Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Ножкин Илья Игоревич

Разработка профайлера для процессоров ARC

Выпускная квалификационная работа бакалавра

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
Ведущий инженер-программист группы программных инструментов разработки ООО "Синопсис СПб"
Краснуха Т. В.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administraction of Information Systems

System Programming

Ilya Nozhkin

Development of a profiler for ARC processors

Bachelor's Thesis

Scientific supervisor:
Professor Andrey Terekhov

Reviewer:
Software Engineer, Sr II at Synopsys SPb LLC
Tatyana Krasnukha

Saint-Petersburg
2020

Оглавление

Введение

Оптимизация программного обеспечения часто является одним из необходимых этапов процесса разработки. Особенно сильно эта необходимость проявляется при работе со встраиваемыми системами или при программировании вспомогательных модулей аппаратного обеспечения, поскольку в данных случаях имеет значение не только общая производительность и скорость выполнения алгоритма, но также потребление энергии и возможность удовлетворить требования используемых протоколов взаимодействия с другими компонентами.

Вместе с тем, анализ выполнения программы, запущенной на отдельном чипе, а не на компьютере разработчика, привносит ряд дополнительных трудностей. Во-первых, такие системы обладают невысокой производительностью и небольшим по сравнению с рабочими станциями объемом памяти, вследствие чего встраивание профилирующих инструментов непосредственно в код программы может привести к существенным погрешностям в измерениях и невозможности сохранять данные анализа на устройстве. Во-вторых, даже при работе с удаленным анализатором, запущенным на компьютере разработчика, возникает проблема в коммуникации между двумя системами по каналу связи, заведомо более медленному, чем процессоры этих систем, а также необходимость дополнительных затрат на коммуникацию и синхронизацию данных между системами, что также может влиять на результаты анализа.

Однако же, именно при анализе встраиваемых систем с жесткими ограничениями наиболее важна точность измерений и сохранение характеристик выполнения кода, как можно сильнее приближенных к таковым при реальной работе.

1. Постановка задачи

Целью данной работы является реализация профайлера, позволяющего проводить анализ производительности при выполнении программ на процессорах ARC.

Основные задачи:

- изучить существующие методы разработки профайлеров и рассмотреть средства измерения производительности, предоставляемые процессорами ARC;
- обозначить необходимые пользователю сценарии использования профайлера и разработать алгоритмы их выполнения;
- спроектировать и реализовать систему, предоставляющую возможность выполнения обозначенных сценариев;
- протестировать полученное решение и оценить точность измерений.

2. Обзор

Так как профилирование — это не только часто необходимая, но также и хорошо изученная часть процесса разработки ПО, то существует набор известных техник, применимых в тех или иных ситуациях. Рассмотрим некоторые из них, а также их преимущества и недостатки в случае анализа ПО для встраиваемых систем.

2.1. Существующие техники профилирования

Один из наиболее распространенных методов называется инструментирующим профайлингом (Instrumentation-based profiling). Классическим примером утилиты, реализующей данный подход, является gprof [2]. Данный метод заключается во внедрении в код программы дополнительных действий, направленных на сбор интересующих данных средствами того же процессора, на котором запускается анализируемая программа. Например, чтобы измерить время выполнения какой-либо функции достаточно сохранить значение системного счетчика до ее вызова и далее вычесть его из значения счетчика после выхода из нее.

Несмотря на внешнюю простоту этого подхода, он обладает некоторыми сложностями и недостатками.

Во-первых, внедрение дополнительных действий в код программы — это отдельная нетривиальная задача, автоматическое решение которой должно быть либо поддержано со стороны компилятора, либо сведено к изменению исходного или результирующего машинного кода.

Во-вторых, такой подход неизбежно приводит к искажению результатов измерений, поскольку выполнение дополнительных действий также занимает некоторое время. Кроме того, поток выполнения на современных процессорах имеет крайне нетривиальную структуру и является очень чувствительным к любым изменениям кода. Из-за этого, внедрение измеряющего кода может неявно изменить и время выполнения последующих инструкций. Естественный пример — изменение содержимого кэша при записи результатов измерений в память.

В-третьих, результаты замеров должны быть куда-нибудь записаны.

Здесь также есть несколько вариантов. Первый — запись необработанных результатов для каждого измеренного отрезка, например, в формате: нижняя граница, верхняя граница, затраченное время. Этот вариант является наиболее быстрым, однако приводит к потреблению большого количества памяти, что особенно нежелательно в случае встраиваемых систем. Второй вариант — статистическая обработка результатов непосредственно по завершении замера. Например, по окончании измерения времени выполнения функции производится поиск записи, соответствующей этой функции, в некоторой ассоциативной таблице и далее результат прибавляется к уже существующему значению. Данный вариант потребляет меньше памяти, однако сильнее искажает время выполнения. Третий вариант — отправка результатов измерений по внешнему каналу. В данном случае все сильно зависит от пропускной способности этого канала и затрат на передачу данных. Например, при использовании первого подхода для накопления буфера, отдельного потока для отправки содержимого этого буфера и достаточно быстрого канала, этот подход может использоваться для получения сравнительно точных результатов.

Второй распространенный подход к профилированию — это сэмплирующий (статистический) профайлинг (sampling-based profiling, statistical profiling). Одним из примеров утилиты, реализующий данный подход является Intel VTune. Описание реализации представлено на странице [5]. Он заключается в периодическом извлечении значения указателя текущей инструкции (instruction pointer) и проверке его попадания в один из предопределенных диапазонов. Например, перед выполнением программы извлекаются диапазоны, соответствующие каждой функции. Далее, строится гистограмма количества попаданий в различные функции, что позволяет приблизительно оценить наиболее затратные функции.

Данный подход также имеет ряд существенных недостатков. Во-первых, чтение позиции текущей инструкции может быть сопряжено с вмешательством в поток управления программы вплоть до полной ее остановки. Это особенно заметно при профилировании кода, запу-

щенного на удаленном чипе. В этом случае попытка чтения регистра приводит к дорогостоящей цепочке обмена сообщениями по отладочному каналу и влияет на внутреннее состояние процессора. Во-вторых, этот подход не сообщает никакой информации о реальном времени выполнения участков. Напротив, он лишь утверждает, что каждый участок занимает соответствующую часть от общего времени с некоторой вероятностью.

Более того, некоторые маленькие, но часто вызываемые функции могут быть и вовсе проигнорированы при данном подходе. Этот случай в полной мере продемонстрирован в работе [8] Одерова Р. С. в тесте "Пила".

Далее рассмотрим техники, специфичные для низкоуровневого профайлинга.

2.2. Профилирование с использованием встроенных в оборудование средств

Достаточно подробный обзор такого рода методов приведен в работе [6]. В частности, наиболее многообещающими выделены два подхода.

Первый — измерение производительности посредством добавления в схему процессора дополнительных отслеживающих модулей, подсоединяющихся к интересующим компонентам и загрузка полученной схемы процессора вместе с дополнительными модулями в программируемую логическую интегральную схему (FPGA Based Profiling). Например, можно добавить счетчики, включающиеся при выполнении инструкции на одном адресе и выключающиеся при выполнении инструкции на другом, что позволит замерить время выполнения этого участка целиком и получить нечто похожее на профайлинг с инструментированием кода, но без дополнительных потерь производительности и изменения программы.

Главным преимуществом данного подхода является отсутствие какого-либо вторжения в процесс выполнения кода и тем самым обеспечение наиболее точных результатов измерений, а также возможность встро-

ить сколь угодно сложное и функциональное расширение (разумеется, до ограничений по объему ПЛИС). Наиболее же существенный недостаток состоит в необходимости проектирования индивидуального измеряющего модуля для каждой модели процессора и в некоторых случаях даже для каждой исследуемой программы. Кроме того, такой подход не дает возможности оценить производительность на реальном процессоре.

Второй — использование заранее спроектированных и зашитых в процессор счетчиков различных происходящих событий (Performance Counters, счетчики производительности), начиная с выполнения одной инструкции и заканчивая высокоуровневыми событиями, такими как, например, промахи при обращении к кэшу. (Hardware Counters Based Profiling). Этот подход соединяет в себе как точность измерений, так и возможность настройки счетчиков прямо во время работы под каждую проверяемую программу и интересующие события. Кроме того, такие счетчики часто содержатся и в уже использующемся готовом процессоре, что позволяет исследовать поведение программы в реальных условиях. Главный же минус применения счетчиков заключается, во-первых, в их ограниченном количестве, а во-вторых, в их относительно слабой конфигурируемости по сравнению с перепрограммированием ПЛИС или со внедрением измеряющего кода в исследуемую программу.

2.3. Ядра ARC

ARC [12] являются широко конфигурируемыми лицензируемыми ядрами, разрабатывающимися компанией Synopsys. Поддерживают удаленную отладку с использованием специального интерфейса. Имеют возможность включения в структуру нескольких счетчиков производительности, настраиваемых на подсчет как непосредственно количества циклов, так и событий, связанных с работой внутренних подсистем. Также могут быть сконфигурированы на работу только в пределах заданного диапазона адресов.

2.4. Существующие инструменты для профилирования на ARC

Компания Synopsys предоставляет набор инструментов разработчика, которые могут быть использованы для исследования и оптимизации производительности пользовательских приложений. Каждый из них имеет свои преимущества, недостатки и область применимости.

2.4.1. xCAM

xCAM [10] — это инструмент для создания точных программных моделей процессора, действующих на уровне логических вентилей. Главное преимущество таких моделей — возможность исследовать производительность с абсолютной точностью относительно циклов процессора. Однако скорость выполнения модели на несколько порядков ниже, чем тактовая частота готового процессора, что создает дополнительные неудобства пользователю при профилировании. Кроме того, в случае взаимодействия программы с другими компонентами системы, требуется также создать и их модель.

2.4.2. nSIM

nSIM [9] — программный симулятор процессора, действующий на уровне инструкций. Имеет существенно более высокую скорость работы по сравнению с xCAM, однако не гарантирует точную симуляцию относительно количества циклов, затраченных на выполнение инструкций. Может использоваться для поиска высокозатратных участков программы, но не подходит для получения точных результатов. Кроме того, как и в случае с xCAM, взаимодействие со сторонним оборудованием требует создания симулирующих его компонент.

2.4.3. RTT

RTT [11] (Real-Time Tracing) — это возможность процессора генерировать большое количество информации о процессе выполнения про-

граммы в реальном времени с помощью специального модуля, включаемого в схему. Данный инструмент не позволяет напрямую получить информацию о циклах, затраченных процессором на выполнение того или иного участка, однако сообщает о различных событиях, таких как чтение из памяти или запись в нее, взаимодействия с регистрами, а также предоставляет полную трассу инструкций. Инструмент объединяет в себе такие преимущества, как отсутствие вмешательства в поток выполнения программы и выполнение анализа на полной скорости процессора. Главным недостатком является необходимость использования специального оборудования с быстрым каналом связи, позволяющим получать большой объем информации. Такое оборудование является достаточно дорогостоящим и может отсутствовать у пользователей.

2.5. Используемые инструменты

Поскольку процессоры ARC поддерживают взаимодействие с инструментами, запущенными вне самого чипа, посредством отдельного канала связи и специального протокола, предполагается реализовать профайлер в виде программы, запущенной на внешнем устройстве и не задействующей ресурсы анализируемой системы. Это позволит избежать лишнего влияния на точность измерений. Кроме того, при профилировании будет необходима информация о структуре исследуемой программы. В частности, расположение функций и граф их вызовов.

Для удовлетворения этих требований подходит существующий отладчик для процессоров ARC на основе LLDB [13]. Он запускается на независимой от чипа машине и предоставляет, в частности, такие базовые возможности, как запуск кода на чипе, чтение и запись регистров и участков памяти и извлечение отладочной информации из исполняемого файла, загруженного в память.

При этом, отладчик может быть настроен таким образом, чтобы минимизировать влияние на работу процессора при его остановке. В остальное время, пока программа исполняется процессором ARC, отладчик не вносит абсолютно никаких изменений в работу ядра.

LLDB предоставляет публичный API, охватывающий большую часть возможностей отладчика от добавления команд во встроенный интерпретатор до перехвата и автоматической обработки возникающих событий. API представлен на C++, а также существует полностью соответствующий аналог для Python, позволяющий подключать расширения динамически.

3. Сценарии и алгоритмы анализа

В профайлере реализуются несколько основных сценариев исследования производительности программы. Рассмотрим подробнее каждый из них и метод их реализации.

3.1. Ручной замер одного прохода участка кода

Данный сценарий является самым примитивным и реализуется следующим образом. Пользователь останавливается перед началом интересующего участка и конфигурирует посредством низкоуровневых компонентов профайлера счетчики производительности на подсчет интересующих событий. Далее, пользователь продолжает выполнение до конца участка и считывает накопленные данные. Со стороны профайлера в этом случае нужна только прямая обертка взаимодействия со счетчиками.

3.2. Автоматический замер многократного выполнения участка кода

В этом сценарии важно различать 2 возможных понятия термина "участок кода". В одном случае это только некоторый диапазон адресов в коде программы. В другом случае это код, ограниченный двумя адресами, но включающий также и вызовы подпрограмм, выходящие за пределы этих адресов.

В первом случае пользователь конфигурирует счетчики на активацию в заданном диапазоне и выполняет программу до накопления нужного количества данных.

Во втором случае пользователь сообщает профайлеру две точки и параметры счетчиков. Далее, профайлеру необходимо включить счетчики в первой точке и отключить по достижении второй. Это можно сделать несколькими способами. В данной работе предложен подход на основе динамической инструментации кода программы. То есть, профайлер внедряет код, включающий и отключающий счетчики средства-

ми инструкций процессора вместо остановки выполнения программы и конфигурирования счетчиков посредством отладочного интерфейса.

Это позволяет добиться минимального вмешательства в выполнение программы и тем самым обеспечить сравнительно высокую точность.

3.3. Точный сбор статистики потребления ресурсов функциями для однопроходных программ

Назовем однопроходной программу, выполняющую обработку некоторого пакета данных и завершающуюся по окончании обработки. Таким программам противопоставлены те, которые работают долгое время в цикле и реализуют обработку последовательно поступающих пакетов, например, осуществляют взаимодействие по некоторому протоколу связи. Назовем такие программы циклическими.

В случае однопроходных программ имеет место важное свойство — их поток выполнения полностью детерминируется содержанием обрабатываемого пакета данных. Таким образом, возможно запускать программу несколько раз с одними и теми же данными и замерять время работы ограниченного подмножества составляющих их функций. Объединив результаты по всем подмножествам, будет получена полная гистограмма распределения ресурсов по функциям.

Таким образом, сценарий такого анализа состоит в следующем.

- Пользователь предоставляет профайлеру машинный код программы с отладочной информацией и пакет данных для запуска.
- Профайлер, используя средства LLDB, извлекает из отладочной информации полный список функций исходной программы и диапазоны их адресов.
- Список функций разбивается на подмножества, каждое из которых содержит количество функций, равное количеству имеющихся счетчиков.

- Для каждого подмножества профайлер конфигурирует счетчики на счет в диапазоне этих функций и запускает программу.
- Не вмешиваясь, профайлер дожидается окончания выполнения программы и извлекает накопленные значения счетчиков.
- Процесс повторяется до обработки всех подмножеств.
- Далее, результаты объединяются и считается доля каждой функции в общей сумме.

3.4. Сбор статистики потребления ресурсов функциями для циклических программ

В данном случае профайлер, во-первых, лишен возможности детерминировать поток выполнения программы, а во-вторых, такие программы могут работать неограниченное количество времени и нет четкого понимания, в какой момент программу можно прервать для перезапуска.

Однако, такие программы часто обладают другим важным свойством — внутри них имеется участок, во время которого программа не выполняет никаких действий и только ожидает поступления новых данных. В этот момент программу можно безопасно остановить, не повлияв существенно на результаты измерений производительности кода, относящегося к ее непосредственной логике.

Поэтому анализ можно свести к данному сценарию.

- Пользователь предоставляет профайлеру машинный код программы с отладочной информацией и адрес инструкции, на которой можно безопасно произвести остановку.
- Профайлер извлекает список функций и разбивает его на соответствующие количеству доступных счетчиков подмножества.
- Далее, профайлер засекает некоторое заданное время и по его истечении активирует точку останова на предоставленном пользователем адресе.

- По остановке, профайлер производит переконфигурирование счетчиков на новое подмножество функций.
- По истечении непроанализированных функций профайлер возвращается к первому подмножеству.
- Процесс прерывается по запросу пользователя.

Данный подход не гарантирует точность измерений, поскольку поток выполнения сильно не детерминирован. Однако преимущество этого способа заключается в минимальном вмешательстве в код непосредственной логики программы и тем самым позволяет анализировать высоконагруженные ограниченные по времени ответа программы в реальных условиях.

3.5. Оптимизация анализа

Стоит отметить, что количество функций, содержащихся в программе, существенно превышает количество имеющихся счетчиков. Таким образом, наивная схема разбиения множества функций на подмножества приводит к линейному росту времени анализа относительно количества функций. Например, программа, содержащая 1000 функций и выполняющаяся 10 секунд, при использовании 8 счетчиков, потребует $\frac{1000}{8} \cdot 10 = 1250$ секунд (21 минуту) на полный анализ. 2000 функций потребуют 42 минуты соответственно.

Однако, при необходимости поиска наиболее затратных участков, пользователя не интересуют функции, потребляющие сравнительно небольшое количество времени, и напротив, интересуют наиболее тяжелые функции. Таким образом, профайлер способен динамически исключать из анализа незатратные функции, используя следующую оценку.

Пусть $M(f)$ — количество измеряемых событий, произошедших за время исполнения функции f. $P(f)$ — множество функций, вызывающих функцию f. Тогда $M(f) \leq \sum_{p \in P(f)} M(p)$.

Доказательство: пусть $O(p, f)$ — количество измеряемых событий, произошедших во время выполнения f в рамках выполнения p. То-

гда $\forall p \in P(f), M(p) = O(p, f) + T(p, f)$, где $T(p, f)$ — события, произошедшие во время выполнения p, но не связанные с f. При этом, $\sum_{p \in P(f)} O(p, f) = M(f)$, поскольку никакими другими путями, кроме как будучи вызванной одной из старших функций, функция f не могла быть выполнена. Тогда $\sum_{p \in P(f)} M(p) = M(f) + \sum_{p \in P(f)} T(p, f)$. Что включает в себя $M(f)$ и следовательно не меньше, чем $M(f)$.

Таким образом, если пользователю необходимо найти все функции, чья доля затрачиваемых ресурсов не меньше $n\%$, то любая функция, оценка которой не превышает n, может быть проигнорирована при анализе.

На практике, оптимизация производится посредством спуска по графу вызовов начиная с функций, которые не были вызваны никакими другими сущностями. Далее, по мере сбора данных о функциях верхних уровней, вычисляется оценка для функций, вызванных из них, и производится исключение заведомо наименее затратных функций.

3.6. Метрики

Помимо получения непосредственно количества срабатываний того или иного события возможно также вычислить по этим результатам некоторые новые полезные данные, обладающие полезной для пользователя семантикой. Назовем такие функции от первичных данных метриками. Например, доля потребления процессорного времени некоторой функцией программы является метрикой этой функции. Она рассчитывается как отношение количества циклов, затраченного на выполнение функции, к количеству циклов, затраченному на выполнение всей программы.

Необходимо также отметить важное свойство, присущее только части метрик. Назовем его монотонностью. Метрика является монотонной, если ее значение для любой функции не меньше суммы значений этой метрики для всех функций, вызванных из данной, если эти функции не были вызваны никакими иными сущностями. Например, доля потребления процессорного времени является монотонной, поскольку

время выполнения любой функции включает в себя выполнение всех дочерних функций.

Напротив, пусть, например, дана метрика, представляющая долю циклов, потраченных на ожидание данных при промахе кэша. Она рассчитывается как отношение циклов в ожидании данных к полному количеству циклов, затраченных функцией. Такая метрика не является монотонной.

Монотонность метрики позволяет производить спуск по графу вызовов, используя тот же метод ее оценки, что и для простого количества событий.

4. Решение

Разработанный профайлер представлен в виде модуля Python, подключаемого к отладчику и добавляющего набор команд для управления процессом анализа.

Решение подразделяется на несколько подсистем. Их структура представлена на диаграмме 1.

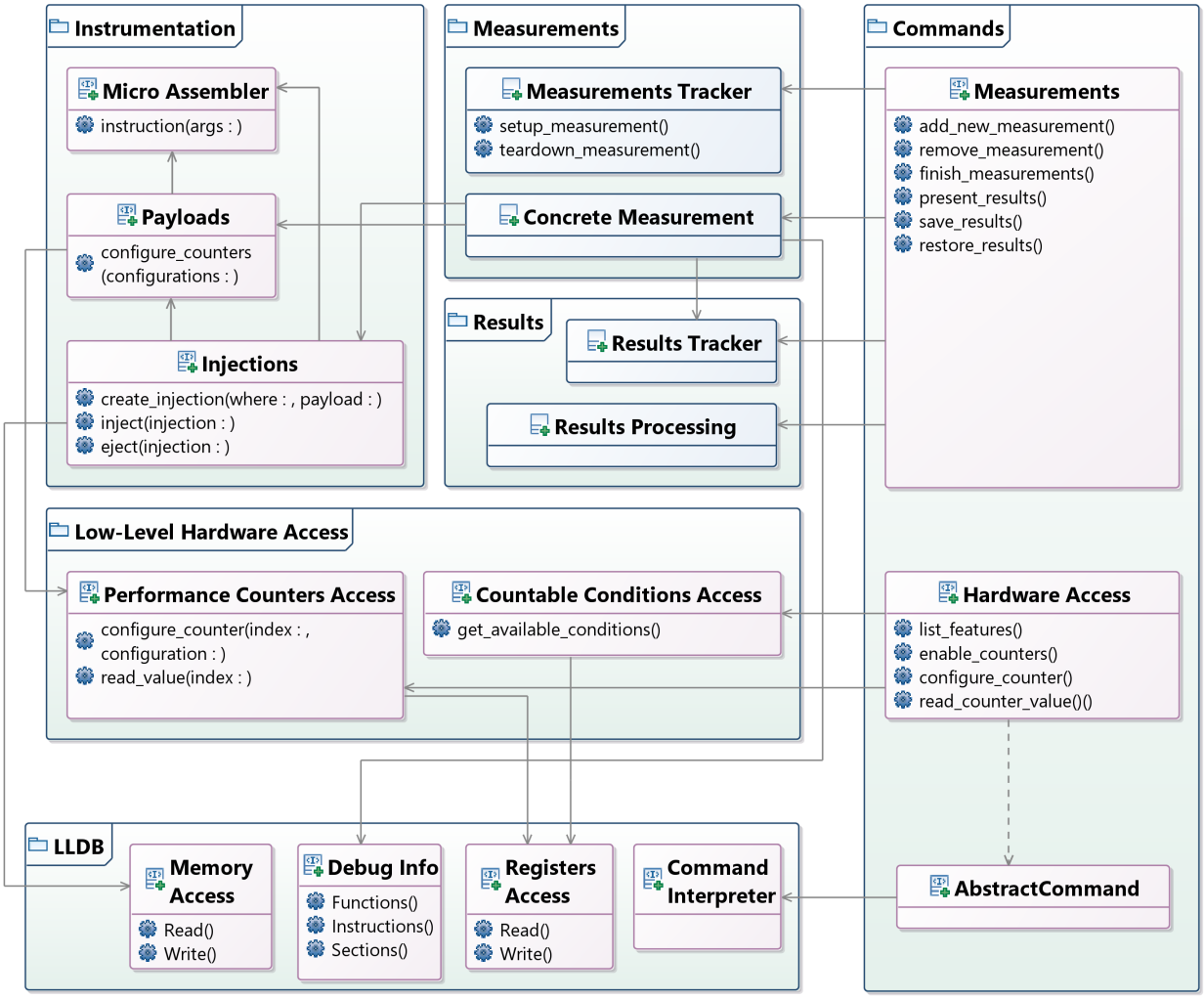

Рис. 1: Общая архитектура решения

4.1. Взаимодействие с оборудованием

Первая из них (Hardware Access) абстрагирует непосредственное взаимодействие с регистрами, конфигурирующими счетчики производительности, от высокоуровневых компонентов программы. Она предо-

ставляет возможность запросить все доступные для счета события, сконфигурировать счетчики, а также получить их значения.

Реализация этой подсистемы не содержит каких-либо обрабатывающих алгоритмов и лишь объединяет последовательности изменения состояний регистров в семантически обособленные действия.

4.2. Инструментация

Данная подсистема реализует один из двух представленных способов проведения замеров на участке кода. А именно, добавляет возможность внедрения произвольного кода в память запущенного процесса, в частности, для программной активации и деактивации счетчиков производительности на концах участка. Она также подразделяется на несколько модулей.

4.2.1. Инъекции

Первый из них — это модуль, непосредственно отвечающий за внедрение произвольного кода в код программы (Runtime Injection). Сущности, с которыми оперирует данный модуль, называются инъекциями. Имеется возможность создания инъекции по участку кода, внедрение инъекции, а также ее изъятие с целью вернуть состояние кода в прежний вид.

Инъекция призвана изменить поток управления программы и заставить процессор выполнить последовательность дополнительных инструкций. В частности, для проведения замера на участке предлагается внедрить в его начало последовательность инструкций, активирующую счетчики, а в конец последовательность, деактивирующую их.

При этом необходимо учитывать ряд ограничений. Первое из них состоит в том, что готовый машинный код может использовать относительную адресацию, поэтому невозможно произвести инъекцию, сдвинув последующие инструкции на несколько байт вперед. Таким образом, предлагается замещать часть инструкций инструкцией безусловного перехода (трамплином) в участок памяти, где размещается по-

следовательность дополнительных инструкций, а за ними размещается замещенная последовательность. Иллюстрация потока управления программы в таком случае изображена на рисунке 2.

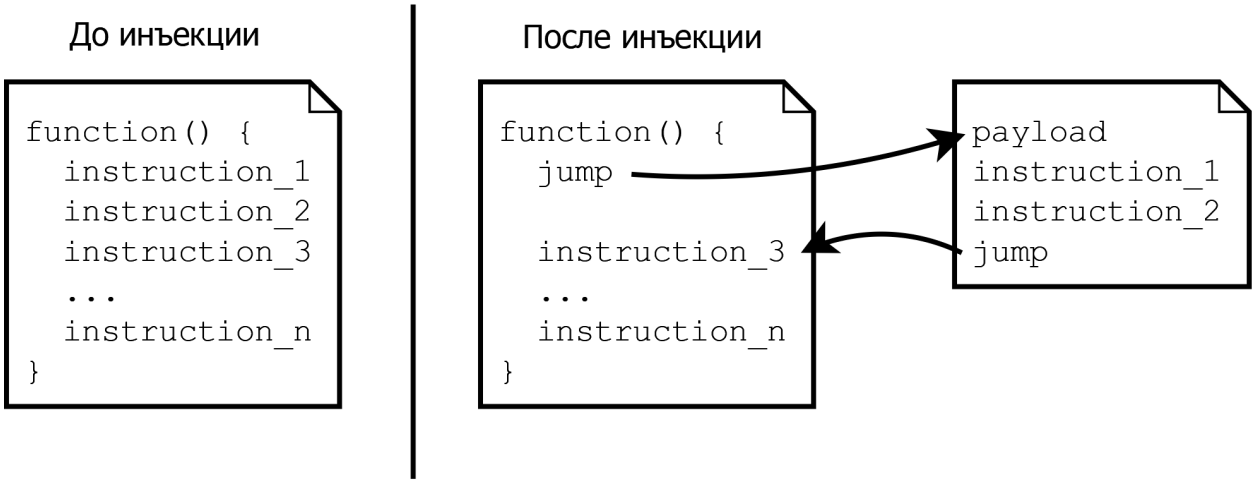

Рис. 2: Изменение потока управления при проведении инъекции.

В приведенном примере две инструкции были замещены трамплином и перенесены в отдельный участок кода. Перед ними добавлена полезная нагрузка, а после них выполняется возврат в точку, с которой должно продолжиться выполнение программы.

Второе ограничение заключается в том, что перемещение некоторых инструкций приведет к некорректному их поведению. Например, к таким инструкциям относятся использующие относительную адресацию. Поэтому, при создании инъекции требуется проверить, не осуществляется ли попытка переноса непереносимых инструкций.

Третье ограничение следует из того, что длина инструкции безусловного перехода, в зависимости от удаленности цели, может превышать размер самой короткой инструкции. В данном случае необходимо проверить, не существует ли в программе какой-либо инструкции, осуществляющей переход в середину замещенного участка. Пример инъекции, не соблюдающей данное ограничение, приведен на рисунке 3.

В данном примере в программе существует инструкция, осуществляющая переход на одну из замещенных инструкций. После замещения переход будет производиться в ”середину” трамплинной инструкции, что приведет к ошибке времени выполнения.

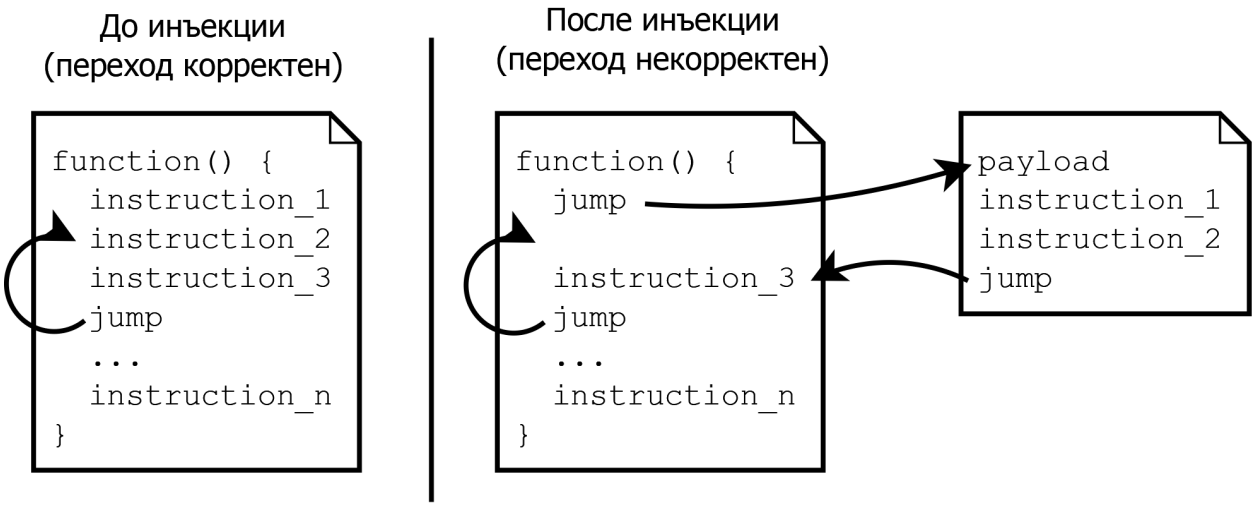

Рис. 3: Некорректное поведение программы после инъекции.

Наконец, четвертое ограничение требует того, чтобы все пути в графе потока управления, исходящие из начала участка, гарантированно приходили к его концу. Это ограничение следует из необходимости гарантированно отключить счетчики по достижении конца участка. Пример подходящего и неподходящего участков приведен на рисунке 4.

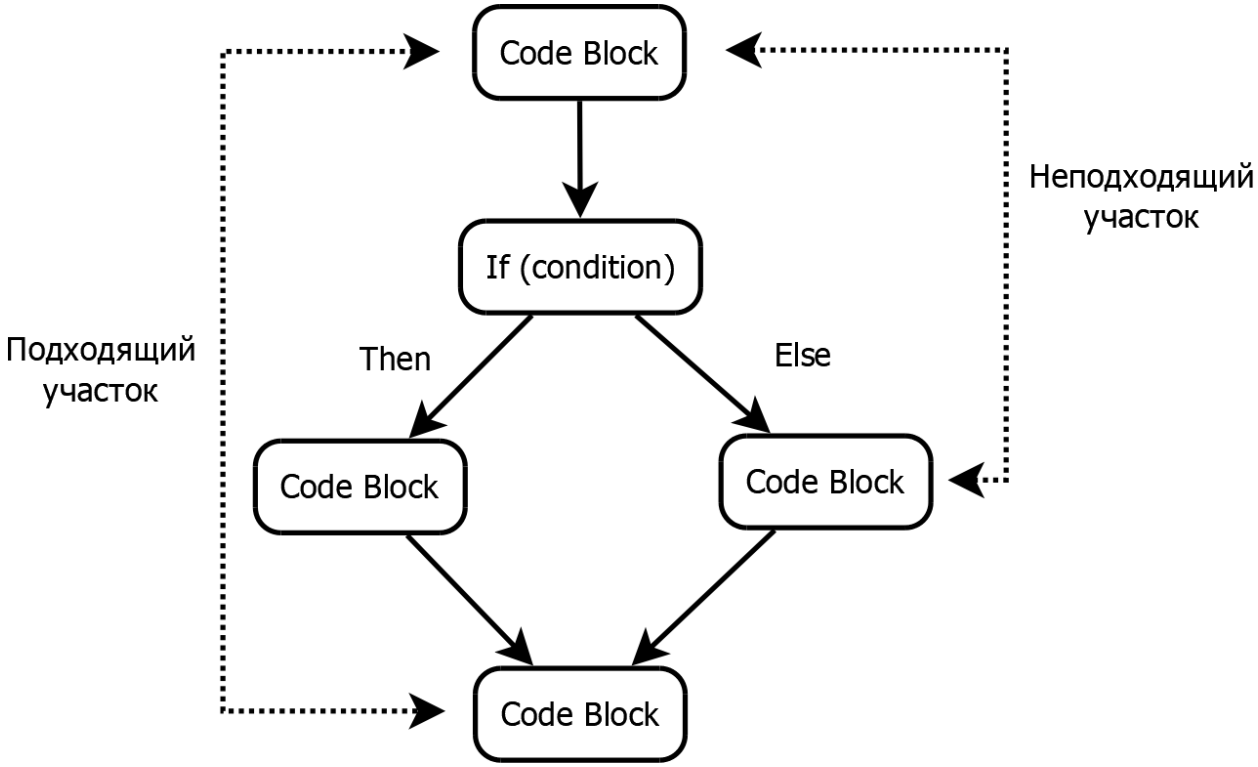

Рис. 4: Участки кода, поддающиеся и не поддающиеся инструментированию.

В данном примере при прохождении по любому из двух путей в графе потока управления отключение счетчиков по достижении конца подходящего участка будет гарантированно произведено. Однако, если поток пройдет только по пути, соответствующему истинности условия, счетчики не будут отключены по достижении конца неподходящего участка и их показания будут превышать реальные значения.

В решении реализована автоматическая проверка удовлетворения второго и третьего ограничений. Удовлетворение четвертого ограничения требует существенно более сложного статического анализа и вследствие этого должно гарантироваться пользователем.

В связи с перечисленными ограничениями, инъекции не всегда могут быть внедрены в произвольную позицию в программе. Кроме того, для активации счетчиков в начале участка и деактивации их в конце, требуется провести две инъекции, что также снижает вероятность успеха. Однако на практике, при инструментировании участков, соответствующих функциям, инъекции возможны в подавляющем большинстве случаев. Кроме того, в данной работе также применена оптимизация парных инъекций в случае функций, смягчающая четвертое ограничение и позволяющая сократить требуемое количество инъекций с двух до одной.

Данная оптимизация основана на знании о том, что функция по завершении произведет переход по адресу возврата, сохраненному во время вызова. Таким образом, внедряется только одна инъекция на входе в функцию, которая также замещает адрес возврата адресом, по которому расположен код, который необходимо выполнить при выходе из функции. Данная схема изображена на рисунке 5.

4.2.2. Полезная нагрузка

Данный модуль (Payloads) отвечает за генерацию кода, реализующего необходимые профайлеру действия. В их число входит конфигурация счетчиков, а также их включение и отключение. Далее, сгенерированная нагрузка может быть использована как параметр при создании инъекции. Основное требование к сгенерированному коду — отсутствие

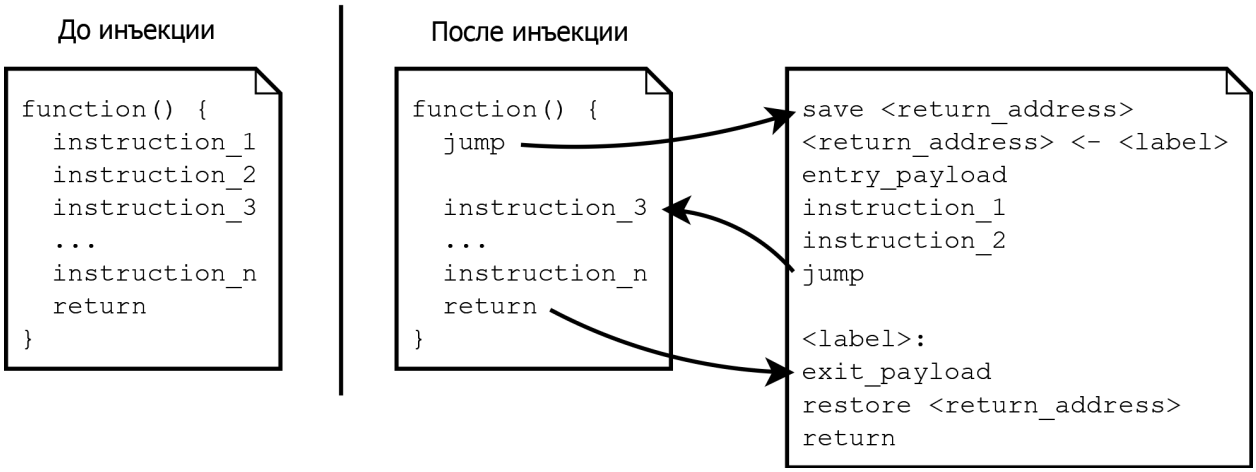

Рис. 5: Выполнение двух полезных нагрузок посредством одной инъекции в случае функций.

любых побочных эффектов, поскольку он может быть внедрен между любыми инструкциями программы.

4.2.3. Вспомогательный ассемблер

Данный модуль предоставляет интерфейс для генерации машинного кода, использующийся как при создании полезной нагрузки, так и при создании составляющих элементов инъекции, таких как трамплины и взаимодействие с адресами возврата. Он реализован как набор функций, каждая из которых соответствует имени какой-либо инструкции процессора и принимает соответствующие аргументы. Результатом таких функций является последовательность байт, представляющих машинный код сгенерированной инструкции.

4.3. Измерения

Данная подсистема более подробно изображена на рисунке 6. Она отвечает за создание, удаление, отслеживание и проведение различных измерений, в числе которых:

- извлечение графа вызовов;
- измерение количества событий на протяжении всего времени выполнения программы.

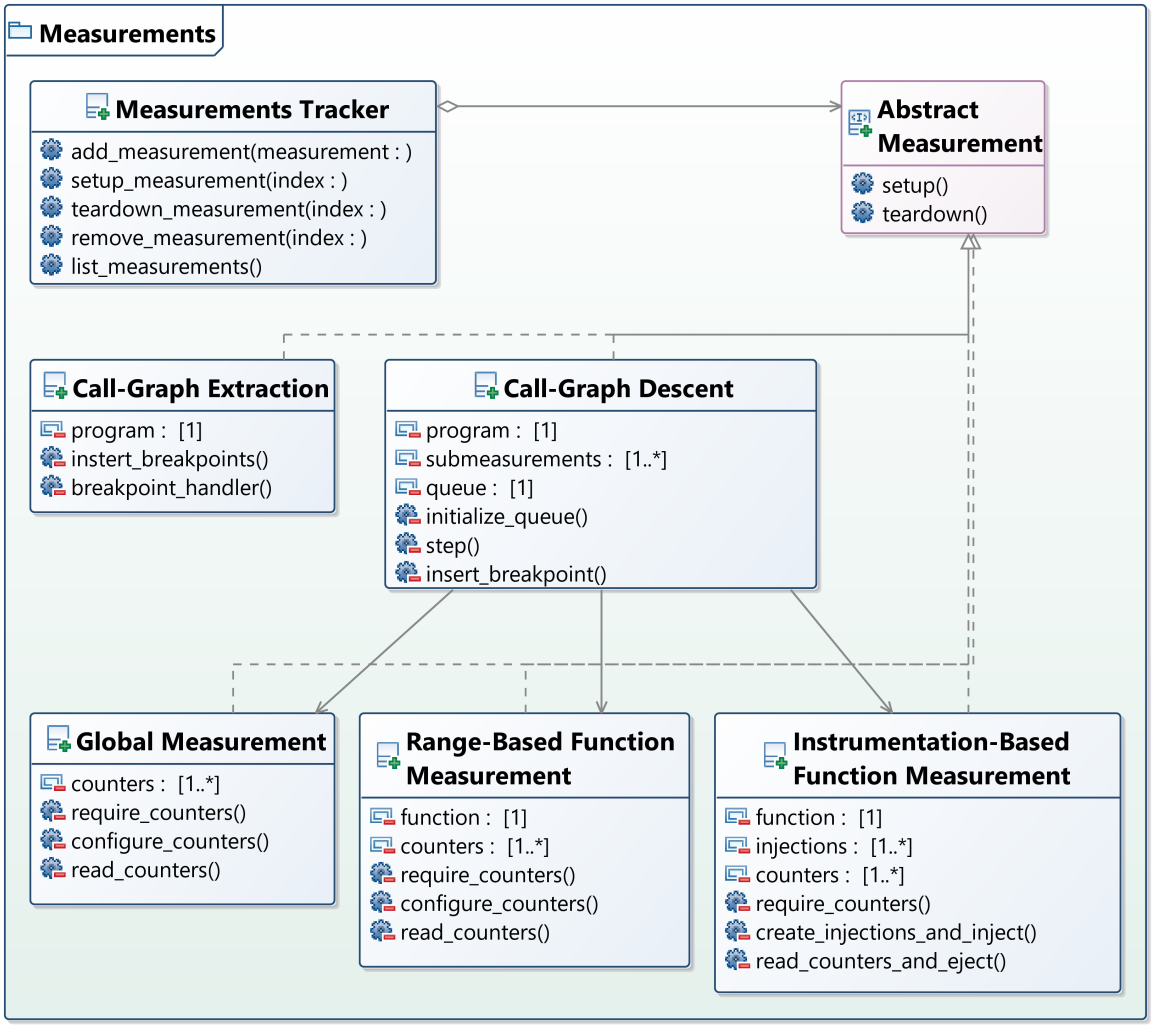

Рис. 6: Структура модуля измерений.

- измерение количества событий, произошедших во время выполнения одной функции (включая ее зависимости или исключая их);
- выполнение автоматического спуска по графу вызовов с целью выделить наиболее затратные функции.

Каждое из этих измерений реализует интерфейс абстрактного измерения, позволяющего в любой момент выполнения произвести его активацию или деактивацию. По деактивации измерение обязано сохранить накопленные результаты и вернуть код и использованные счетчики в прежнее состояние.

Управление измерениями производится посредством обращения к трекеру измерений. Он хранит список всех измерений и сопоставляет

им уникальные идентификаторы, использующиеся при взаимодействиях с измерениями.

4.3.1. Извлечение графа вызовов

Данное измерение реализуется посредством установки точек останова на входе в каждую функцию, которым назначется обработчик, считывающий последние два кадра стека и добавляющий в граф вызовов ребро, идущее из функции, соответствующей предпоследнему кадру, в функцию, соответствующую последнему.

Данное измерение приводит к существенному замедлению выполнения программы, поэтому подразумевается единоразовое его выполнение и последующее переиспользование сохраненных результатов. Кроме того, проведение данного измерения может быть заменено любым другим способом получения графа вызовов, вплоть до статического извлечения из исходного кода.

4.3.2. Глобальный подсчет событий

Данное измерение реализуется посредством активации заданного набора счетчиков при инициализации и их деактивации по окончании измерения. Оно используется совместно с локальным подсчетом событий для вычисления доли конкретного участка по отношению ко всей программе. Кроме того, оно реализует сценарий ручного замера событий на участке кода.

4.3.3. Подсчет событий на участке кода

Данное измерение принимает от пользователя требуемый набор измеряемых событий и реквизирует соответствующее ему количество счетчиков. Далее, при инициализации, может использоваться один из двух методов активации счетчиков на участке: инструментация или задание диапазона. По завершении измерения, значения использованных счетчиков добавляются в результаты измерений с пометкой о принадлежности соответствующему участку. При этом, на протяжении всего измере-

ния не происходит никакого вмешательства в выполнение программы со стороны профайлера. Таким образом, данное измерение реализует сценарий автоматического замера событий на участке кода.

4.3.4. Спуск по графу вызовов

Данный тип измерений требует наличия информации о графе вызовов, который может быть получен другим соответствующим измерением ранее либо загружен извне. Также, пользователю требуется указать метрику, по которой будет вычисляться оценка дочерних функций по родительским, минимальное значение метрики, точку, в которой будет производиться переконфигурация и функцию, с которой будет начинаться анализ. По умолчанию анализ начинается со всех функций, которые не вызываются никакими другими, так как они потенциально могут быть точками входа в программу.

Анализ производится следующим образом.

Во-первых, производится обход графа вызовов в ширину и функции в том же порядке добавляются в очередь. Такой порядок важен, поскольку позволяет минимизировать погрешность при измерении функций верхних уровней, возникающую из-за дополнительных расходов в функциях нижних уровней. Кроме того, это позволяет вовремя вычислять оценку и отбрасывать заведомо незначительные функции. То есть, к тому времени, когда функция будет извлечена из очереди, функции, вызывающие ее, с большой вероятностью уже будут проанализированы.

Стоит отметить, что гарантировать вычисление оценки для всех функций, имеющих предков, возможно, но нецелесообразно, поскольку в таком случае, даже при наличии доступных счетчиков, будет необходимо дождаться завершения анализа предков перед тем как приступить к анализу самой функции и часть счетчиков будет простаивать. Предложенный подход позволяет, хоть и не идеально, распределить оставшиеся счетчики по функциям, оценка для которых еще не может быть посчитана.

Во-вторых, вводится понятие шага анализа. На каждом шаге в цикле из очереди извлекается по одной вершине, пока доступно необходи-

мое для вычисления метрики количество счетчиков. Далее вычисляется оценка метрики для извлеченной функции по ее родителям, и если оценка меньше заданного числа, функция игнорируется. В случае, если функция не была отброшена, создается измерение типа "подсчет событий на участке" и сохраняется его идентификатор. Когда доступные счетчики иссякнут, запускается выполнение программы до точки, на которой производится переконфигурация. Также, опционально, перед каждым шагом может производиться перезапуск программы.

По достижении точки переконфигурации выполняются следующие действия. Все созданные измерения типа "подсчет событий на участке" завершаются. На основе их результатов вычисляются значения метрики для каждой проанализированной функции. Выполняется следующий шаг анализа.

Анализ завершается либо по требованию пользователя, либо как только очередь становится пустой. При необходимости, очередь инициализируется снова.

4.4. Обработка результатов

Данный модуль отвечает за хранение результатов, вычисление метрик на их основе, а также за представление результатов в удобочитаемом виде. Более подробная схема представлена на диаграмме 7.

4.4.1. Хранение результатов

Данный подмодуль оперирует структурами, соответствующими каждому "набору" измерений. Измерения принадлежат одному набору в том случае, если они были выполнены в рамках одного выполнения программы и из-за этого их результаты могут интерпретироваться совместно. Например, если в одном наборе расположен глобальный подсчет циклов и подсчет циклов в рамках одной функции, то имеется возможность, разделив результат второго измерения на результат первого, получить относительное потребление процессорного времени данной функцией. Напротив, если глобальный подсчет циклов выполнен в

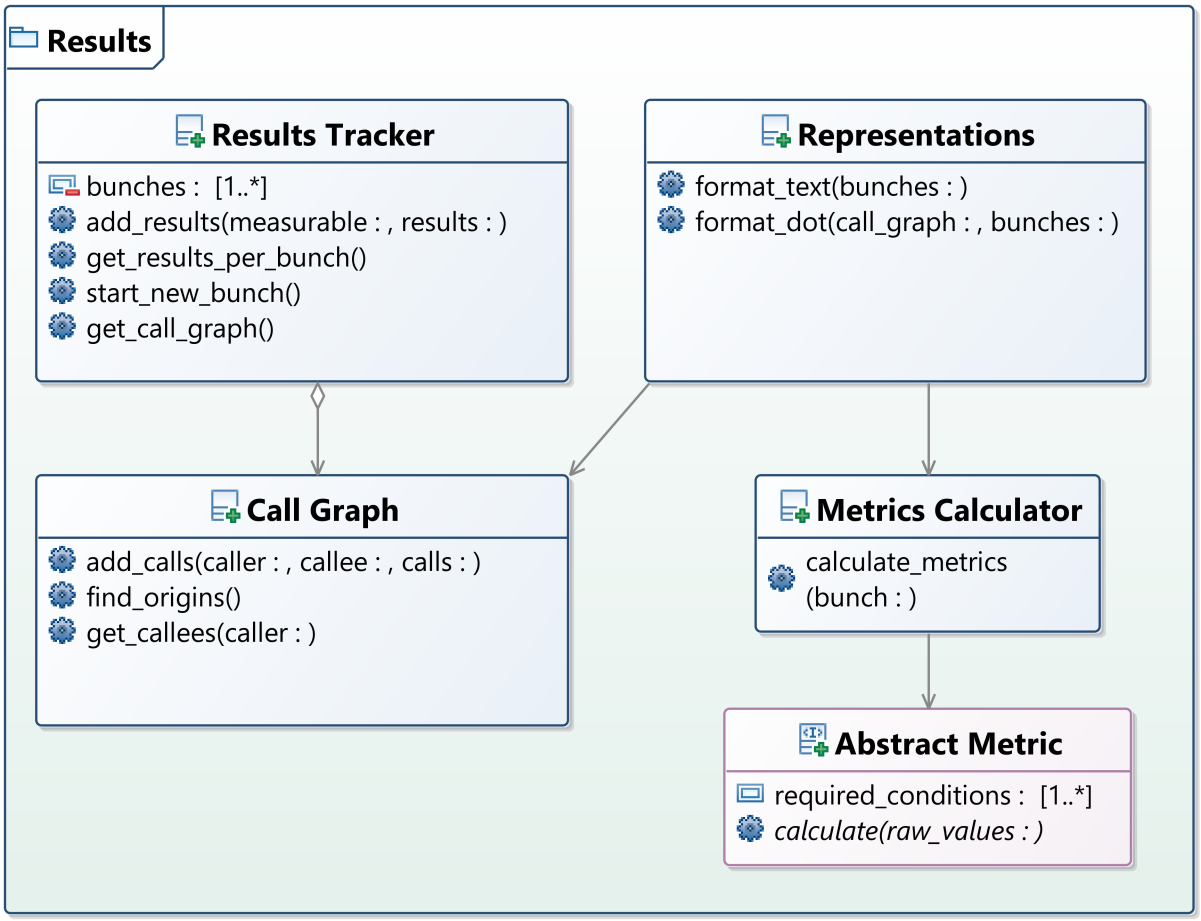

Рис. 7: Структура модуля результатов.

рамках одного выполнения программы, а подсчет циклов в функции в рамках другого, то вычисление описанной метрики не будет репрезентативным, поскольку два разных выполнения программы могут происходить за разное количество циклов. Таким образом, эти два измерения должны принадлежать разным наборам.

Профайлер хранит список всех результатов, соответствующих каждому набору, а также позволяет сохранять их на диск и загружать для продолжения анализа.

Также к этому подмодулю относится хранение информации о графе вызовов, которая также может быть сохранена на диск и загружена с него.

4.4.2. Вычисление метрик

Данный подмодуль отвечает за обработку первичных данных, полученных с помощью счетчиков. Его задача заключается в абстрагировании вычисления всех возможных метрик по известным данным. В результате, модуль предоставляет внешний интерфейс, получающий набор измерений и возвращающий рассчитанные метрики.

В связи с потенциально большим количеством возможных метрик применяется следующая схема поиска доступных из них. Сначала строится и кэшируется отображение из возможных исходных данных в зависящие от них метрики. Например, метрика, заключающаяся в расчете относительного количества циклов для некоторой функции добавляется в зависимые от полного количества циклов и от локального количества циклов. Таким образом образуется граф зависимостей между исходными данными и метриками (пример на рисунке 8).

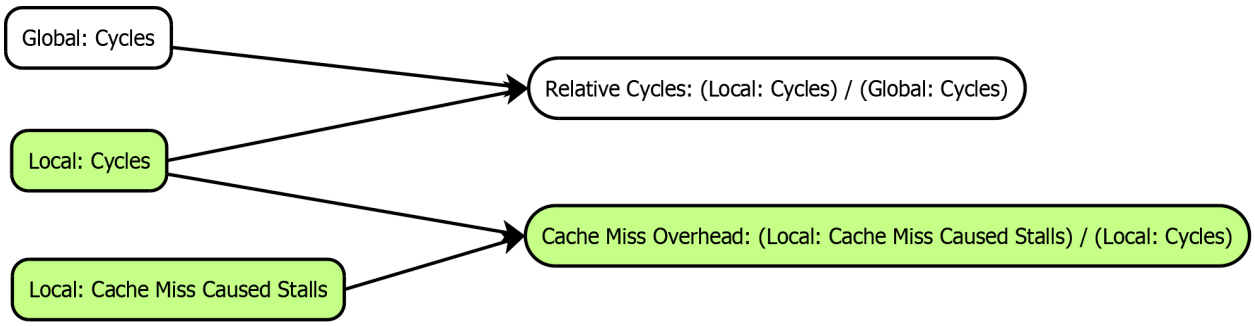

Рис. 8: Пример графа вычисления метрик.

Далее, перебираются все доступные исходные данные и все зависимые от них метрики помечаются как обладающие этими данными. Если после очередной серии пометок некоторая метрика обладает всеми необходимыми данными, происходит ее вычисление и добавление в результат.

Такой подход позволяет избежать полного перебора возможных метрик, что в подавляющем большинстве случаев и не является необходимым, поскольку чаще всего пользователь собирает данные для одной-двух метрик из возможных сотен.

4.4.3. Представление результатов

Данный подмодуль предоставляет три возможных представления.

Во-первых, результаты могут быть представлены в необработанном виде. Это полезно при небольших и очень специфичных измерениях, по которым нельзя рассчитать метрики либо проассоциировать результат с функциями из графа вызовов. К таким измерениям относятся, например, измерения на участках, чьи границы не соответствуют границам ни одной функции.

Во-вторых, результаты могут быть представлены в текстовом виде. В этом случае каждому измеренному участку или функции сопоставляется как можно больше информации, которая может быть получена из имеющихся необработанных результатов. Например, производится попытка рассчитать все известные метрики.

В-третьих, результаты могут быть представлены в виде аннотированного графа вызовов в формате DOT. В этом случае также производится расчет метрик, а их результат записывается вместе с исходными данными как дополнительный текст к узлу в графе. Далее граф может быть визуализирован с помощью graphviz [3] или любой похожей программы.

4.5. Пользовательские команды

Данная подсистема предоставляет пользовательский интерфейс для большей части возможностей профайлера в виде команд для интерпретатора LLDB.

Каждая команда представлена в виде класса, импортируемого в LLDB при попытке импорта модуля профайлера. Каждый класс регистрирует себя как отдельную команду для встроенного интерпретатора LLDB и задает ссылку на код обработчика команды. После чего команды доступны для пользователя напрямую из интерпретатора LLDB.

Команды логически подразделяются в соответствии с делением остальной части профайлера на подсистемы. В частности, существуют наборы команд, отражающие подсистему взаимодействия с оборудованием, ав-

томатические сценарии анализа и обработку результатов.

Реализация команд не несет никакой дополнительной логики и содержит лишь обращения к имеющимся интерфейсам других модулей.

5. Тестирование

В качестве основного подхода к тестированию кода разработанного решения было выбрано написание автоматических тестов ко всем подсистемам с сохранением зависимостей от подсистем более низкого уровня. Например, подсистема взаимодействия с оборудованием тестируется в интеграции с реальным отладчиком, запускающим программу на симуляторе процессора ARC. Пользовательские команды в свою очередь тестируются в предположении корректности подсистемы взаимодействия с оборудованием и также с полной интеграцией с отладчиком и симулятором.

Для непосредственного написания тестов используется библиотека unittest [7]. Однако, существует проблема с запуском тестов напрямую, поскольку они требуют предварительной настройки отладчика и загрузки в него модуля профайлера. Таким образом, для запуска тестов выбран следующий подход. Диаграмма последовательности действий приведена на рисунке 9.

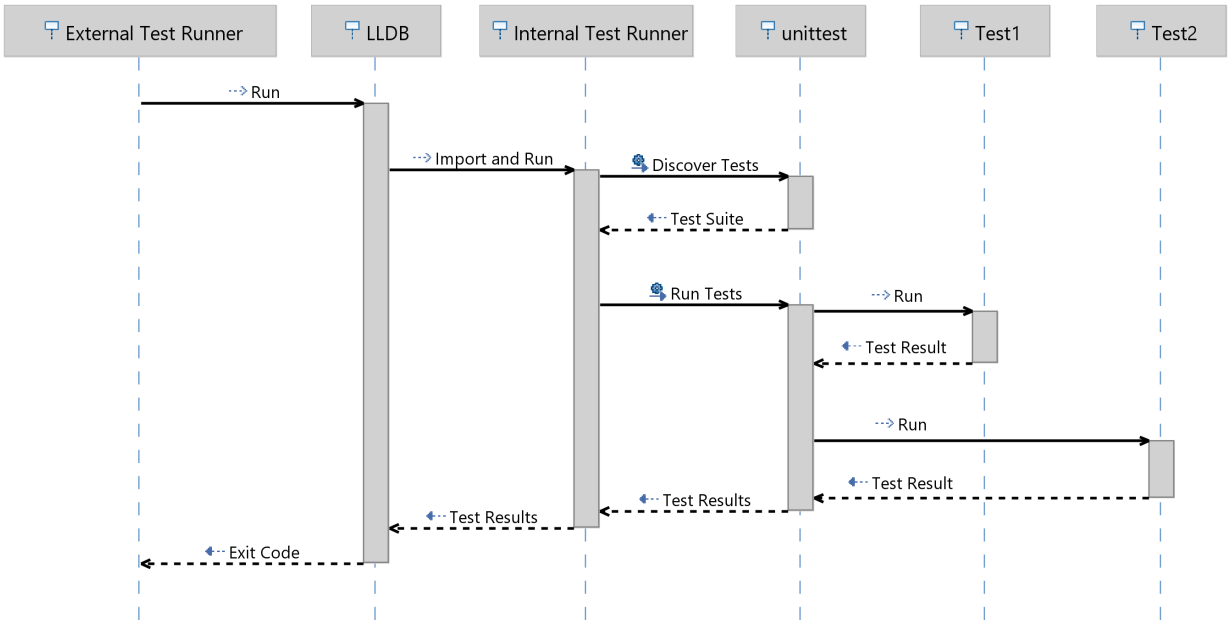

Рис. 9: Процесс выполнения тестов

Сначала, запускается скрипт, цель которого запустить отладчик и передать в него команды для загрузки тестов. Далее выполнение тестов

начинается уже внутри интерпретатора Python, встроенного в LLDB. Технически это реализовано как добавление и выполнение отдельной команды интерпретатора команд LLDB. Внутри кода обработчика этой команды происходит поиск и загрузка тестов, посредством встроенных инструментов unittest. В результате, тесты запускаются внутри отладчика и имеют полный доступ к его интерфейсу.

6. Апробация

Решение было опробовано на бенчмарке для встраиваемых систем Embench [1]. Также была проанализирована небольшая программа, использующая библиотеку zlib [4]. Все примеры запускались на одной из стандартных конфигураций ядра ARC HS.

При этом было необходимо оценить точность полученного решения и вносимые дополнительные расходы, поскольку данные характеристики являются наиболее важными при оценке качества профайлера. Кроме того, было необходимо изучить границы применимости профайлера, поскольку метод инструментации, выбранный в качестве одной из основ, требует специальных условий.

6.1. Оценка точности

Для оценки точности был использована симуляция xCAM, позволяющая измерить количество циклов, затраченное на выполнение какой-либо функции с идеальной точностью. Далее, результаты для тех же функций были получены с помощью реализованного профайлера. После чего было рассчитано отклонение полученных результатов от идеальных. Результаты приведены в таблице 1.

Тест	Длительность теста (в циклах)	Количество функций в тесте	Интервал абсолютных погрешностей по функциям (в циклах)	Количество вызовов в тесте	Дополнительные расходы на вызов (в циклах)	Интервал относительных погрешностей по функциям (%)
aha_mont64	34734	5	13-859	21	40.9	0.2-2.47
crc32	194253	2	14-81	2	40.5	0.01-0.04
cubic	201084	1	18-18	1	18.0	0.01-0.01
edn	553189	9	4-253	9	28.1	0.0-0.84
huffbench	907810	3	818-3277	96	34.1	0.35-3.39
matmult_int	1253556	4	61-7412	803	9.2	0.0-5.81
minver	23416	3	27-379	11	34.5	0.33-2.57
nettle_aes	138350	10	12-659	12	54.9	0.02-1.76
nettle_sha256	14077	7	13-499	9	55.4	0.34-12.05
slre	92523	13	77-45949	1168	39.3	13.44-65.09
st	701853	9	9-29018	613	47.3	0.05-6.48
statemate	4403	4	110-239	5	47.8	2.86-5.43
ud	17212	2	19-70	2	35.0	0.15-0.41
zlib	234445	28	1-298	28	10.6	0.0-4.83

Таблица 1: Погрешности измерений на тестах из Embench и программе с zlib.

На большинстве функций профайлер показывает значения с отно-

сительной погрешностью, не превышающей 5% (в среднем — 3%). Дополнительные расходы на каждый вызов близки к константным и сопоставимы с вызовом одной пустой функции. Наихудшие результаты проявляются в случае большого количества часто вызываемых маленьких функций (например, в тесте slre). В этом случае погрешность накапливается из-за дополнительных расходов.

6.2. Оценка применимости

В ходе исследования 100 из 117 функций из zlib и Embench были успешно проинструментированы. При этом проблемы возникали только в случае небольших по размеру функций, поскольку в них не было достаточного количества переносимых инструкций для инъекций.

6.3. Сравнение с существующими инструментами

Также было проведено эмпирическое и статистическое сравнение с существующими инструментами, предоставляемыми компанией Synopsys для анализа программ для процессоров ARC. Результаты приведены в таблице 2.

Инструмент	Используемый метод	Может использоваться на реальном оборудовании	Требует предварительной подготовки	Анализ за один проход	Замедление	Точность
xCAM	Симуляция на уровне логических вентилей	Нет	Нет	Да	На порядки	Близка к идеальной
nSIM	Симуляция инструкций	Нет	Нет	Да	В разы	Результаты коррелируют с реальными
gprof	Инструментация + сэмплирование	Да	Да	Да	до 120%	Результаты коррелируют с реальными при длительном выполнении
Разработанное решение	Инструментация + счетчики производительности	Да	Нет	Нет	Сравнимо с вызовом одной пустой функции на каждую анализируемую	Погрешность в среднем 3%. Увеличивается с уменьшением анализируемой функции

Таблица 2: Сравнение полученного решения с существующими инструментами для профилирования на ARC.

Данное решение может применяться на реальном оборудовании в отличие от xCAM и nSIM. Оно не требует дополнительной подготовки перед проведением анализа, что является преимуществом перед gprof, который требует компиляции анализируемой программы со специальным флагом. Решение практически не приводит к замедлению выполнения программы и обладает точностью, сравнимой с xCAM. Главными недостатками представленного решения является невозможность проанализировать всю программу за один проход, а также то, что существуют функции, не поддающиеся инструментированию описанным методом.

7. Заключение

В ходе выполнения данной работы были достигнуты следующие результаты.

- Изучены существующие методы разработки профайлеров, в качестве средств замера производительности выбраны встроенные в ядро ARC счетчики производительности.
- Рассмотрены следующие сценарии использования профайлера: ручной и автоматический замеры участка кода, поиск наиболее затратных функций. Предложены алгоритмы их выполнения и их оптимизации.
- Спроектирована архитектура профайлера и реализованы все рассмотренные сценарии.
- Решение протестировано. Точность оценена на примерах zlib и Embench, погрешность измерения циклов составила в среднем 3%, однако она существенно увеличивается при анализе коротких функций.

Список литературы

[1] Free and Open Source Silicon Foundation CIC. Embench: A Modern Embedded Benchmark Suite. — 2019. — Access mode: https://www.embench.org/ (online; accessed: 2020-06-03).

[2] Graham Susan L., Kessler Peter B., Mckusick Marshall K. Gprof: A Call Graph Execution Profiler // SIGPLAN Not. — 1982. — Jun. — Vol. 17, no. 6. — P. 120–126. — Access mode: https://doi.org/10.1145/872726.806987.

[3] Graphviz Team. Graphviz - Graph Visualization Software. — 2020. — Access mode: https://www.graphviz.org/ (online; accessed: 2020-05-20).

[4] Greg Roelofs Jean-loup Gailly, Adler Mark. zlib. — 2017. — Access mode: https://zlib.net/ (online; accessed: 2020-06-03).

[5] Intel Corporation. Hardware Event-based Sampling Collection. — 2020. — Access mode: https://software.intel.com/en-us/vtune-help-hardware-event-based-sampling-collection (online; accessed: 2020-04-12).

[6] Patel Rajendra. A Survey of Embedded Software Profiling Methodologies // International Journal of Embedded Systems and Applications. — 2011. — Dec. — Vol. 1, no. 2. — P. 19–40. — Access mode: http://dx.doi.org/10.5121/ijesa.2011.1203.

[7] Python Software Foundation. unittest. — 2020. — Access mode: https://docs.python.org/3/library/unittest.html (online; accessed: 2020-03-17).

[8] R.S. Oderov. Evaluation and testing the sampling profiling approach using the example of Intel VTune Amplifier XE 2011. — 2012. — Access mode: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/345/345_Oderov_report.pdf (online; accessed: 2020-04-12).

[9] Synopsys. DesignWare ARC nSIM. — 2020. — Access mode: https://www.synopsys.com/dw/ipdir.php?ds=sim_nsim (online; accessed: 2020-05-20).

[10] Synopsys. DesignWare ARC xCAM. — 2020. — Access mode: https://www.synopsys.com/dw/ipdir.php?ds=sim_xcam (online; accessed: 2020-05-20).

[11] Synopsys. DesignWare Real-Time Trace Options. — 2020. — Access mode: https://www.synopsys.com/dw/ipdir.php?ds=arc-real-time-trace (online; accessed: 2020-05-20).

[12] Synopsys. Processor Solutions. — 2020. — Access mode: https://www.synopsys.com/designware-ip/processor-solutions/general.html (online; accessed: 2020-04-12).

[13] The LLDB Team. LLDB Homepage. — 2020. — Access mode: https://lldb.llvm.org/ (online; accessed: 2020-03-17).