косвенным и возвратам;

- RAS misses (событие 0xC9) учитывают исключительно промахи, связанные с некорректным предсказанием адреса возврата.

Соотношение этих метрик позволяет отделить влияние переполнения RAS от прочих причин промахов предсказателя.

Выбор языка программирования, инструментария

Для реализации микробенчмарка выбран язык Zig [5]. Zig – системный язык программирования, ориентированный на низкоуровневое программирование с прямым контролем над генерацией машинного кода. В контексте данной работы критичны несколько свойств. Во-первых, стандартная библиотека Zig предоставляет типизированные обёртки над системными вызовами Linux, включая *perf_event_open*, *ioctl* и *mlockall*, что позволяет взаимодействовать с подсистемой производительности ядра без использования библиотек-обёрток языка C. Во-вторых, Zig предоставляет атрибут вызова *.never_tail*, который запрещает компилятору применять оптимизацию хвостового рекурсивного вызова (*tail call optimization*). Это критически важно для данного бенчмарка: если компилятор преобразует рекурсию в цикл, инструкции call/ret исчезнут из машинного кода, и RAS не будет задействован – измерение потеряет смысл.

В-третьих, декларация указателя на функцию как **const volatile fn* предотвращает инлайнинг и другие межпроцедурные оптимизации, гарантируя, что каждый вызов процедуры *task* генерирует реальную инструкцию *call* на уровне машинного кода.

Реализация микробенчмарка

Архитектура бенчмарка. Программа принимает пять аргументов командной строки: номер ядра CPU для привязки (*used_core*), количество итераций измерения (*try_count*), две глубины рекурсии для сравнения (*depth_a, depth_b*) и количество последовательных вызовов за одну итерацию (*call_amount*).

Объект измерения. Центральным элементом бенчмарка является функция *task*, реализующая рекурсивный спуск заданной глубины (рис. 1).

```
noinline fn task(depth: usize) callconv(.c) void {
    if (depth > 1) @call(.never_tail, &task, .{depth - 1});
}
```

Рисунок 1 - Функция, реализующая рекурсивный спуск заданной глубины

На уровне машинного кода эта функция транслируется компилятором Zig в следующую последовательность инструкций через дизассемблирование из собранного ELF-файла (рис. 2).

```
000000000100f070 <main.task>:
 10121d0:   cmp    $0x1,%rdi          ; сравнить depth с 1
 10121d4:   jbe    10121e3            ; если depth <= 1, перейти к ret
 10121d6:   push   %rax               ; сохранить регистр (формирование кадра стека)
 10121d7:   dec    %rdi               ; depth = depth - 1
 10121da:   call   10121d0            ; рекурсивный вызов task(depth - 1)
 10121df:   add    $0x8,%rsp          ; восстановить стек
 10121e3:   ret                       ; возврат
```

Рисунок 2 -Набор инструкций функции после дизассемблирования

Каждый рекурсивный вызов генерирует пару *call/ret.* При глубине *depth* = N выполняется N - 1 инструкций *call* и столько же инструкций *ret*. Каждая инструкция *call* помещает адрес возврата в аппаратный RAS, а каждая *ret* извлекает его. Если N - 1 превышает глубину RAS-буфера, ранние адреса вытесняются и при соответствующих *ret* происходят промахи.

Атрибут *callconv(.c)* обеспечивает использование стандартного соглашения о вызовах System V AMD64 ABI, при котором аргумент *depth* передаётся через регистр *%rdi*. Атрибут *.never_tail* в *@call* запрещает *tail call optimization* – без него компилятор мог бы заменить *call* и *ret* на *jmp*, устранив пару инструкций и исключив RAS из цепочки предсказания.

Очередь задач

Для обеспечения статистической значимости функция task вызывается многократно – *call_amount* раз за одну итерацию измерения (рис. 3). Использование volatile в типе указателя предотвращает оптимизации компилятора, то есть вызовы одной и той же функции с одинаковыми аргументами (common subexpression elimination) или вынести вызов за пределы цикла (loop-invariant code motion).

Взаимодействие с подсистемой perf

В таблице 1 показаны три аппаратных счётчика производительности, используемых программой и настраиваемых через системный вызов *perf_event_open* [6].

```
const task_type = *const volatile fn (usize) callconv(.c) void;
const task_ptr: task_type = @ptrCast(&task);
```

Рисунок 3 – Код заполнения массива косвенными указателями

Таблица 1 – Аппаратные счётчики производительности

Счётчик	Конфигурация	Назначение
BRANCH_INSTRUCTIONS	стандартный	общее число инструкций ветвления
BRANCH_MISSES	стандартный	промахи предсказателя ветвлений
RAS misses	0xC9	промахи предсказания адресов возврата

Событие 0xC9 является событием, специфичным для микроархитектуры AMD Zen 4 [5]. Оно соответствует счётчику *retired_indirect_branch_instructions_mispredicted* с фильтрацией по подтипу RAS и не входит в стандартный набор событий, абстрагируемых ядром Linux.

Все три файловых дескриптора создаются с флагами *exclude_kernel* и *exclude_hv*, что исключает из подсчёта события, произошедшие в коде ядра операционной системы и гипервизора. Это гарантирует, что измеряются только события пользовательского пространства.

Протокол измерения

Измерение каждой глубины состоит из следующих этапов:

1. Прогрев. Перед началом измерений выполняется один полный проход по очереди задач без замера метрик. Это необходимо для прогрева кэшей инструкций (L1i, L2i), данных (L1d) и структур предсказателя ветвлений (BTB, PHT). Без прогрева первая итерация измерения содержала бы значительное количество обязательных промахов (compulsory misses), не связанных с поведением RAS.

2. Сброс счётчиков (PERF.EVENT_IOC.RESET). Перед каждой итерацией все три счётчика обнуляются через *ioctl*.

3. Включение счётчиков (PERF.EVENT_IOC.ENABLE). Счётчики начинают инкрементироваться при каждом соответствующем микроархитектурном событии.

4. Исполнение нагрузки. Все *call_amount* вызовов *task(depth)* выполняются последовательно.

5. Отключение счётчиков (PERF.EVENT_IOC.DISABLE). Счётчики замораживаются.

6. Чтение значений. Значения счётчиков считываются из файловых дескрипторов.

Порядок включения и отключения счётчиков выбран так, чтобы минимизировать влияние самих системных вызовов: при включении сначала активируются счётчики промахов, затем – ветвлений; при отключении – в обратном порядке. Это обеспечивает, что все промахи, возникшие в период измерения, гарантированно охвачены.

Блокировка адресного пространства

Перед началом измерений программа вызывает *mlockall* с флагами CURRENT и FUTURE (рис. 4). Этот системный вызов блокирует все текущие и будущие страницы процесса в физической памяти, предотвращая их вытеснение в swap. Без этого во время измерения может произойти page fault, вызывающий переключение в ядро и искажение результатов. В

сочетании с привязкой к конкретному ядру CPU (через taskset при запуске) и приоритетом реального времени (chrt -f 99) это минимизирует факторы, влияющие на воспроизводимость результатов.

```
var return_value = linux.mlockall
(.{ .CURRENT = true, .FUTURE = true });
```

Рисунок 4 – Код блокировки адресного пространства

Исследования и результаты

Измерения проводились на процессоре AMD Ryzen 7 7840HS (микроархитектура Zen 4, ядро Phoenix) под управлением Linux 6.19 [6]. Параметры запуска:

call_amount = 10000,

try_count = 10,

сравниваемые глубины: 24 и 64.

В таблице 2 приведены результаты измерений при глубине 24.

Таблица 2 – Результаты измерений при глубине 24

Итерация	Ветвления	Промахи	% промахов	RAS-промахи
1	730 004	4	0,00	0
2	730 005	4	0,00	0
3	730 004	4	0,00	0
4	730 004	4	0,00	0
5	730 004	4	0,00	0
6	730 006	45	0,01	35
7	730 004	4	0,00	0
8	730 004	4	0,00	0
9	730 004	4	0,00	0
10	730 004	4	0,00	0
Среднее	730 004,3	8,1	~0,00	3,5

При глубине 24 среднее количество RAS-промахов составляет 3,5 на 10 000 вызовов – менее 0,05 % от числа вызовов. Отдельные выбросы (итерация 6) объясняются внешними прерываниями (NMI, таймерные прерывания) и переключениями контекста, которые невозможно полностью устранить даже с приоритетом реального времени.

Принципиально, RAS-промахи при данной глубине практически отсутствуют – стек вызовов полностью помещается в RAS-буфер.

В таблице 3 приведены результаты измерений при глубине вызовов, равной 64.

При глубине 64 наблюдается качественно иная картина. Среднее количество RAS-промахов составляет 10 021,6 – практически ровно по одному промаху на каждый из 10 000 вызовов *task(64)*. Это означает, что в каждом вызове как минимум один адрес возврата вытесняется из RAS-буфера.

Таблица 3 – Результаты измерений при глубине 64

Итерация	Ветвления	Промахи	% промахов	RAS-промахи
1	1 930 004	10 008	0,52	10 000
2	1 930 006	10 098	0,52	10 085
3	1 930 006	10 033	0,52	10 015
4	1 930 004	10 004	0,52	10 000
5	1 930 006	10 065	0,52	10 050
6	1 930 004	10 004	0,52	10 000
7	1 930 006	10 021	0,52	10 008
8	1 930 004	10 004	0,52	10 000
9	1 930 006	10 012	0,52	10 000
10	1 930 006	10 068	0,52	10 058
Среднее	1 930 005,2	10 031,7	~0,52	10 021,6

Анализ соотношения метрик

При глубине 64 среднее количество общих промахов ветвлений (10 031,7) почти точно совпадает с количеством RAS-промахов (10 021,6). Разница составляет ~10 промахов – величину, сопоставимую с фоновым шумом при глубине 24. Это свидетельствует о том, что при глубине 64 практически все промахи предсказателя ветвлений обусловлены исключительно переполнением RAS, а не ошибками предсказания условных переходов или косвенных вызовов.

Число RAS-промахов на один вызов (~1,0) при глубине 64 согласуется с предполагаемой глубиной RAS-буфера ~32 записи: при 63 парах call/ret и буфере в 32 записи 31 адрес будет вытеснен, однако из-за механизмов спекулятивного восстановления (speculative RAS checkpointing), применяемых в Zen 4, не все вытесненные адреса приводят к реальным промахам – процессор способен частично компенсировать переполнение за счёт других структур предсказателя (например, BTB или indirect target array).

Определение границ переполнения

Программа автоматически определяет наличие переполнения RAS по порогу: если среднее количество RAS-промахов на один вызов

превышает 0,1 % (порог threshold = 0.001), считается, что переполнение детектировано. В данном эксперименте:

- глубина 24: 4,5 / 10 000 = 0,045 % < 0,1 % – переполнения нет;
- глубина 64: 10 021,6 / 10 000 = 100,2 % > 0,1 % – переполнение есть.

Вывод программы: *RAS overflow detected between depth 24 and 64*. Это означает, что граница переполнения RAS-буфера в данной микроархитектуре лежит в интервале от 24 до 64.

Однако для решения практических задач, таких как точное профилирование производительности или тонкая настройка параметров компиляции, подобная вилка является слишком широкой.

Для уточнения границы предлагается провести серию дополнительных замеров с промежуточными значениями глубины, например, 28, 32, 36 и так далее. Это позволит последовательно сузить диапазон неопределенности и определить точный размер аппаратного RAS-буфера с дискретностью до одной записи, что даст более полное представление о микроархитектурных особенностях исследуемого процессора.

Обеспечение достоверности результатов

Для минимизации влияния внешних факторов на результаты измерений применён комплекс мер:

1. Привязка к ядру CPU (taskset). Процесс бенчмарка закрепляется за конкретным физическим ядром, исключая миграцию между ядрами, которая привела бы к инвалидации структур предсказателя и кэшей.
2. Отключение SMT (Simultaneous Multithreading). Второй аппаратный поток на том же ядре деактивируется, устраняя конкуренцию за разделяемые ресурсы микроархитектуры — в том числе за RAS-буфер, который в Zen 4 разделяется между аппаратными потоками одного ядра.
3. Приоритет реального времени (chrt -f 99). Процессу назначается наивысший приоритет планировщика FIFO, что минимизирует вероятность вытеснения другим процессом во время измерения.
4. Режим производительности CPU *governor* (performance). Частота процессора фиксируется на максимальном значении, предотвращая динамическое масштабирование (DVFS), которое могло бы изменять латентность доступа к структурам предсказателя.
5. Блокировка памяти (*mlockall*). Все страницы процесса фиксируются в физической памяти, исключая page faults во время измерения.
6. Фильтрация событий ядра. Счётчики настроены с флагами *exclude_kernel u exclude_hv*, учитывая только события пользовательского пространства.
7. Фаза прогрева. Предварительный проход по нагрузке обеспечивает заполнение кэшей и обучение предсказателя ветвлений.

Практическая значимость

Знание размера RAS-буфера имеет практическое значение при оптимизации программного обеспечения. Рекурсивные алгоритмы с глубиной, превышающей размер RAS, будут систематически терять производительность на каждом вызове. Это актуально для:

- рекурсивных парсеров (JSON, XML, языки программирования);
- алгоритмов обхода деревьев (компиляторы, файловые системы);
- генераторов кода, использующих взаимную рекурсию (coroutine-подобные паттерны);
- систем с глубокими цепочками виртуальных вызовов (полиморфизм в ООП-языках).

Для таких случаев рекомендуется либо ограничивать глубину рекурсии размером RAS-буфера целевой микроархитектуры, либо трансформировать рекурсию в итерацию с явным стеком в куче.

Кроме того, представленная методика измерения может быть адаптирована для других микроархитектур (Intel, ARM) путём замены события 0xC9 на соответствующий аппаратный счётчик. Это позволяет систематически сравнивать глубину RAS-буфера между поколениями процессоров и архитектурами.

Выводы

Экспериментально подтверждено, что глубина стека вызовов непосредственно влияет на точность предсказания адресов возврата в микроархитектуре AMD Zen 4. При глубине рекурсии 24 промахи RAS практически отсутствуют (в среднем 4,5 на 10 000 вызовов), тогда как при глубине 64 каждый вызов гарантированно порождает как минимум один промах RAS (в среднем 10 021,6 на 10 000 вызовов). Граница переполнения RAS-буфера лежит в интервале 24–64, что согласуется с оценками размера RAS в ~32 записи для Zen 4.

Разработанный микробенчмарк на языке Zig демонстрирует преимущества прямого доступа к аппаратным счётчикам производительности через системный вызов perf_event_open без промежуточных абстракций. Атрибуты компилятора Zig (*.never_tail, volatile, noinline*) обеспечивают точный контроль над генерацией машинного кода, что критически важно для микроархитектурных бенчмарков, где

каждая лишняя или отсутствующая инструкция может исказить результаты.

Предложенная методика может быть расширена для определения точного размера RAS-буфера путём бинарного поиска между глубинами 24 и 64, а также адаптирована для исследования других микроархитектурных структур – BTB (Branch Target Buffer), PHT (Pattern History Table) и их взаимодействия с RAS при спекулятивном исполнении.

Литература

1. Орлов, С. А. Организация ЭВМ и систем : учебник для вузов. 4-е изд. Фундаментальный курс по архитектуре и структуре современных компьютерных средств / С. А. Орлов. – СПб. – М. – Минск : «Питер», 2024. – 682 с.

2. AMD. Software Optimization Guide for the AMD Zen 4 Microarchitecture. – Publication No. 57647. – Rev. 1.00. – 2023.

3. Fog A. The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers. – Copenhagen University College of Engineering, 2024. [Electronic resource] – URL: https://www.agner.org/optimize/microarchitecture.pdf

4. AMD. Open-Source Register Reference for AMD Family 19h Processors. – Event 0xC9. – [Electronic resource]. - URL: https://github.com/torvalds/linux/blob/v6.17-rc7/tools/perf/pmu-events/arch/x86/amdzen4/branch.json

5. Zig Programming Language [Electronic resource]. – URL: https://ziglang.org/

6. Linux Kernel Documentation. perf_event_open(2) – Linux manual page [Electronic resource]. – URL: https://man7.org/linux/man-pages/man2/perf_event_open.2.html

7. Исходный код проекта ras-influence. – URL: https://github.com/fidelicura/uarch/ras-influence

Безуглый К. А., Мальчева Р. В. Исследование влияния глубины стека вызовов на точность предсказания Return Address Stack в процессорах AMD ZEN 4. *Исследована зависимость между глубиной рекурсивных вызовов и числом промахов предсказателя адресов возврата (RAS) в микроархитектуре AMD Zen 4. Разработан микробенчмарк на языке Zig, использующий аппаратные счётчики производительности через системный вызов perf_event_open для измерения событий ветвлений, промахов предсказателя ветвлений и промахов RAS. Экспериментально определена граница переполнения RAS-буфера между глубиной вызовов 24 и 64 на процессоре AMD Ryzen 7 7840HS. Представлен анализ дизассемблированного кода и механизма взаимодействия с подсистемой perf ядра Linux.*

Ключевые слова: *микроархитектура, предсказатель ветвлений, Return Address Stack, AMD Zen 4, аппаратные счётчики производительности, perf_event_open, Zig.*

Bezuglyi K. A., Malcheva R. V. Investigation of the influence of call stack depth on Return Address Stack prediction accuracy in AMD Zen 4 processors. *The dependence between recursive call depth and the number of Return Address Stack (RAS) prediction misses in the AMD Zen 4 microarchitecture is investigated. A microbenchmark in the Zig language was developed, using hardware performance counters via the perf_event_open system call to measure branch events, branch prediction misses, and RAS misses. The RAS buffer overflow boundary between call depths of 24 and 64 on an AMD Ryzen 7 7840HS processor was experimentally determined. An analysis of the disassembled code and the mechanism of interaction with the Linux kernel perf subsystem is presented.*

Keywords: *microarchitecture, branch predictor, Return Address Stack, AMD Zen 4, hardware performance counters, perf_event_open, Zig.*

Статья поступила в редакцию 21.09.2025
Рекомендована к публикации профессором Зори С. А.