Санкт-Петербургский государственный университет

Кафедра Системного Программирования

Ефремов Ростислав Сергеевич

Поиск состояний гонки в приложениях, исполняемых в среде ОС z/OS

Бакалаврская работа

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

Рецензент:
ст. инженер-программист ООО «Эксперт-Система», к. т. н. Трифанов В. Ю

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engeneering

Rostislav Efremov

Data race detection in z/OS environment

Graduation Thesis

Scientific supervisor:
Professor Andrey Terekhov

Reviewer:
Sr. Software Engeneer, Candidate of Engineering Sciences Vitaly Trifanov

Saint-Petersburg
2016

Оглавление

Введение

Многопоточные приложения становятся все более и более распространенными. Однако ошибки, связанные с параллельным исполнением потоков, являются наиболее неуловимыми и сложными для отладки. А для такой платформы как Mainframe все осложняется тем, что очень многие многопоточные приложения для нее являются достаточно старыми, плохо задокументированными. Тем не менее, их надо поддерживать. Поэтому необходима автоматизация поиска возможных ошибок, связанных с параллельным исполнением потоков. Например, для состояний гонки. Состояние гонки - это ситуация, когда два или более потоков в параллельной программе одновременно обращаются к одной структуре данных в общей памяти, между этими обращениями нет принудительного упорядочивания по времени, и хотя бы одно из этих обращений является записью[1, 20].

Специфика операционной системы z/OS и приложений для нее (в частности, продуктов, разрабатываемых компанией EMC) накладывает определенные ограничения и предъявляет следующие требования к детектору:

- Детектор должен иметь возможность исследовать большие приложения (от миллиона строк кода);
- Детектор должен быть способным находить гонки в приложениях, написанных на языке ассемблера HLASM;
- Детектор должен сохранять реентерабельность исследуемого приложения.

Предполагается, что имеется доступ к исходному коду исследуемого приложения.

Искать состояния гонки можно с помощью либо динамического, либо статического анализа. В данной работе подробно рассматривается динамический анализ кода.

Задачами данной работы являются:

- Анализ способов сбора информации о блокировках и операциях с памятью в среде ОС z/OS;
- Изучение подходов к поиску состояний гонки и реализация прототипа детектора состояний гонки, удовлетворяющего заданным требованиям.

1. Обзор

1.1. Статический и динамический анализ

Способы поиска состояний гонки разделяют на два основных класса [1, 20]:

- Методы, использующие статический анализ. Рассмотрим несколько наиболее популярных подходов. Система Race-Free Java [2], описывает статическую систему типов и аннотаций, не допускающих состояний гонки, тупиков или даже любых некорректных межпоточных коммуникаций. Детектором RacerX по исходному коду программы строится граф потока управления (Control Flow Graph - CFG), на нем размечаются блокировки, он редуцируется, а затем анализируются обращения к переменным [3]. Детектор RccJava реализует статическую версию динамического алгоритма lockset [6]. Детектор ESC/Java использует метод автоматического доказательства теорем для проверки аннотированной программы на наличие состояний гонки[5]. Все перечисленные методы реализованы применительно к языкам высокого уровня. Существует ряд отличий ассемблерных программ, которые приводят к некоторым проблемам со статическим анализом:

  – Отсутствие функций в языке ассемблера приводит к тому, что код делится на блоки, между которыми совершаются статические или динамические переходы;

  – Наличие в языке ассемблера динамических и относительных переходов приводит к необходимости статического анализа содержимого регистров и памяти. Это, в частности, затрудняет выделение линейных участков кода, потому что динамический переход может иметь целью не метку, а просто некий адрес памяти;

  – Отсутствие типизации в языке ассемблера приводит к тому, что невозможно отличить, содержит ячейка памяти указа-

тель или же нет;

  - Наличие регистров и флагов состояния процессора, сложная семантика инструкций и их большое число приводит к тому, что модель становится очень большой и требует значительного количества ресурсов для анализа программы.

Приведенные особенности языка ассемблера делают построение CFG весьма трудоёмкой задачей. Основными используемыми на практике подходами являются либо использование ограниченной модели без поддержки содержимого регистров, либо использование динамического анализа для обнаружения динамических связей внутри программы. Примерами второго подхода являются анализатор BitBlaze [16], который, в дополнение к использованию динамического анализа, транслирует инструкции ассемблера в промежуточный язык, чтобы избавиться от анализа их сложной семантики, и плагин TrEx для анализатора IDA [19], работающий похожим образом. Кроме того, компанией IBM была разработана программа ASMPUT (Assembler Program Understanding Tool) [8]. Она работает в среде операционной системы Windows, начиная с версии Windows NT. Утилита позволяет вывести на экран граф потока управления исследуемой программы, используя в качестве исходных данных листинг типа ADATA. Перенос соответствующего файла с mainframe на PC осуществляется вручную. ASMPUT не поддерживает достаточно большое количество важных возможностей языка HLASM, в частности, использование индексных регистров в командах перехода. Построенный CFG может быть экспортирован в формате BMP и, в принципе, по нему можно восстановить настоящий CFG. Однако изначально, видимо, не предполагалось, что эта программа будет использоваться для статического анализа кода.

- Методы, использующие динамический анализ. В данном случае имеется «монитор», который получает информацию о событиях, происходящих во время исполнения исследуемой программы. Мо-

нитор может либо записывать эти события в трассу (для дальнейшего post-mortem анализа), либо осуществлять анализ немедленно («на лету» - on-the-fly). Во втором случае есть возможность предотвращения потери данных. Существует значительное количество подходов к динамическому поиску состояний гонки, однако большинство из них не получило большой популярности за исключением двух - алгоритма lockset и алгоритма Лампорта. Классической реализацией первого считается детектор Eraser[15], а второго детектор Djit+[14]. Алгоритм lockset проверяет, пусто ли множество блокировок, которые безусловно захватываются во время работы с разделяемой переменной. Алгоритм Лампорта проверяет выполнение отношений предшествования для операций над разделяемой памятью. Конкретные реализации алгоритмов различаются набором оптимизаций, либо сочетают в себе оба подхода. Примером детектора с оптимизациями алгоритма Лампорта является достаточно известная программа Intel Thread Checker[13]. Она обладает системой фильтров обращений к переменным (фильтр обращения к стековым переменным, хранение состояний переменных, отсеивание всех обращений в таймфрейме кроме первого чтения и первой записи), которая серьезно сокращает потребление памяти алгоритмом Лампорта и увеличивает скорость его работы. Детектор jDRD также эксплуатирует этот алгоритм и дополнительно вводит понятие happens-before контрактов, которые описывают связи определенного вида между вызовами методов. Если между вызовами есть связь, и они произошли в определенном порядке, это гарантирует корректную синхронизацию потоков [20]. В качестве примеров гибридного детектора можно привести работы MultiRace[4], RaceTrack[17], ThreadSanitizer[10].

Сравнение достоинств и недостатков статического и динамического анализа приведено в таблице:

	Достоинства	Недостатки
Статический анализ	+ Не влияет на время исполнения программы + Теоретически позволяет извлечь информацию обо всех отношениях предшествования	- Достаточно медленно работает - Часто встречаются ложноположительные и ложноотрицательные срабатывания - Необходим анализ содержимого памяти и регистров, что является нерешенной задачей
Динамический анализ	+ Можно избежать ложноположительных срабатываний + Не требуется расширений языка или сложного для понимания статического анализа	- Замедляет работу исследуемого приложения - Чувствителен к покрытию тестами - Низкоуровневая работа с памятью приводит к тому, что увеличивается количество операций для трассировки - Необходим статический анализ для распознавания низкоуровневых синхронизирующих примитивов

В условиях недостатка знаний об ассемблерной программе стоит использовать динамический анализ, исходя из следующих фактов:

- Статический анализ большого проекта на языке HLASM так или иначе потребует динамического сбора данных, а общая сложность

реализации статико-динамического анализа достаточно велика;

- Поддержка больших ассемблерных программ часто осуществляется в условиях нехватки знаний о коде, что ограничивает возможности их аннотирования, которое обычно необходимо при статическом анализе;

- Динамический анализ позволяет избежать большинства ложных срабатываний.

1.2. Алгоритмы динамического анализа. Алгоритм Лампорта

Алгоритм Лампорта отслеживает выполнение отношения предшествования (happens-before [11]) для инструкций исследуемой программы.

Описание алгоритма Лампорта:

- Обозначения:
  - $C(y)$ - векторные часы для объекта y. Векторные часы отсчитывают таймфреймы, переключение которых происходит по снятию или установке блокировки;
  - T - индексированное множество потоков;
  - $C(y)(t_i)$ - компонент векторных часов объекта y в потоке t_i;
  - s - синхронизирующий объект (мьютекс, семафор или иной примитив);
  - x - разделяемая переменная.

- Векторные часы инициализируются следующим образом:
  - $\forall i : C(t_i)(t_i) = 1$;
  - $\forall i, \forall j, j \neq i : C(t_i)(t_j) = 0$;
  - $\forall y, \forall i : C(y)(t_i) = 0$.

- Захват синхронизирующего объекта s потоком t:

  $\forall i : C(t)(t_i) = max(C(t)(t_i), C(s)(t_i))$

  Компонент векторных часов потока t в каждом потоке t_i либо остается прежним, либо приобретает значение большего компонента векторных часов синхронизационного объекта s в потоке t_i;

- Освобождение синхронизирующего объекта s потоком t:

  $C(t)(t) = C(t)(t) + 1$

  Компонент векторных часов потока t в самом себе увеличивается (произошел захват, а затем снятие блокировки, в результате чего наступил следующий таймфрейм)

  $\forall i : C(s)(t_i) = max(C(t)(t_i), C(s)(t_i))$

  Компонент векторных часов синхронизирующего объекта s в каждом потоке t_i либо остается прежним, либо приобретает значение большего компонента векторных часов синхронизирующего объекта s в потоке t_i;

- Проверка отношения предшествования при обращении к разделяемой переменной x осуществляется следующим образом:

  $\forall i : C(t)(t_i) \geq C(x)(t_i) \& \exists j : C(t)(t_j) > C(x)(t_j)$

  Это выражение является математической записью идеи о том, что каждое обращение к разделяемой переменной должно происходить в своем таймфрейме, а это эквивалентно исполнению отношения предшествования. Если данное выражение вернет False, то в данном таймфрейме произошло еще одно обращение к данной разделяемой переменной, отношение предшествования не выполнено, и получается, что обнаружено состояние гонки.

  $C(x) = C(t)$

  Векторные часы потока копируются в векторные часы переменной.

Алгоритм основан на идее использования векторных часов для потоков, разделяемых переменных и синхронизирующих объектов [11, 20]. Определение состояния гонки происходит с помощью сравнения показаний часов интересующего нас объекта с показаниями часов потоков исследуемого приложения.

Лампорт определил отношение предшествования следующим образом [11]. События A и B удовлетворяют отношению A→B , если выполнено хотя бы одно из условий:

- A и B находятся в одном потоке, и A происходит до B;
- A - это посылка сообщения одним потоком, а B - это прием этого сообщения другим;
- Если A→B и B→C, то A→C.

Заметим, что этот алгоритм не сможет определить возможную гонку, если порядок исполнения команд потоков случайно выстроился в корректном порядке.

Иными словами, алгоритм не полон, это значит, что некоторые гонки могут быть пропущены, с другой стороны он точен, то есть у него нет ложных срабатываний.

Применительно к ассемблерным программам возникают следующие проблемы:

- Необходим статический анализ для обнаружения синхронизационных примитивов и их идентификации;
- По сравнению с программами на языках высокого уровня ассемблерные программы содержат значительно большее количество операций с памятью. Например, реализация аналога присваивания m[a[i]] = 1 в ассемблере займет несколько команд: загрузка адреса a в регистр, загрузка содержимого a[i] в регистр, загрузка адреса m в регистр, вычисление результирующего адреса и пересылка в него константы 1;

- Так как z/OS не обладает средствами трассировки или обработки по прерыванию чтения из памяти[7], команды чтения должны быть оснащены вызовом трассирующей подпрограммы, что уменьшит скорость исполнения программы, в которой происходит поиск конфликтов чтения-записи.

1.3. Алгоритмы динамического анализа. Алгоритм lockset

Описание алгоритма lockset:

- Обозначения:
  - $Locksheld(t)$ - множество блокировок, захваченных потоком t в момент исполнения текущей инструкции;
  - v - разделяемая переменная;
  - $C(v)$ - множество блокировок разделяемой переменной v. Изначально в нее помещаются все блокировки, синхронизирующие работу с v.
- Для каждого обращения к разделяемой переменной v потока t:
  - $C(v) = C(v) \cap Locksheld(t)$;
  - Если $C(v) = \emptyset$, то, возможно, возникло состояние гонки.

Таким образом, алгоритм lockset проверяет существование блокировок, которые захватываются при любом обращении к переменной [15, 20]. Из-за этого lockset выигрывает у алгоритма Лампорта в производительности и менее чувствителен к расписанию исполнения потоков. Основной сложностью в данном случае является то, что необходимо уметь обрабатывать все возможные синхронизирующие объекты: мьютексы, семафоры и некоторые неявные способы для того, чтобы понять, является ли разделяемая переменная заблокированной или нет. В противном случае, алгоритм будет давать не точный и не полный результат.

Применительно к ассемблерным программам здесь возникают те же проблемы, что и у алгоритма Лампорта.

Гибридный динамический анализ обычно является комбинацией алгоритма Лампорта и алгоритма lockset а применительно к ассемблерным программам имеет те же недостатки.

1.4. Направления развития алгоритмов динамического поиска состояния гонки

Существует несколько основных подходов для оптимизации поиска состояний гонки:

- Определение состояния исследуемых областей памяти:
  - По первому доступу - изначально считаем всю память неразделяемой. При первом обращении потока B к странице, выделенной потоком A - помечаем ее как разделяемую [4];
  - Динамическая гранулярность - память объединяется в мини-страницы, которые затем можно разделять при подозрении, что некоторая часть мини-страницы не является разделяемой [4];
  - Построение полного замыкания для переменной по данным и коду, чтобы понять является ли она разделяемой [12].
- Сжатие анализируемых трасс [12, 21];
- Выявление потенциальных состояний гонки с помощью lockset, а затем отбрасывание ложных срабатываний с помощью алгоритма Лампорта (используется в детекторе MultiRace[4]);
- Модификация алгоритма Лампорта Djit+ [14]:
  - Упрощение проверки исполнения отношений предшествования за счет использования физического времени доступа к переменной;

- Использование следующего свойства отношения предшествования: если A→C, A и B находятся в одном таймфрейме, A и C находятся в разных таймфреймах, то B→C. Это позволяет трассировать только первое чтение и первую запись в разделяемую переменную в текущем таймфрейме. Данная оптимизация используется и в работе [13];
- Поскольку отношение предшествования транзитивно, то стоит проверять только последний доступ к разделяемой переменной на наличие состояния гонки.

- Ключевые особенности модификаций алгоритма Лампорта и алгоритма lockset, предлагаемые в работе ThreadSanitizer[10];
  - Алгоритм lockset был модифицирован следующим образом. Исследуемая программа разбивается на сегменты - последовательные блоки кода, содержащие только операции с памятью. Для каждого потока хранится информация, о том какие блокировки им захвачены. Для каждой ячейки памяти хранится история обращений сегментов к ней. Эти данные позволяют делать полный пересчёт пересечения множества всех возможных блокировок с множеством блокировок, которые были захвачены потоком на момент обращения к этой переменной, для установления наличия состояния гонки. В этом и состоит отличие от классического алгоритма lockset, который хранит для каждой разделяемой переменной готовое пересечение и при обращении к разделяемой памяти обновляет его, строя пересечение с множеством текущих блокировок[15]. Как отмечают авторы работы, динамическое вычисление пересечения множеств лишь незначительно снижает производительность ThreadSanitizer [10];
  - Для того чтобы избежать ложноположительных срабатываний, была добавлена проверка отношения предшествования для кандидатов на состояние гонки, определенных модифи-

кацией алгоритма lockset. Другим примером такой оптимизации может служить детектор Multirace. Кроме того, было введено расширенное отношение предшествования, проверка которого позволяет увеличить производительность, несколько снизив точность. Его использование вместо отношения предшествования не позволяет обнаружить некоторые типы состояний гонки. Анализ выполнения расширенного отношения предшествования опционален;

– Быстрый режим - участок памяти, выделенный в потоке A не считается разделяемым, и операции с ним не анализируются, пока к нему не произойдет обращение из некоторого потока B.

Описанные методы, несомненно, ускорили работу современных детекторов состояний гонки, но применительно к ассемблерным программам, коренным способом они ситуацию не меняют.

1.5. Динамический анализ в среде ОС z/OS: on-the-fly, post-mortem

Существует два основных подхода к реализации динамического детектора состояний гонки: [1, 20]:

- Post-mortem детектор во время исполнения исследуемой программы собирает информацию. По окончании работы приложения она анализируется, и в результате может быть принято решение о тех или иных действиях. Например, о дополнительной инструментации кода и перезапуске;

- Анализ on-the-fly происходит «на лету». Это дает возможность немедленно реагировать на состояние гонки, предотвращая потерю данных. Несомненно, за это приходится платить снижением производительности программы. Чтобы увеличить скорость работы, часто жертвуют точностью анализа, в результате чего возни-

кают ложные срабатывания, или полнотой, иными словами, возможен пропуск некоторых состояний гонки.

Mainframe обладает модулем PER (Program Event Recoder), который позволяет возбуждать прерывания при возникновении тех или иных событий. Например, по исполнению инструкции в указанном диапазоне адресов или обращению к определенной памяти. Обзор документации по z/OS[22] показал, что, к сожалению, не существует официального способа использовать свой обработчик прерываний PER. В операционной системе существует стандартная рутина для этих целей, которая называется FLIH (First Level Interrupt Handler - обработчик прерываний первого уровня). Единственным способом обратиться к возможностям PER по трассировке операций с памятью, является настройка FLIH командой SLIP TRAP. Существуют и другие подходы к получению информации об операциях с памятью. Можно оснащать исследуемые инструкции вызовом пользовательской TRAP или SVC рутины, либо генерацией ошибки для ее последующей обработки, однако это слишком медленные способы[22], по сравнению с PER, в данной ситуации. Таким образом, on-the-fly детектор, построенный на использовании специальных рутин, будет работать крайне медленно, и предпочтительным является post-mortem подход с использованием PER.

В операционной системе z/OS существует стандартное средство для сбора трасс, которое называется GTF (Generalized Trace Facility). Оно может собирать системные сообщения, указанного пользователем типа, в трассу. Например, GTF может зарегистрировать сообщение типа SLIP, созданное FLIH в ответ на прерывание, возбужденное PER. Для настройки PER и управления созданием сообщений существует команда SLIP TRAP. SLIP TRAP с параметром SA позволяет включить генерацию сообщений по сохранению данных в некоторую память. Далее будем называть эту команду и сообщения, к генерации которых она приводит, SLIP SA.

Рассмотрим подробно сбор GTF трассы сообщений типа SLIP в режиме STD+REGS (стандартная информация+регистры).

Сообщения типа SLIP генерируются в следующих случаях:

- IF (instruction fetching) - пошаговая трассировка исполнения отдельного модуля приложения. Подробнее это рассмотрено в работе [18];
- SBT (successful branch) - трассировка успешных переходов внутри программы;
- SA (storage alteration) - трассировка операций записи в определенную область памяти;
- ZAD (zero access detection) - трассировка операций с памятью, в которых адрес сформирован с участием регистра со значением ноль.

GTF, в свою очередь, позволяет собирать трассы в трех режимах:

- MODE=EXT - в данном режиме трассировочная запись сразу сохраняется в набор данных;
- MODE=DEFER - в данном режиме трассировочные записи сохраняются в временном буфере GTF в ОЗУ, размер которого определяется параметром SADMP, а по окончании трассировки буфер сбрасывается в набор данных;
- MODE=INT - в данном режиме трассировочные записи сохраняются в буфере GTF в ОЗУ, а его содержимое обычно просматривают с помощью сохранения SVC DUMP.

Существенная проблема, которая может возникнуть при использовании GTF+SLIP - потеря трассировочных записей. Это происходит в следующих случаях:

- Трассировка происходит в режиме DEFER, и размер SADMP слишком мал для того, чтобы в него поместились все трассировочные записи;
- Трассировка происходит в режиме DEFER или EXT, и размер набора данных слишком мал для того, чтобы в него поместились все записи;

- GTF запущен с приоритетом ниже системного.

Стоит заметить, что GTF использует только первый экстент набора данных для записи и не производит его автоматического расширения. Согласно документации, для сохранения одной трассировочной записи типа SLIP в режиме STD+REGS в набор данных требуется 330 байт. Тестирование, в свою очередь, показало, что в среднем каждой записи требуется еще около 10 байт для создания служебных структур. Исходя из этого, можно спрогнозировать сколько памяти потребуется для трассировки.

Чтобы избежать потери записей, надо убедиться в следующем:

- Выставлен максимальный приоритет для GTF в WLM (Workload Manager - подсистема управления нагрузкой z/OS);
- Задан достаточно большой размер буфера SADMP;
- Используется достаточно большой набор данных для сохранения трассы.

Логично предположить, что трассировка в режиме DEFER будет быстрее, чем в режиме EXT. Приведем статистику производительности GTF. Она была собрана на тестовом стенде с на LPAR пиковой производительностью 50 MIPS. Использовалась следующая методика тестирования:

- Установка SLIP SA на область памяти RANDOMAREA с помощью команды

  SLIP SET,SA, RANGE=(62F9,6385),JOBNAME=SPEEDTST,

  ACTION=TRACE,TRDATA=(STD,REGS),OK,END
- Запуск GTF с опцией DEFER, либо EXT;
- Запуск SPEEDTST;
- Остановка GTF;

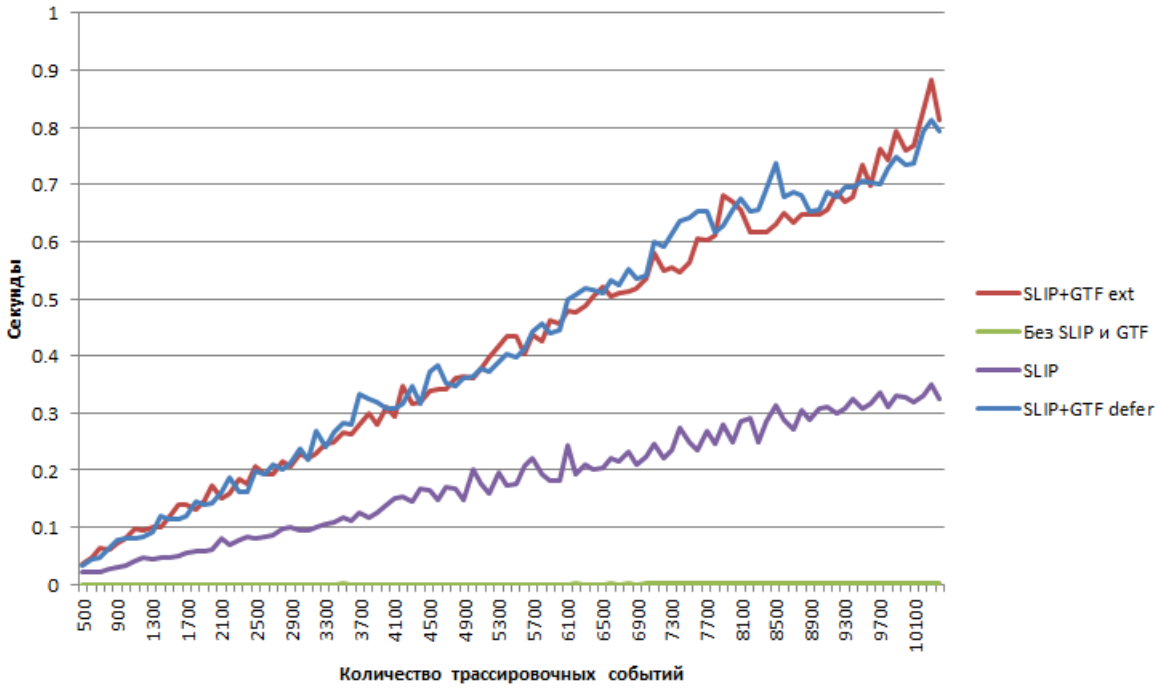

Рис. 1: Производительность стендового приложения в зависимости от вида трассировки

- Проверка количества трассировочных записей утилитой ICETOOL.

Псевдокод программы SPDTST, которая вызывается в JCL скрипте SPEEDTST, приведен в листинге 1.

JOB SPEEDTST содержит в себе вызов программы SPDTST для параметра равного 500. Рис. 1 демонстрирует, что время возрастает линейно, хотя присутствуют некоторые флуктуации. Оказывается, нет существенных различий в скорости работы исследуемой программы с GTF, запущенным в режимах DEFER или EXT. Видимо, на это оказывает влияние хорошо организованный механизм записи данных в набор. Таким образом, можно сделать вывод, что стоит использовать трассировку в режиме EXT, так как она по производительности не отличается от режима DEFER, а записи при этом гарантированно не будут потеряны. Тем не менее, каждая запись в трассу вносит значимый вклад в замедление работы программы. При обработке 10000 событий GTF тратит на создание записей около 0.4 секунды, а FLIH необходимо около 0.3

```
int SPDTST (p1:string) {
  int randomarea[area_size];
  int count = strToInt(p1);
  for (int i = 0; i < step_count; i++){
    start_time = storeCPUClock();
    for (int j = 0; j < count; j++){
      k = random(area_size);
      randomarea[k] = k
    }
    finish_time = storeCPUClock();
    cout << count << '' '' << finish_time-start_time << ''\n'';
    count+=step_size
  }
}
```

Listing 1: Псевдокод программы SPDTST

секунды для того чтобы обработать прерывания PER. При этом время необходимое для исполнения самой программы ничтожно по сравнению с накладными расходами (менее 0.001 секунды). Исходя из подобного теста на целевой машине, можно планировать частоту трассировочных событий и их общее количество, для контроля ожидаемой производительности приложения.

Стоит отметить, что указанное ухудшение производительности актуально только на данном тестовом стенде, на котором трассировочные события генерировались с частотой 1 событие на 20 инструкций кода. В реальности трассировка может работать и быстрее при снижении частоты.

1.6. Datacollider

Не так давно была опубликована работа, описывающая абсолютно новый и достаточно оригинальный способ определения состояния гонки. В ней предлагается следующий метод[9]:

- Выделить набор команд, обращающихся к разделяемой переменной;

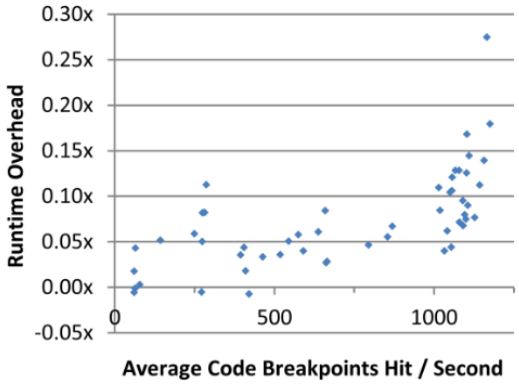

Рис. 2: Зависимость производительности исследуемого приложения от количества сработавших точек останова[9]

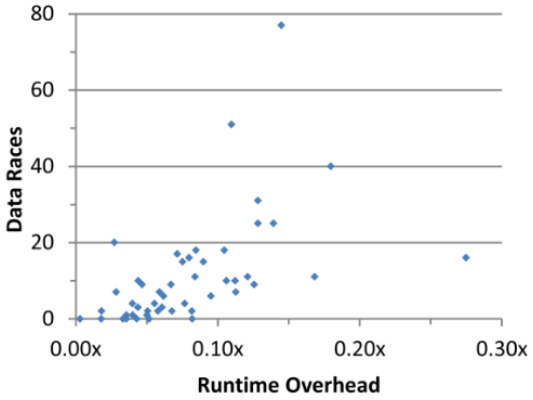

Рис. 3: Зависимость производительности исследуемого приложения от количества обнаруженных состояний гонки[9]

- Добавить после каждой из них вызов инициализирующей рутины, которая:
  - Устанавливает точку останова по данным на область памяти, с которой работала текущая команда;
  - Создает задержку;
  - Снимает точки останова.
- Запустить программу.

Утверждается, что на тестах авторов работы[9] детектор находит около 76% состояний гонки. Ими было принято решение измерять производительность приложения в количестве сработавших точек останова, так как накладные расходы при их отсутствии ничтожно малы, что ясно по рис. 2. Кроме того, на рис. 2 и рис. 3 показано, что производительность исследуемого приложения снижается всего на 5-10% при указанных количествах сработавших точек останова. В то же время, детектор можно держать включенным постоянно, и динамически менять набор исследуемых инструкций, при необходимости меняя число точек останова. Заметим, что для поиска состояний гонки может потребоваться

многократный перезапуск исследуемого приложения с разными параметрами, чтобы как можно сильнее расширить покрытие кода.

Авторы обнаружили, что большая часть найденных детектором состояний гонки - доброкачественные, то есть, не влияющие на работу программы[9]. Оказалось, что большая часть таких состояний гонки может быть эвристически отфильтрована, о чем подробно написано в их работе.

Применительно к ассемблерным программам данный подход характеризуется тем, что:

- Скорость поиска состояний гонки вида чтение-запись и запись-запись увеличивается, поскольку лишь одна из этих команд должна быть оснащена вызовом инициализирующей рутины. Получается, что происходит «обработка» относительно небольшого набора инструкций, но охват кода и охват исследуемой памяти трассой может оказаться достаточно большим. Это зависит от того, как были выбраны оснащаемые инструкции;

- Статический анализ ассемблерного кода не является обязательным для реализации данного подхода. Для использования Datacollider в исследовании ассемблерной программы пользователю достаточно обладать знанием об участках памяти, на которых может произойти вход в состояние гонки.

2. Реализация прототипа детектора

2.1. Общая схема поиска состояний гонки

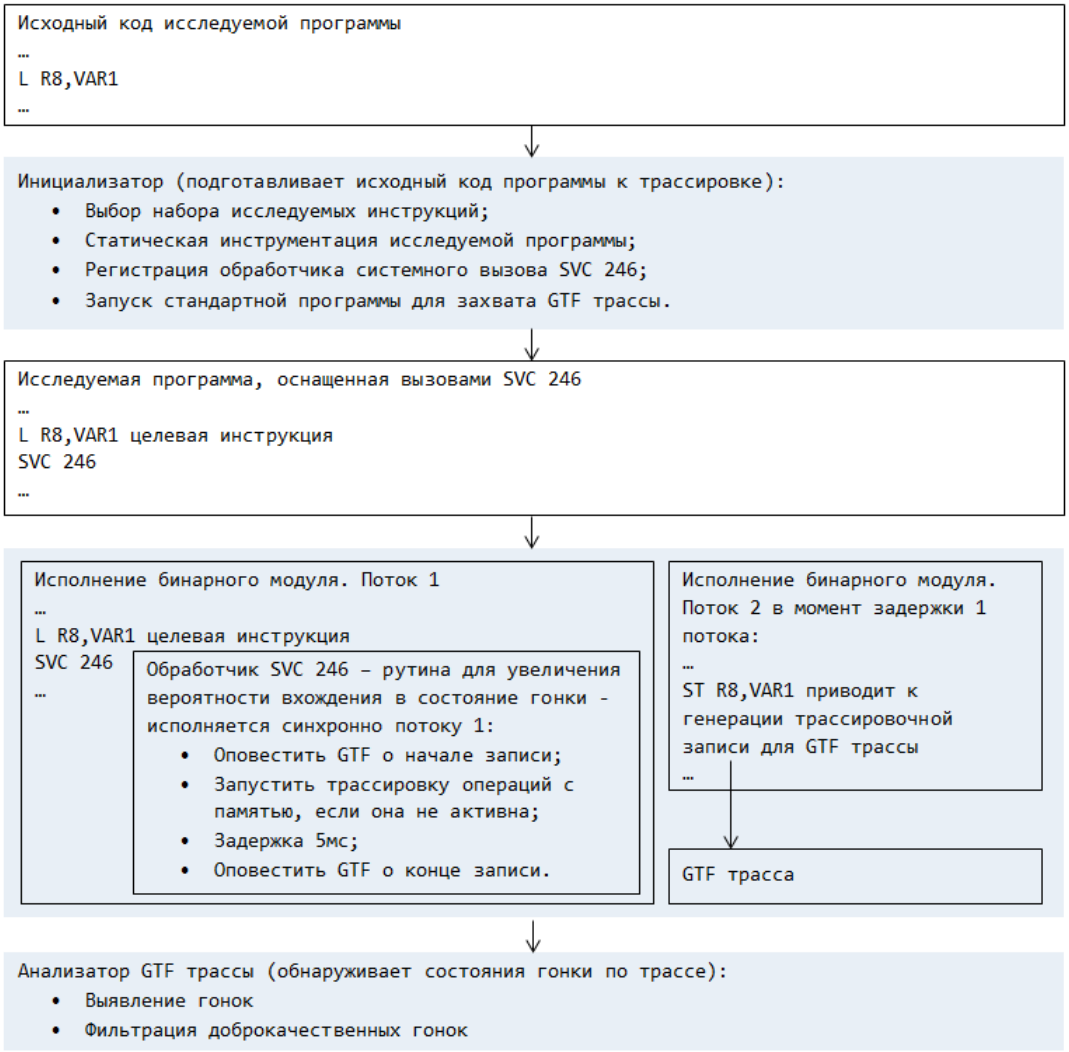

Рис. 4: Общая схема работы детектора

Таким образом, согласно рис. 4 детектор состоит из трех основных частей:

- Инициализатор, занимающийся настройкой окружения и инструментацией исходного кода исследуемого приложения;
- Обработчик системного вызова SVC, обеспечивающий инициализацию трассировки операций с памятью и поиск состояний гонки;

- Анализатор GTF трассы, выявляющий возможные гонки и фильтрующий доброкачественные.

2.2. Инициализатор

Задачи инициализатора:

- Зарегистрировать обработчик SVC 246. Его регистрация подробно описана в работе [22] и не вызывает особой сложности;
- Запустить сборщик GTF трассы, настроенный на сбор событий SLIP SA и событий пользователя;
- Составить множество команд SampleSet, являющихся кандидатами на оснащение вызовом SVC 246. Можно рассмотреть два подхода:
  - Включение в SampleSet команд, которые работают с разделяемой памятью;
  - Включение в SampleSet всех команд и отбрасывание тех, которые работают с локальной памятью и памятью, выделенной в текущем потоке.

  Память может стать разделяемой, только если ее адрес передадут дочернему потоку при его инициализации макросом ATTACH и никаким иным образом. В таком случае, очевидно, что вся разделяемая память будет описываться деревом адресов доступных через список параметров, переданный дочернему потоку. Получается, что достаточно извлечь из параметров макроса ATTACH имя вызываемого модуля и оснастить его инструкции, работающие памятью, адресуемой с помощью списка параметров, последующим вызовом SVC 246. Ключевой проблемой является построение дерева адресов: необходимо обладать знанием полного CFG и def-use цепочек ассемблерной программы. Для построения CFG необходимо в каждой точке модуля знать содержимое регистров, что является серьезной проблемой.

Опишем, в каких случаях обращения к памяти проверять не надо:

Память не стоит проверять, если она располагается в программной секции (CSECT), поскольку реентерабельный модуль не должен содержать инструкций, которые сохраняют значения в ее полях. Исключение составляют команды семейства COMPARE AND SWAP, используемые для реализации синхронизационных примитивов.

Память также не стоит проверять, если она выделена в текущем потоке. Пусть она выделена в потоке X. В таком случае, она может быть разделяемой только тогда, когда адрес этой памяти передается в качестве одного из параметров потоку Y. Допустим, что так случилось для некоторого блока памяти. В таком случае он будет классифицирован как блок памяти подозрительный на наличие состояния гонки при анализе модулей потока Y. Из этого следует, что инструкции модулей потока Y, которые работают с этим блоком, будут оснащены вызовами SVC 246. Во время исполнения исследуемой программы внутри SVC 246 будет инициализирован сбор трассы операций с подозрительной памятью, и операции потока X по ее изменению окажутся зарегистрированными в трассе.

Память можно не проверять, если она представляет собой переменную, используемую для отладки или сбора статистики. Такие переменные могут быть определены только вручную.

- Выбрать случайное подмножество из SampleSet указанного пользователем размера и на нем искать состояние гонки. Это необходимо для управления производительностью. SampleSet может оказаться слишком большим, и это приведет к тому, что частота возникновения трассировочных событий будет слишком высокой, а это вызовет значительное падение производительности;

- Инструментировать исходный код исследуемой программы. Это подразумевает оснащение выбранных инструкций последующим вызовом SVC 246. Необходимо уметь вставлять новые записи в

произвольное место наборов данных с исходными кодами. С этим могут возникнуть проблемы, так как обычно, программные проекты хранятся под управлением системы контроля версий SCLM. Исходный код доступен для редактирования только с помощью стандартного средства SCLM, поэтому инструментация реализована с помощью ISPF макросов, которые, оказалось, можно из него вызывать.

Вторым уровнем отсеивания исследуемых инструкций может быть исключение команд, вызывающих доброкачественные гонки. Согласно авторам статьи [9], доброкачественным состоянием гонки можно назвать такое, которое не влияет на ход исполнения программы. А значит, это могут быть:

- Состояния гонки, использующие переменные, которые не влияют на исполнение программы или необходимы лишь для отладки;
- Состояния гонки, при которых присваивание не меняет значения переменной;
- Состояния гонки на переменной, которая используется для подсчета статистики;
- Состояния гонки на переменной, хранящей битовые флаги. Потоки могут работать с разными битами одного байта;
- Состояния гонки на специальных переменных, отмеченных вручную.

По мнению авторов статьи[9], нет смысла фильтровать сохранение в ячейку памяти, не меняющее ее значения, так как это свойство сохранения может выполняться не всегда. Поэтому пользователю предлагается обратить внимание на подобные случаи. В то же время авторы работы[9] считают, что стоит отсеивать операции над битовыми флагами. В итоге, детектором Datacollider не рассматриваются следующие типы состояний гонки:

- Состояния гонки, использующие специальные переменные, отмеченные вручную;
- Состояния гонки, использующие переменные, которые собирают статистику. Такие переменные в работе [9] ищут по их имени;
- Состояния гонки, использующие переменные, хранящие битовые флаги.

Также предлагается сопоставлять состояниям гонки приоритеты. Состояния гонки, при которых происходит неизменяющая запись, выводятся, но помечены как состояния гонки низкого приоритета.

Стоит отметить, что зачастую правила именования в HLASM программах не определены, это не позволяют эффективно избавляться от переменных для отладки или сбора статистики, поэтому необходимо выделять их вручную.

Поскольку разрабатывается прототип, а статический анализ ассемблерного кода не является целью данной работы, было принято решение, что пользователь сам определяет, какие команды и память будут протестированы на наличие состояния гонки. Чтобы помочь ему в этом был написан ISPF макрос PREPARE, который оснащает вызовом SVC 246 команды, которые работают с полями из списка, указанного в конфигурационном файле. Пользователь перед компиляцией проекта может убрать часть вызовов инициализирующей рутины или добавить свои.

Таким образом, реализация инициализатора для z/OS содержит три программы:

- PREPARE - макрос ISPF редактора, написанный на языке REXX. Просматривает файл с исходным кодом исследуемой программы и оснащает подошедшие под критерии пользователя команды вызовом SVC 246;
- REMSVC - макрос ISPF редактора, написанный на языке REXX. Убирает все вызовы SVC 246;

- INITENV - JCL скрипт, регистрирующий обработчик SVC 246.

2.3. Обработчик

Для инициализации трассировки операций с памятью программа должна исполняться в привилегированном режиме, однако иногда это небезопасно. Для того чтобы избежать наделения программы повышенными правами, было принято решение написать подпрограмму, которая будет вызываться при исполнении команды SVC. Такие рутины исполняются в привилегированном режиме. Они работают как "шлюз" для исполнения действий, которым требуются повышенные права. Рутине при регистрации присваивается номер: от 0 до 255. Номера 0-200 зарезервированы IBM, поэтому можно выбрать любой больший 200, например, номер 246. В нашем случае подпрограмма, вызванная командой SVC 246, оповещает GTF о начале сбора информации для текущей инструкции, инициализирует трассировку операций с памятью, с которой работает эта инструкция, а затем увеличивает вероятность того, что программа войдет в состояние гонки, путем создания искусственной задержки исполнения потока. Особенности окружения SVC рутин подробно описаны в работе [22].

Обработчик прерывания SVC 246 реализует задержку исполнения программы и инициализацию генерации событий SLIP SA. Для того чтобы инициализировать генерацию событий SLIP SA было предложено два метода:

- Предварительная установка SLIP SA на диапазон адресов 0-0, и последующее динамические изменение диапазона с помощью манипуляций над управляющими регистрами. Это не является задокументированной возможностью, но вполне возможно. Исследуемая с помощью PER область памяти обозначается двумя регистрами: CR10 - стартовый адрес и CR11 - последний адрес. К счастью, обработчик прерываний PER не производит дополнительных проверок, поэтому достаточно изменить значения этих управляющих регистров для изменения поведения команды. Од-

нако оказалось, что для каждого потока существуют свои экземпляры регистров управляющих PER, а межпоточные методы манипуляций с ними, увы, неизвестны, поэтому данный подход не может быть использован;

- Динамический вызов команды SLIP SA. В таком случае приходится ввести следующее ограничение: пользователь может определить только один блок памяти, на котором детектор будет пытаться найти состояния гонки. SVC 246 во время своего первого вызова динамически определяет его адрес и вызывает команду SLIP SA. В дальнейшем изменение ее параметров возможно, но будет требовать достаточно много времени, потому что потребуется "отменить" предыдущую команду. Таким образом, было принято решение, не менять параметры SLIP SA до окончания исполнения исследуемого приложения.

Таким образом, предлагается следующий алгоритм работы обработчика:

- Если блокировка не захвачена, то:
  - Захватить блокировку;
  - Извлечь информацию о том, какие данные были загружены или сохранены, об адресе в памяти, и оповестить об этом GTF;
  - Если SLIP TRAP SA не установлен, то установить его;
  - Задержка;
  - Снять блокировку.

Блокировка на обработчик нужна для исключения ситуации, когда два и более потоков инициализируют поиск состояния гонки. Если второй поток вызвал SVC 246, то гонки, которые произойдут после окончания работы подпрограммы, будут пропущены при поиске состояний гонки инициализированном первым потоком. Этого нельзя допускать. Обработчик реализован на языке HLASM.

2.4. Анализатор GTF трассы

```
SLIP S+U          ASCB.... 00FBD280 CPU..... 0001     JOBN.... SPEEDTST TID..... 0001     ASID.... 001A
                  JSP..... SPDTST   TCB..... 008FF048 MFLG.... 0149     EFLG.... 0000     SFLG.... 60
                  DAUN.... 0000     MODN.... SPDTST   OFFS.... 000000C2 IADR.... 1DD000C2 INS..... 5011D0B0
                  EXSIAD.. N/A      EXSINS.. N/A      BRNGH... 00000000 BRNGA... 1DD010B0 BRNGD... 00000000
                  OPSW.... 478D0000 9DD000C6          PIC/ILC. 00040080 PERC.... 20       TYP..... 30
                  PKM..... 00C0     SASID... 001A     AX...... 0000     PASID... 001A     ASC..... 0
                  SA-SPACE 001A
                  GENERAL PURPOSE REGISTER VALUES
                  0-3..... 00000007 00000007 00002710 1DD010B0
                  4-7..... 1DD01880 008FF350 008C2FE0 FD000000
                  8-11.... 80006006 008CBCC8 00000000 008FF350
                  12-15... 9DD00006 1DD01000 80FDDE08 00000096
                  GPR HIGH HALF VALUES
                  0-3..... 00000000 00000000 00000000 00000000
                  4-7..... 00000000 00000000 00000000 00000000
                  8-11.... 00000000 00000000 00000000 00000000
                  12-15... 00000000 00000000 00000000 00000000
                  ACCESS REGISTER VALUES
                  0-3..... 00000000 00000000 00000000 00000000
                  4-7..... 00000000 00000000 00000000 00000000
                  8-11.... 00000000 00000000 00000000 00000000
                  12-15... 00000000 00000000 00000000 00000000
```

Рис. 5: Пример записи в GTF трассе

К доброкачественным отнесем следующие типы состояний гонки, не влияющие на поведение программы:

- Состояния гонки на переменных, хранящих битовые флаги. Такие случаи можно отмечать, проверяя тип инструкции. Например, это могут быть команды test under mask, or, and;
- Состояния гонки, при которых присваивание не меняет значения переменной. Это вполне обнаружимо при сохранении значения регистра в область памяти, содержимое которой было напечатано в GTF трассу. Анализатор может отмечать эти гонки для того, чтобы пользователь обратил на них внимание.

Анализатор просматривает GTF трассу, собранную за время работы исследуемой программы. Изначально предполагалось использовать неформатированную трассу, однако в документации ее структура описана не полностью, что приводит к ряду проблем. Форматировать ее можно с помощью IPCS. Для этого был написан специальный JCL скрипт. Пример записи из форматированной GTF трассы приведен

на рис. 5. В ней печатаются некоторые системные данные и содержимое регистров. В поле MODN содержится имя модуля, где произошло прерывание. В поле OFFS - смещение от его начала. Это позволяет однозначно определить, где произошло прерывание PER. К тому же, имеется возможность отслеживать содержимое первых четырех байтов трассируемой области памяти: поле BRNGD.

Помимо SLIP SA записей в GTF трассу посылаются записи о начале регистрации состояний гонки. Они генерируются вручную в обработчике SVC 246 с помощью макроса GTRACE. Эти записи содержат значение ячейки памяти до инициализации поиска состояний гонки и ее адрес.

Предлагается следующий алгоритм определения состояния гонки:

- Сбросить адрес исследуемой переменной и ее значение в 0;
- Рассматриваем очередную запись из трассы, если таковая есть:
  - Если текущая запись обозначает начало поиска состояний гонки, то сохранить значение исследуемой переменной и ее адрес;
  - Если текущая запись обозначает окончание поиска состояний гонки, то сбросить значение исследуемой переменной и ее адрес в 0;
  - Если текущая запись является результатом сохранения в память, и адрес исследуемой переменной не 0, то проверить, было ли изменено значение по адресу, сохраненному ранее. Если не было изменено, то зарегистрировать гонку как низкоприоритетную, а в противном случае как обычную;
  - Если текущая запись описывает битовую операцию, и адрес исследуемой переменной не 0, то зарегистрировать ее, как состояние гонки на переменной, содержащей битовые флаги.

2.5. Пути развития

У прототипа имеется ряд недостатков, которые должны быть устранены:

- Большое количество конфигурационных файлов, необходимость раздельно запускать компоненты прототипа;
- Отсутствие автоматического выявления подозрительной памяти вследствие нерешенной проблемы выяснения содержимого регистров во время статического анализа;
- Все компоненты исполняются на стороне Mainframe, хотя имело бы смысл анализировать трассу на стороне x86 клиента, потому что это позволяет модифицировать существующие инструменты для статического анализа HLASM программ;
- Вызов ISPF макроса PREPARE для каждого модуля проекта не располагает к анализу крупных программных комплексов. Стоит продолжить изучение способов редактирования исходных текстов, находящихся под управлением SCLM.

Прототип является полуавтоматическим детектором состояний гонки и предполагает наличие у пользователя некоторых знаний о программе и возможной причине ошибки. Тем не менее, он показывает, что подход DataCollider может быть использован для поиска состояния гонки в многопоточных приложениях среды z/OS.

Прототип можно легко модифицировать и получить анализатор потоков данных, что может стать хорошим подспорьем в поддержке существующих продуктов.

2.6. Тестирование и границы применимости

Тестирование происходило на том же стенде, что и изучение производительности GTF+SLIP SA. Детектор был подвергнут стресс тесту на программе TESTAPP, исполняемой в два потока, где RANDOMAREA

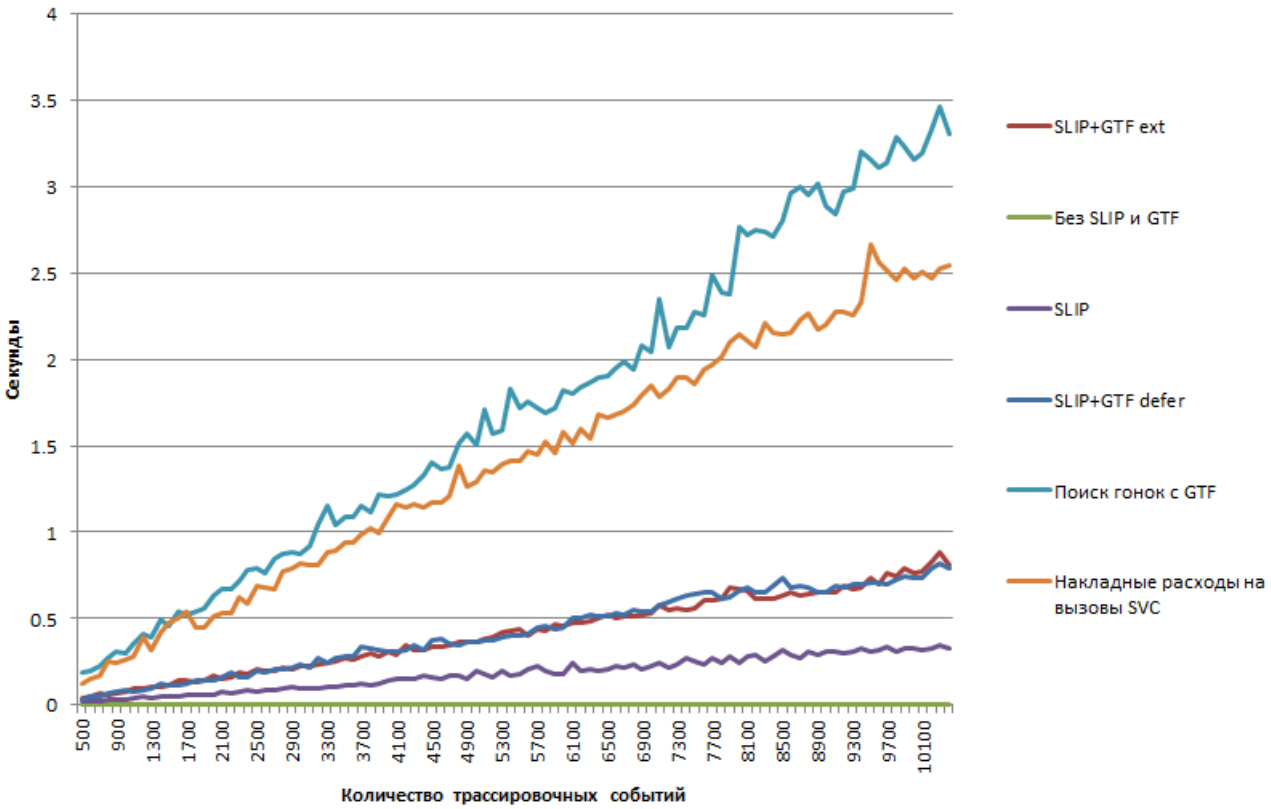

Рис. 6: Производительность приложения

является разделяемой областью памяти, для которой происходил поиск состояний гонки. Псевдокод программы TESTAPP приведен на листинге 2.

Для каждой операции сохранения вызывается SVC рутина, которая при определенных условиях инициализирует поиск состояний гонки. Частота вызова SVC рутины и генерации SLIP SA событий совпадает и достаточно высока - 1/15 событий на инструкцию. Это очень плохой сценарий использования.

Тестирование согласно рис. 6 показало, что накладные расходы на вызовы SVC, ее внутренние вычисления крайне велики и превышают задержку, созданную для детекции состояний гонки. Это дает повод еще раз задуматься о том, как можно было бы организовать инструментацию. Тем не менее, снижение производительности во всех случаях происходит по линейному закону. По графику видно, что имеет

```
int TESTAPP (string p1){
  int randomarea[area_size];
  startThreads(thread,p1,randomarea);
  waitThreads();
  return 0
}
int thread (string p1, int randomarea[area_size]) {
  int count = strToInt(p1);
  for (int i = 0; i < step_count; i++){
    start_time = storeCPUClock();
    for (int j = 0; j < count; j++){
      k = random(area_size);
      randomarea[k] = k;
      SVC 246
    }
    finish_time = storeCPUClock();
    cout << count << '' '' << finish_time-start_time << ''\n'';
    count+=step_size
  }
}
```

Listing 2: Псевдокод многопоточной программы TESTAPP

смысл в первую очередь уменьшать частоту совершения вызовов SVC рутины, так как эти вызовы вносят наибольший вклад в замедление исследуемой программы.

На тестовом LPAR в среднем исполняется около 600000000 инструкций тестового приложения в секунду. Например, чтобы его производительность упала не более чем в 2 раза, частота генерации SLIP SA не должна превышать 0.00005 событий на исполненную инструкцию, и частота вызова SVC 246 не должна превышать 0.000001. Это позволяет приблизительно оценить предполагаемое количество вызовов SVC 246. Таким образом, детектор может быть применен на проектах любого размера, необходимо лишь проконтролировать частоту вызова SVC 246.

В частности, на предмет наличия состояний гонки был исследован один из продуктов компании EMC. Продукт по объему занимает около

600000 строк кода без раскрытия макросов. В тестирующей конфигурации программой использовалось 30 потоков. Исследованию подвергалась часть области разделяемой памяти размером 40 байт, которая хранит общие флаги и указатели на общие списки. Было ограничено количество вставляемых вызовов SVC с увеличением задержки внутри обработчика SVC. Всего было обнаружено три кандидата на состояние гонки:

- Два состояния гонки на разных битовых флагах одного байта. Состояние гонки могло оказаться недоброкачественным, но проверка показала, что взведение данных флагов может быть несинхронизированным.

- Состояние гонки, возникшее на указателе на общий список. Оно оказалось ложным, так как происходила трассировка записей в участок разделяемой памяти размером 40 байт, в котором хранились байтовые флаги и данный указатель. Поиск был инициализирован на одном из байтовых флагов, который не имеет отношения к указателю.

Тестирование показало, что детектор может быть использован на объемных многомодульных многопоточных приложениях. Однако для основательного изучения ассемблерного продукта остро не хватает утилиты для автоматического поиска разделяемых областей памяти. Объем памяти, используемый приложением весьма велик и знанием о свойствах каждого участка в данный момент не обладает ни один человек.

3. Заключение

В результате выполнения бакалаврской работы были решены следующие задачи:

- Произведен обзор способов сбора информации о блокировках и операциях с памятью в ОС z/OS;
- Изучены подходы к поиску состояний гонки;
- Был выбран подход к поиску состояний гонки исходя из критериев его принципиальной реализуемости и применимости к ассемблерным программам, исполняемым в среде z/OS, реализован прототип приложения для поиска состояний гонки и протестирован на реальном проекте.

Прототип детектора отвечает следующим требованиям:

- Детектор способен находить гонки в приложениях, написанных на языке ассемблера HLASM;
- Есть возможность исследовать большие приложения (от миллиона строк кода);
- Не требуется аннотирование исследуемой программы;
- Детектор сохраняет реентерабельность исследуемого приложения.

Продемонстрирована применимость подхода Datacollider к исследованию ассемблерных программ, исполняемых среде ОС z/OS

Список литературы

[1] Beckman Nels E. A Survey of Methods for Preventing Race Conditions // Carnegie Mellon School of Computer Science. — 2006. — URL: `http://www.cs.cmu.edu/~nbeckman/papers/race_detection_survey.pdf` (online; accessed: 12.11.2015).

[2] Boyapati C. Rinard M. A parameterised type system for race-free java programs. — In ACM Conference on Object-Oriented Programming Systems Languages and Applications, 2001.

[3] Dawson Engler Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. — SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[4] Eli Pozniansky Assaf Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. — Concurrency and Computation: Practice Experience - Parallel and Distributed Systems: Testing and Debugging, 2007.

[5] Flanagan C.; Leino K.R.M.; Lillibridge M.; Nelson G.; Saxe R.; Stata J. B. Extended static checking for Java. — Proceedings of the Conference on Programming Language Design and Implementation, 2002. — P. 234–245.

[6] Flanagan C. Freund S. Type-Based Race Detection for Java. — In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, 2000. — P. 219–232.

[7] IBM. z/Architecture Principles of Operation. — IBM, 2012.

[8] IBM. High Level Assembler for z/OS z/VM z/VSE Toolkit Feature User's Guide. — IBM, 2013.

[9] John Erickson Madanlal Musuvathi Sebastian Burckhardt Kirk Olynyk. Effective Data - Race Detection for the Kernel. — USENIX, 2010.

[10] Konstantin Serebryany Timur Iskhodzhanov. ThreadSanitizer – data race detection in practice // WBIA '09 Proceedings of the Workshop on Binary Instrumentation and Applications. — 2009. — URL: http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35604.pdf (online; accessed: 21.03.2016).

[11] Lamport Leslie. Time, Clocks, and the Ordering of Events in a Distributed System // Microsoft Research - Turning ideas into reality. — 1978. — URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf (online; accessed: 12.11.2015).

[12] Netzer R. Miller B. What Are Race Conditions? Some Issues and Formalizations. — In ACM Letters On Programming Languages and Systems, 1994. — P. 74–88.

[13] Paul Sack Brian E. Bliss Zhiqiang Ma Paul Petersen Josep Torrellas. Accurate and efficient filtering for the Intel thread checker race detector. — ASID '06 Proceedings of the 1st workshop on Architectural and system support for improving software dependability, 2006.

[14] Pozniansky E. Schuster A. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. — In Proceedings of The Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003. — P. 179–190.

[15] Savage S. Burrows M. Nelson G. Sobalvarro P. Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. — ACM Transactions on Computer Systems, Vol. 15, No. 4, 1997. — P. 391–411.

[16] Saxena Dawn Song; David Brumley; Heng Yin; Juan Caballero; Ivan Jager; Min Gyung Kang; Zhenkai Liang; James Newsome; Pongsin Poosankam; Prateek. BitBlaze: A New Approach to Computer Security via Binary Analysis // In Proceedings of the 4th International Conference on Information Systems Security

(ICISS). — 2008. — URL: http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf (online; accessed: 21.03.2016).

[17] Yu Y. Rodeheffer T. Chen W. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. — In SOSP, 2005. — P. 221–234.

[18] А.В Щербаков. Трассировка загрузочного модуля в z/OS // - Математико-механический факультет СПБГУ. — 2014. — URL: se.math.spbu.ru/SE/YearlyProjects/2014/YearlyProjects/2014/444/444-Shcherbakov-report.pdf (дата обращения: 21.03.2016).

[19] А.Ю. Тихонов А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода // Труды Института системного программирования РАН. — 2012. — URL: http://cyberleninka.ru/article/n/kombinirovannyy-staticheskiy-i-dinamicheskiy-analiz-binarnogo-ko (дата обращения: 21.03.2016).

[20] В.Ю Трифанов. Динамическое обнаружение состояний гонки в многопоточных Java-программах // - Математико-механический факультет СПБГУ. — 2013. — URL: http://www.math.spbu.ru/user/dkoznov/papers/vtrifanov.pdf (дата обращения: 12.11.2015).

[21] М.Ю. Кудрин А.С. Прокопенко А.Г. Тормасов. Метод нахождения состояний гонки в потоках, работающих на разделяемой памяти. — ТРУДЫ МФТИ Том 1, № 4, 2009.

[22] Р.С Ефремов. Трассировка многомодульного приложения в среде операционной системы z/OS // - Математико-механический факультет СПБГУ. — 2015. — URL: http://se.math.spbu.ru/SE/YearlyProjects/spring-2015/344/344-Efremov-report.pdf/view (дата обращения: 21.03.2016).