DOI: 10.15514/ISPRAS-2023-35(6)-8

Инструмент для поиска гонок по данным RaceHunter

Е.А. Герлиц, ORCID: 0000-0002-1747-075X <*gerlits@ispras.ru*>

Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Гонки по данным - это ошибки в многопоточных программах, возникающие, когда два потока выполняют доступ к одной ячейке памяти без синхронизации. Гонки по данным трудно находить и отлаживать. В этой статье представлен метод динамического поиска гонок по данным RaceHunter, который выполняет мониторинг многопоточной программы, выявляет конфликтующие доступы к памяти и систематически проверяет их на гонки по данным. RaceHunter не выдаёт ложных сообщений о гонках по данным, когда программа использует нестандартные примитивы синхронизации или неизвестные способы синхронизации потоков, и может находить гонки по данным, которые упускают другие подходы. Инструменты динамического поиска гонок по данным могут использоваться для мониторинга длительных выполнений программы либо для проверки относительно непродолжительных выполнений, инициированных тестами. Последнее - основной сценарий использования RaceHunter.

Ключевые слова: гонка по данным; динамический анализ; синхронизация потоков; состояние гонки; параллельная программа; многопоточная программа.

Для цитирования: Герлиц Е.А. Инструмент для поиска гонок по данным RaceHunter. Труды ИСП РАН, том 35, вып. 6, 2023 г., стр. 135–156. DOI: 10.15514/ISPRAS–2023–35(6)–8.

RaceHunter Dynamic Data Race Retector

Gerlits E. A., ORCID: 0000-0002-1747-075X<*gerlits@ispras.ru*>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Tt., Moscow, 109004, Russia

Abstract. Data races are a class of concurrency errors where two threads access a shared memory location without proper synchronization. Data races are hard to reveal and debug. This paper presents RaceHunter - a dynamic data race detection technique which monitors executions of shared memory concurrent programs, discovers pairs of conflicting memory accesses and systematically verifies them for data races. RaceHunter does not report false data races when the target software exploits non-standard synchronization primitives or unknown synchronization protocols and can find data races missed by other techniques. Dynamic data race detectors can monitor continuous, e.g. real-life, program executions or they can verify relatively short program executions, e.g. organized by system tests. The latter is the primary use case scenario for RaceHunter.

Keywords: data race; dynamic data race detection; dynamic analysis; concurrent error; concurrency error; thread synchronization; race condition; concurrent program, shared-memory program, multithreaded program; racehunter.

For citation: Gerlits E. A. RaceHunter dynamic data race detector. *Trudy ISP RAN/Proc. ISP RAS*, 2023, vol. 35, issue 6, pp. 135 – 156. DOI: 10.15514/ISPRAS–2023–35(6)–8.

1. Введение

Настоящая работа посвящена теме верификации программного обеспечения. В качестве объекта верификации выступают многопоточные программы, т.е. параллельные программы с общей памятью. Помимо ошибок, присущих последовательным программам, в многопоточных программах дополнительно возникают новые типы ошибок. Одним из них являются гонки по данным.

Определение 1 (Гонка по данным). *Гонка по данным - это ситуация в многопоточной программе, когда одновременно осуществляются два конфликтующих доступа к памяти.*

Определение 2 (Конфликтующие доступы к памяти). *Два доступа к памяти конфликтуют, если:*

1. *Их осуществляют разные потоки выполнения.*
2. *Участки памяти, к которым происходят обращения, пересекаются.*
3. *Один из доступов выполняет запись.*
4. *Один из доступов выполняется неатомарно.*

Гонка по данным может приводить к некорректным вычислениям и исключительным ситуациям в программе. Поэтому выявление гонок по данным является актуальной задачей.

Для её решения к настоящему времени предложено несколько подходов. В них можно условно выделить два направления - статический анализ [1—3] и динамический анализ. К статическим относятся подходы, в которых код программы исследуется без её выполнения. Динамические подходы ищут гонки по данным во время выполнения программы.

Востребованность обоих направлений обоснована их преимуществами и недостатками. Ключевым преимуществом статического анализа является более полный перебор потенциально возможных выполнений параллельной программы. Однако статические подходы могут выдавать

ложные предупреждения о гонках по данным. В случае большого их количества, обнаружение реально возмозможных гонок по данным существенно затрудняется.

Метод поиска гонок по данным, изложенный в данной работе, относится к динамическому анализу. В существующих динамических подходах можно условно выделить три направления:

1. Happens-before [8] - мониторинг выполнения параллельной программы с отслеживанием частичного порядка на множестве доступов к памяти в этом выполнении. Этот подход реализует, например, инструмент TSan [4].
2. Lock-set - мониторинг выполнения параллельной программы с вычислением множеств захваченных блокировок на доступах к памяти. Этот подход реализует инструмент Eraser [5].
3. Breakpoint-watchpoint - непосредственное обнаружение гонок по данным при помощи точек останова и наблюдения. Этот подход реализуют инструменты DataCollider [6] и KCSAN [7].

Все три направления востребованы ввиду их преимуществ и недостатков. Метод, который изложен в данной работе, использует breakpoint-watchpoint подход, ключевым преимуществом которого является то, что для вердикта о гонке по данным он не отслеживает события синхронизации. Поэтому он применяется к системному программному обеспечению, такому как операционные системы и виртуальные машины, где зачастую используются способы синхронизации, которые не удовлетворяют lock-unlock семантике и/или при применении которых трудно автоматически строить полное отношение happens-before. Например, это барьеры памяти, запрет переключения потоков, запрет прерываний.

1.1 Мотивация

Изначально breakpoint-watchpoint подход был реализован в инструменте DataCollider. Инструмент периодически устанавливает точки останова на некотором количестве инструкций доступа к памяти. Каждая инструкция выбирается случайным образом из множества инструкций доступа к памяти в программе[1]. Обработчик точки останова:

1. Читает данные, к которым обращается инструкция.
2. Устанавливает точку наблюдения на участок памяти, к которому обращается инструкция.
3. Выполняет ожидание в течение небольшого периода времени.
4. Удаляет точку наблюдения.
5. Второй раз читает данные, к которым обращается инструкция.
6. Сообщает о гонке по данным, если сработала точка наблюдения либо изменились данные.
7. Произвольным образом выбирает инструкцию доступа к памяти и устанавливает точку останова на неё.

Предполагается, что точка наблюдения срабатывает, когда происходит доступ к участку памяти, который пересекается с наблюдаемым.

Видно, что в подходе DataCollider доступы к памяти, на которые устанавливаются точки останова, выбираются случайным образом. Следствием этого является то, что инструмент может пропускать инструкции доступа к памяти, которые участвуют в гонке по данным. Следовательно, подход может пропускать гонки по данным.

Рассмотренный подход получил своё развитие в инструменте KCSAN. Точки останова и точки наблюдения инструмент реализует программно на основе статического инструментирования.

[1] Некоторые инструкции предварительно исключаются из множества. Например, те, что несомненно обращаются только к автоматической памяти.

Функции, инструментирующие инструкции доступа к памяти, выполняют действия обработчика точки останова, если они выполняются N-ми по порядку данным ядром процессора, где N генерируется случайным образом.

В подходе KCSAN инструкции доступа к памяти для проверки на гонки по данным также выбираются случайным образом. Следовательно, этот вариант breakpoint-watchpoint подхода также может пропускать гонки по данным.

Гипотеза 1. *Если доступы к памяти для проверки на гонки по данным выбирать не случайным образом, а систематически проверять только конфликтующие доступы к памяти (определение 2), то это позволит более эффективно выявлять гонки по данным.*

1.2 Структура работы

Задача работы сформулирована в разделе 2. Основные этапы подхода RaceHunter последовательно изложены в разделе 3. Расчёт объёма памяти, потребляемой инструментом RaceHunter в худшем случае, дан в разделе 4. О применении RaceHunter на практике рассказано в разделе 5. Сравнение RaceHunter с существующими подходами к поиску гонок по данным выполнено в разделе 6. Выводы по результатам работы сделаны в разделе 7. В разделе 8 обозначены направления будущих исследований. В приложении A приведены вспомогательные алгоритмы - функции, используемые алгоритмами и формулами из основных разделов.

2. Постановка задачи

Предложить динамический подход для поиска гонок по данным в многопоточных программах, который:

- Должен систематически проверять конфликтующие доступы к памяти на гонки по данным.
- Не должен выдавтать ложных предупреждений о гонках по данным при наличии не наблюдаемых событий синхронизации.

3. Подход

На рис. 1 (fig. 1) изображена схема подхода RaceHunter к поиску гонок по данным. Основные этапы подхода следующие:

1. **Инструментирование.** В определённые места кода программы компилятор автоматически вставляет вызовы функций из библиотеки поддержки выполнения RaceHunter. Это позволяет отслеживать события в программе во время её выполнения и реагировать на них.
2. **Мониторинг.** Выполнить программу. Во время выполнения для каждого потока отслеживать и записывать события: доступ к памяти, вызов функции, завершение функции, порождение потока, завершение потока и др. Результат мониторинга - трасса событий в программе.
3. **Анализ трассы**. Результат - множество целей для проверки на гонки по данным, где цель - это пара описателей конфликтующих доступов к памяти.
4. **Провокация гонки по данным.** Для каждой цели повторно выполнить программу, установив программные точки останова на двух доступах к памяти, соответствующих описателям из цели. В обработчиках точек останова ожидать выполнения второй точки останова. Если вторая точка останова сработала, то обнаружена гонка по данным.

Для провокации гонки по данным программа должна быть выполнена повторно. Подразумева-

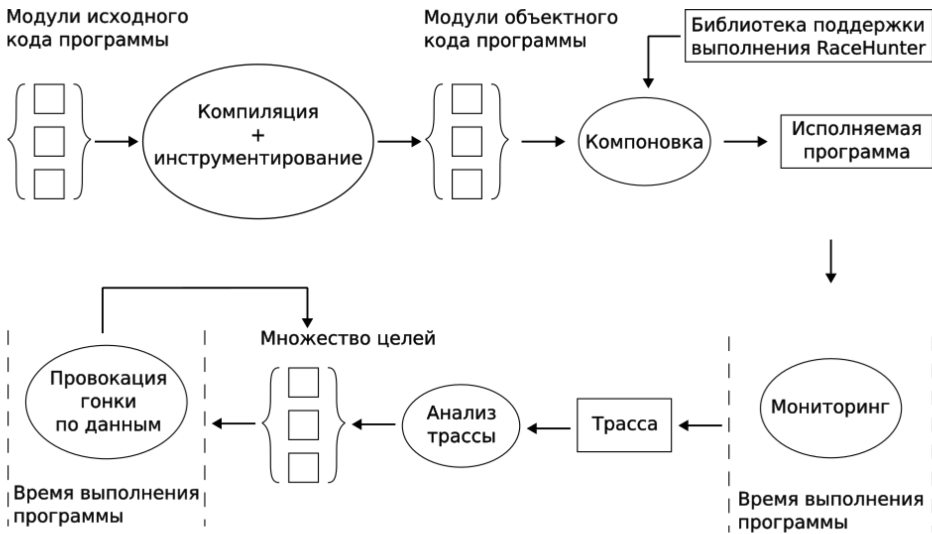

Рис. 1. Схема подхода RaceHunter
Fig. 1. General scheme of RaceHunter approach

ется, что повторные выполнения - это не произвольные выполнения, а попытки воспроизвести то выполнение, которое зафиксировано на этапе мониторинга. Для этого необходимо (но не достаточно):

1. Выполнить программу в том же начальном состоянии. Соответственно, для применения RaceHunter необходимо иметь способ перевода программы в начальное состояние либо её перезапуска.
2. Подать на вход программе те же входные данные.

Из-за непостоянности скорости исполнения потоков ядрами процессора места переключений потоков планировщиком в повторных выполнениях программы могут отличаться. Следствательно, даже при выполнении двух вышеуказанных необходимых условий повторное выполнение многопоточной программы может отличаться от исходного. По мере изложения будем пояснять, как RaceHunter учитывает этот недетерминизм, свойственный многопоточным программам.

Дополнительные факторы недетерминизма, такие как использование случайных чисел и взаимодействие с окружением, по-возможности должны быть устранены на время верификации (детерминированы), чтобы повысить её эффективность.

3.1 Инструментирование

RaceHunter отслеживает события в программе и обновляет своё состояние при их возникновении. Для этого код RaceHunter внедряется в проверяемую программу:

1. На этапе компиляции программы в автоматическом режиме выполняется статическое инструментирование инструкций доступа к памяти, а также точек входа и выхода из функций, т.е. непосредственно перед этими инструкциями вставляются вызовы соответствующих функций из библиотеки поддержки выполнения RaceHunter. Код этой библиотеки должен быть скомпонован с кодом верифицируемой программы.
   Инструментирование реализовано на уровне промежуточного представления LLVM IR [9] компилятора Clang [10]. В целом инструментирование схоже с тем, что выполняет

инструмент TSan [11] и другие инструменты [12]. Основное преимущество текущей реализации инструментирования в RaceHunter состоит в том, что собственные инструкции программы не изменяются, а только добавляются новые инструкции - вызовы функций из библиотеки поддержки выполнения RaceHunter.

2. В качестве оптимизации функции для работы с памятью, такие как $memset$, $memcpy$, $memove$ [13] и др., автоматически подменяются на функции из библиотеки поддержки выполнения RaceHunter. Это позволяет множество обращений к памяти из этих функций моделировать одним обращением. Эту оптимизацию можно отключить, если важно не изменять собственные инструкции программы.
3. Вручную дополняется код некоторых системных функций, например, создание потока.

3.2 Мониторинг

На этапе мониторинга RaceHunter отслеживает ряд событий в программе и записывает их в трассу событий.

Обозначение 1 (Множество целых неотрицательных чисел)**.** $\mathbb{N}_0 = \mathbb{N} \cup 0$.

Определение 3 (Доступ к памяти)**.** *Доступ к памяти - это кортеж*[2] $< I, D, L, R, W, A >$*:*

- $I \in \mathbb{N}_0$ *- адрес инструкции доступа к памяти.*
- $D \in \mathbb{N}_0$ *- адрес начала сегмента данных, к которому осуществляется доступ.*
- $L \in \mathbb{N}$ *- длина в байтах сегмента данных, к которому осуществляется доступ.*
- $R \in 0, 1$ *- чтение (1) или нет (0).*
- $W \in 0, 1$ *- запись (1) или нет (0).*
- $A \in 0, 1$ *- атоматрный доступ (1) или нет (0).*

Обозначение 2 (Множество всевозможных доступов к памяти)**.** *Обозначим* Λ *множество доступов к памяти, соответствующих определению 3.*

Определение 4 (Вызов функции)**.** *Вызов функции - это кортеж* $< I, F >$*:*

- $I \in \mathbb{N}_0$ *- адрес инструкции вызова функции.*
- $F \in \mathbb{N}_0$ *- адрес вызываемой функции.*

Обозначение 3 (Множество всевозможных вызовов функций)**.** $\Gamma = \{< I, F >: I, F \in \mathbb{N}_0\}$.

Определение 5 (Событие)**.** *Событие - это кортеж* $< K, V >$. $K \in 1..4$ *- тип события. Семантика значений* K *следующая: 1 - доступ к памяти, 2 - вызов функции, 3 - завершение функции, 4 - порождение потока.*

$$V = \begin{cases} K = 1 & V \text{ - доступ к памяти по определению 3} \\ K = 2 & V \text{ - вызов функции по определению 4} \\ K = 3 & V \in \mathbb{N}_0 \text{ - адрес функции} \\ K = 4 & V \text{ - идентификатор потока по определению 8} \end{cases}$$

Обозначение 4 (Множество всевозможных событий)**.** *Обозначим* Σ *множество событий, соответствующих определению 5.*

---

[2] Формально кортеж - это конечная последовательность элементов или функция с областью определения $\{i \in \mathbb{N} : i \leq L\}$, где $L \in \mathbb{N}$ - количество элементов кортежа, и областью значений совпадающей с множеством элементов кортежа. Для наглядности в работе на элемент E кортежа t будем ссылаться через $t.E$.

Определение 6 (Трасса потока). *Трасса потока - это кортеж* $< U, E, C >$*:*

- U *- идентификатор потока по определению 8.*
- $E : \mathbb{N} \rightarrow \Sigma$ *- конечная последовательность событий в потоке.*
- $C \in \mathbb{N}_0$ *- количество порождённых потоков.*

Определение 7 (Трасса программы). *Трасса программы* $Trace$ *- это конечное множество трасс потоков по определению 6.*

В определениях 5 и 6 фигурирует идентификатор потока. Подходящий идентификатор должен удовлетворять следующим требованиям:

- Не должно быть двух потоков с одинаковым идентификатором в любом выполнении программы.
- В повторных выполнениях программы на этапе провокации гонки по данным поток должен иметь тот же идентификатор.

Библиотека поддержки выполнения RaceHunter предоставляет интерфейс, с помощью которого инструменту может быть указан способ вычисления идентификатора текущего потока. Например, в качестве идентификатора потока можно взять адрес начальной функции потока, если в программе не может быть двух потоков с одной начальной функцией.

По умолчанию RaceHunter использует свой внутренний способ идентификации потоков. Этот способ исходит из того, что программа начинает исполняться с одним потоком и любой поток может порождать новые потоки.

Определение 8 (Идентификатор потока по умолчанию). *Идентификатор потока - это частично определённая функция* $\mathbb{N} \rightarrow \mathbb{N}$. *Функция моделирует конечную последовательность натуральных чисел.*

Определение 9 (Идентификатор начального потока). $\emptyset \rightarrow \mathbb{N}$.

Алгоритм 1 обрабатывает событие порождения одним потоком другого непосредственно перед фактическим порождением потока в программе. Обработка остальных событий в целом сводится к добавлению информации о них в трассу потока, в котором эти события происходят.

Алгоритм 1. Обработка события порождения потока на этапе мониторинга

Вход: $u_1 : \mathbb{N} \rightarrow \mathbb{N}$ - идентификатор текущего потока
$trace$ - трасса программы по определению 7.
Выход: $trace$ - обновлённая трасса программы. Трассу программы параллельно пополняют все потоки. Остальные переменные алгоритма локальные.

1: $t_1 \in trace : equal(t_1.U, u_1)$ {Трасса текущего потока}
2: $u_2 : Dom(u_1) \cup \{|Dom(u_1)| + 1\} \rightarrow \mathbb{N}$ {Строим идентификатор порождаемого потока}

$$u_2(i) = \begin{cases} \forall i \in Dom(u_1) & u_1(i) \\ i = |Dom(u_1)| + 1 & t_1.C + 1 \end{cases}$$

3: $t_2 =< u_2, \emptyset \rightarrow \Sigma, 0 >$ {Порождаемый поток ещё не имеет событий и не порождал потоков}

4: $t_3 =< t_1.U, t_1.E, t_1.C + 1 >$ {Новая трасса текущего потока}
5: $trace = trace \setminus \{t_1\} \cup \{t_2, t_3\}$ {Обновление трассы программы}

Утверждение 1. *Алгоритм 1 порождает трассу программы, в которой все трассы потоков имеют различные идентификаторы.*

Доказательство. Вычисляемый алгоримом 1 идентификатор потока соответствует единственному пути в генеалогическом дереве потоков от начального потока до порождаемого потока. Узел этого дерева - порядковый номер потока-потомка у потока-родителя. □

Замечание 1. *В данной работе в параллельных алгоритмах будем выделять и нумеровать шаги. Положим, что шаг алгоритма является атомарным действием, которое может изменить состояние алгоритма. Например, в алгоритме 1 ровно 5 шагов. Также положим, что параллельное выполнение шагов алгоритма потоками эквивалентно некоторому их последовательному выполнению [14]. Используемая нами семантика выполнения параллельных алгоритмов соответствует семантике выполнения параллельных алгоритмов в языке PlusCal [15].*

3.3 Анализ трассы

Цель анализа трассы - определить конфликтующие пары доступов к памяти и сформировать из них цели для проверки на гонки по данным. Конфликтующие доступы к памяти RaceHunter ищет по определению 2. Формализуем его.

Определение 10 (Конфликтующие доступы к памяти)**.** *Доступ к памяти a_1 из трассы потока t_1 и доступ к памяти a_2 из трассы потока t_2 конфликтуют, если выполнена конъюнкция следующих условий:*

1. $\neg equal(t_1.U, t_2.U)$
2. $a_1.D = a_2.D \lor (a_1.D < a_2.D \land a_2.D < a_1.D + a_1.L)$
   $\lor (a_2.D < a_1.D \land a_1.D < a_2.D + a_2.L)$
3. $a_1.W = 1 \lor a_2.W = 1$
4. $a_1.A = 0 \lor a_2.A = 0$

Однако данных о доступе к памяти по определению 3 недостаточно для идентификации целевого доступа к памяти в повторном выполнении программы на этапе провокации гонки по данным. К примеру, целевая инструкция доступа к памяти ($a_1.I$, $a_2.I$) может выполняться несколько раз и несколькими потоками. Также равенство адресов данных ($a_1.D$, $a_2.D$) в разных запусках программы в общем случае не гарантируется.

Идентификация целевых доступов к памяти дополнительно осложняется тем, что повторное выполнение многопоточной программы может быть не идентично тому, что наблюдалось на этапе мониторинга. Поэтому описатель целевого доступа к памяти должен не только идентифицировать целевой доступ к памяти, но и допускать вариации в выполнении программы.

Определение 11 (Описатель доступа к памяти)**.** *Описатель доступа к памяти - это кортеж $< U, B, I, S, N >$:*

- U *- идентификатор потока по определению 8.*
- $B \in \{0, 1\}$ *- значение 0 (ложь) - любой поток, значение 1 (истина) - поток с идентификатором* U.
- $I \in \mathbb{N}_0$ *- адрес инструкции доступа к памяти.*
- $S : \mathbb{N} \to \Gamma$ *- верхняя часть стека вызовов функций во время доступа к памяти (функция, вызванная последней, является первым элементом последовательности).* $Dom(S) = \emptyset$ *означает любой стек вызовов функций.*

- $N \in \mathbb{N}$ *- порядковый номер доступа к памяти в трассе потока.*

В определении 11 подразумевается, что N нумерует доступы к памяти, соответствующие остальным условиям, т.е. доступы к памяти из потока с идентификатором U, выполненные инструкцией с адресом I, когда стек потока был равен S.

Обозначение 5. *Обозначим Ψ множество всевозможных описателей доступа к памяти по определению 11.*

Определение 12 (Цель для проверки на гонки по данным)**.** *Цель $Target$ это кортеж $< D_1 \in \Psi, D_2 \in \Psi >$ - пара описателей (по определению 11) доступов к памяти (по определению 3) из трассы программы (по определению 7), которые конфликтуют (по определению 10).*

Алгоритм 2. Построение описателя доступа к памяти

Вход: u - идентификатор потока по определению 8, который выполняет доступ к памяти
i - порядковый номер доступа к памяти в трассе потока с идентификатором u
$trace$ - трасса программы по определению 7
Выход: d - описатель доступа к памяти по определению 11

$d.U = u$
$t_u \in trace : equal(t_u.U, u)$
$d.I = t_u.E(i).V.I$
$d.B = \exists t \in trace : \neg equal(t.U, u) \wedge (\exists j \in Dom(t.E) : t.E(j).K = 1 \wedge t.E(j).V.I = d.I)$
$p = \{j : j \in Dom(t_u.E) \wedge t_u.E(j).K = 1 \wedge t_u.E(j).V.I = d.I\}$ {Доступы, выполненные d.I}

$s = \{(j, st) : j \in p \wedge st = stack(u, j, trace)\}$ {Стеки вызовов функций для доступов из p}
$d.S = top(s(i), \{s(j) : j \in p \wedge j \neq i\})$ {Верхняя часть стека, отличающая i от остальных}

$$d.N = |\{j \in p : j < i \wedge Dom(d.S) \subset Dom(s(j)) \wedge \forall k \in Dom(d.S) : s(j)(|Dom(s(j)| - k + 1) = d.S(k)\}| + 1$$
{Доступы к памяти, выполненные инструкцией d.I ранее доступа i}

Алгоритм 2 строит описатель доступа к памяти по определению 11. Основная идея алгоритма 2 состоит в том, чтобы проверять необходимость добавления очередной порции данных к описателю для идентификации целевого доступа к памяти. Необходимой информацией является только адрес инструкции доступа к памяти.

Замечание 2. *Алгоритм 2 разделяет задачу идентификации целевого доступа к памяти на две подзадачи - идентификацию потока, который выполнил целевой доступ к памяти, и идентификацию доступа к памяти в рамках этого потока. Для некоторых приложений трудно подобрать идентификаторы для потоков, которые бы сохранялись в повторных выполнениях. В таком случае как сам алгоритм 2, так и описатель доступа к памяти (определение 11) можно модифицировать.*

Алгоритм построения множества целей по определению 12 по трассе программы по определению 7 выполняет поиск пар событий доступа к памяти ($K = 1$ по определению 5), которые конфликтуют по определению 10, и строит для каждого доступа из пары описатель по определению 11. В данной работе этот алгоритм решено не приводить с целью экономии места, так как его вклад в прояснение метода RaceHunter несущественен.

3.4 Провокация гонки по данным

Для каждой цели по определению 12 из множества целей, построенного по результатам анализа трассы, программа выполняется повторно. Провокация гонки по данным состоит в организации одновременного выполнения доступов к памяти, указанных в цели.

Определение 13 (Состояние потока)**.** *Состояние потока - это кортеж* $< U, S, N, L, C >$*:*

- U *- идентификатор потока по определению 8.*
- $S : \mathbb{N} \to \Gamma$ *- стек вызовов потока.*
- $N : \Psi \to \mathbb{N}_0$ *- количество обнаруженных доступов к памяти, соответствующих описателю.*
- $C \in \mathbb{N}_0$ *- количество порождённых потоков.*

Определение 14 (Состояние программы)**.** *Состояние программы это кортеж* $< T, A, W, V, X >$*:*

- T *- конечное множество состояний потоков по определению 13*
- $A : \Psi \to \Lambda$ *- функция выдаёт целевой доступ к памяти по его описателю.*
- $W : \Psi \to \{0, 1\}$ *- доступ к памяти, соответствующий описателю, произошёл (*1 *- истина) либо нет (*0 *- ложь).*
- $V \in \{-1, 0, 1\}$ *- вердикт о гонке по данным:* 0 *- нет гонки по данным,* 1 *- гонка по данным,* -1 *- вердикт ещё не получен.*
- $X \in \mathbb{N}$ *- таймаут ожидания.*

Этап провокации гонки по данным реализуется набором обработчиков событий. Обработчик события доступа к памяти представлен алгоритмом 4, а обработчики остальных событий приведены в алгоритме 3. Предполагается, что обрабатываемые события возникают в параллельных потоках. Семантика параллельного выполнения обработчиков событий сформулирована в замечании 1.

Алгоритмы 3 и 4 стартуют в начальном состоянии программы $< \{ts_0\}, a, w, -1, x >$:

- $ts_0 = \{< \emptyset \to \mathbb{N}, \emptyset \to \Gamma, \{(Target.D_1, 0), (Target.D_2, 0)\}, 0 >\}$ - начальное состояние начального потока программы.
- $a = \{(Target.D_1, < 0, 0, 0, 0, 0, 0 >), (Target.D_2, < 0, 0, 0, 0, 0, 0 >)\}$.
- $w = \{(Target.D_1, 0), (Target.D_2, 0)\}$.
- $x \in \mathbb{N}$ - некоторое натуральное число.

В алгоритме 4 таймаут ожидания моделируется циклом, увеличивающим счётчик до достижения им значения $ps.X$ (строки 15-17). В реализации это может быть что-то иное, например, ожидание по времени либо ожидание события.

С одной стороны, величина $ps.X$ должна быть достаточно большой, чтобы дождаться второго доступа к памяти. С другой стороны, избыточное ожидание неоправданно увеличивает время верификации.

Если всё-таки решено повысить вероятность дождаться второго доступа к памяти, то для этого цикл в строках 15-17 алгоритма 4 можно выполнять немного дольше, чем длительность этапа мониторинга для данного теста. Предположим, что имеется функция, выдающая текущее системное время. Тогда длительность выполнения кода - это разница между двумя значениями системного времени. Обозначим длительность этапа мониторинга для теста i через m_i. Для измерения длительности выполнения цикла возьмём $ps.X = c$, где c - некоторое большое целое, например, $c > 10^7$, а остальные проверки из условия цикла исключим. Получим l -

Алгоритм 3. Обработка событий на этапе провокации гонки по данным

Вход: u - идентификатор текущего потока по определению 8
e - событие по определению 5
$target$ - цель по определению 12
ps - состояние программы по определению 14
Выход: ps - новое состояние программы. Состояние программы параллельно обновляют все потоки. Остальные переменные алгоритма локальные.

1: $ts \in ps.T : equal(ts.U, u)$
2: **if** $e.K == 2$ {Вызов функции} **then**
3: $ts_2 =< ts.U, push(ts.S, e.V), ts.N, ts.C >$ {Добавляем вызов в стек вызовов}
4: $ps =< ps.T \setminus \{ts\} \cup \{ts_2\}, ps.A, ps.W, ps.V, ps.X >$
5: **else if** $e.K == 3$ {Завершение функции} **then**
6: $ts_2 =< ts.U, pop(ts.S), ts.N, ts.C >$ {Удаляем последний вызов из стека}
7: $ps =< ps.T \setminus \{ts\} \cup \{ts_2\}, ps.A, ps.W, ps.V, ps.X >$
8: **else if** $e.K == 4$ {Порождение потока} **then**
9: $u_2 : Dom(u) \cup \{|Dom(u)| + 1\} \rightarrow \mathbb{N}$ {Строим идентификатор порождаемого потока}

$$u_2(i) = \begin{cases} \forall i \in Dom(u) & u(i) \\ i = |Dom(u)| + 1 & ts.C + 1 \end{cases}$$

10: $n : \Psi \rightarrow \mathbb{N}_0,\ n = \{(target.D_1, 0), (target.D_2, 0)\}$
11: $ts_2 =< ts.U, ts.S, ts.N, ts.C + 1 >$ {Новое состояние текущего потока}
12: $ts_3 =< u_2, \emptyset \rightarrow \Gamma, n, 0 >$ {Начальное состояние порождаемого потока}
13: $ps =< ps.T \setminus \{ts\} \cup \{ts_2, ts_3\}, ps.A, ps.W, ps.V, ps.X >$
14: **end if**

нижняя оценка длительности выполнения цикла при $ps.X = c$. Тогда $ps.X = \lceil c * m_i / l \rceil$ для теста i.

Отметим, корректное вычисление l подразумевает учёт особенностей вычислительной системы, например, отключение кэшей процессора на время измерения.

4. Оценка размера трассы программы в худшем случае

Для длительных выполнений программы объём памяти для хранения конфликтующих пар доступов к памяти на этапе анализа трассы и состояния программы на этапе провокации гонок по данным пренебрежимо мал по сравнению с размером трассы программы, собираемой на этапе мониторинга. Поэтому сосредоточимся на оценке размера трассы программы в худшем случае.

Пусть для хранения целого числа достаточно b бит. Будем подсчитывать только размер полезных данных без учёта памяти, потребляемой для организации структуры данных.

Трасса программы - это конечное множество трасс потоков по определению 7. В трассу потока входит три элемента - идентификатор потока по определению 8, целочисленный счётчик количества порождённых потоков и последовательность событий в потоке.

Пусть в выполнении n потоков. Наибольший объём памяти для хранения идентификаторов потоков достигается, когда каждый поток порождает ровно один поток: $n(n-1)b/2$ бит. Для хранения счётчиков количества порождённых потоков, очевидно, достаточно bn бит.

По определению 5 всего 4 типа событий. Из них только 3 необходимо сохранять в трассе программы - это доступ к памяти (1), вызов функции (2) и завершение функции (3). Таким

Алгоритм 4. Обработка события доступа к памяти на этапе провокации гонки по данным

Вход: u - идентификатор текущего потока по определению 8
e - событие по определению 5
$target$ - цель по определению 12
ps - состояние программы по определению 14
Выход: ps - новое состояние программы. Состояние программы параллельно обновляют все потоки. Остальные переменные алгоритма локальные.

1: $ts \in ps.T : equal(ts.U, u)$
  $n = ts.N$
2: **if** $e.K == 1$ {Доступ к памяти} **then**
3:   **for all** $j \in Dom(target)$ {Кортеж target - это функция $\{1, 2\} \to \Psi$} **do**
4:     $d_1 = target(j), d_2 = target(|Dom(target)| - j + 1)$
5:     **if** $d_1.I = e.V.I \wedge (d_1.B = 0 \vee equal(d_1.U, ts.U))$
    $\wedge (Dom(d_1.S) = \emptyset \vee isTop(d_1.S, ts.S))$ {Доступ соответствует описателю d_1} **then**
6:       $n_2 : \Psi \to \mathbb{N}_0,\ n_2 = \{(d_1, n(d_1) + 1), (d_2, n(d_2))\}$ {Увеличиваем счётчик}
7:       $ts_2 =< ts.U, ts.S, n_2, ts.C >$ {Новое состояние текущего потока}
8:       $ps =< ps.T \setminus \{ts\} \cup \{ts_2\}, ps.A, ps.W, ps.V, ps.X >$ {Обновляем трассу программы}

9:       **if** $d_1.N = n(d_1) + 1$ {Порядковый номер доступа соответствует d_1} **then**
10:         **if** $ps.W(d_1) \neq 1$ **then** {Первый раз обнаружили целевой доступ}
          $a : \Psi \to \Lambda, a = \{(d_1, e.V), (d_2, ps.A(d_2))\}$ {Сохраняем целевой доступ}
          $w : \Psi \to \{0, 1\}, w = \{(d_1, 1), (d_2, ps.W(d_2))\}$ {Сохраняем, что доступ найден}
          $ps =< ps.T, a, w, ps.V, ps.X >$ {Обновляем трассу программы}
        else
          go to 3 {Проверка доступа на соответствие $target.D_2$ или завершение}
        end if
11:         **if** $ps.W(d_2) = 1 \wedge conflict(ps.A(d_1), ps.A(d_2)) \wedge ps.V = -1$ {Гонка} **then**
12:           $ps =< ps.T, ps.A, ps.W, 1, ps.X >$ {Сохраняем в трассе, что гонка найдена}
          go to 22 {Завершение - обнаружена гонка по данным}
13:         **end if**
14:         $i = 0$ {Счётчик цикла ожидания доступа к памяти, соответствующего d_2}
15:         **while** $i < ps.X \wedge ps.V = -1 \wedge$
        $(\exists t \in ps.T : \neg equal(t.U, ts.U) \wedge Dom(t.S) \neq \emptyset)$ **do**
16:           $i = i + 1$
17:         **end while**
18:         **if** $ps.V = -1$ **then** $ps =< ps.T, ps.A, ps.W, 0, ps.X >$ **end if** {Гонки нет}
19:       **end if**
20:     **end if**
21:   **end for**
22: **end if**

образом, для хранения типа события с тремя возможными значениями достаточно 2 бита.

Рассмотрим событие доступа к памяти по определению 5. Нет необходимости сохранять тип операции (чтение или запись) и сведения об атомарности в трассу программы для каждого доступа к памяти. Эти данные могут быть извлечены на этапе инструментирования из LLVM IR, ассоциированы с адресом инструкции в машинном коде и далее с доступом к памяти в трассе программы.

Длина сегмента данных, к которому осуществляется доступ, может вычисляться во время вы-

полнения (случай, когда множество доступов к памяти из функций $memset$, $memcpy$, $memove$ и других моделируется одним доступом к памяти). Таким образом, для хранения события доступа к памяти по определению 5 достаточно $3b + 2$ бит на адрес инструкции, адрес данных, длину сегмента данных и тип события.

Для хранения события вызова функции достаточно $2b + 2$ бит на адрес инструкции вызова функции, адрес вызываемой функции и тип события.

В предположении, что событие завершения функции в трассе программы относится к предыдущему событию вызова функции, приходим к заключению, что хранить адрес завершаемой функции нет необходимости. Следовательно, для хранения события завершения функции достаточно 2 бита (тип события).

Итого, для выполнения программы с n потоками, m доступами к памяти и k вызовами (и завершениями) функций в худшем случае достаточно $n(n+1)b/2 + m(3b+2) + k(2b+4)$ бит.

5. Эксперименты с RaceHunter

Инструмент RaceHunter разрабатывался с целью обнаруживать гонки по данным, которые могут быть упущены существующими breakpoint-watchpoint подходами. Полноценное экспериментальное исследование, подтверждающее эффективность RaceHunter, и индустриальное применение инструмента впереди. Цель экспериментов с RaceHunter, выполненных в настоящей работе, заключается в практическом подтверждении способности RaceHunter выявлять гонки по данным.

В качестве целевого программного обеспечения для экспериментов была выбрана операционная система реального времени [16]. Эта операционная система реализует стандарт ARINC 653. В этом стандарте сформулированы требования к программному интерфейсу, который операционная система предоставляет приложениям. В частности, этот программный интерфейс включает набор функций для обмена сообщениями между потоками. В экспериментах в реализацию этих функций вносилась ошибка, приводящая к гонке по данным, и проверялось, что RaceHunter обнаруживает эту гонку по данным.

Для каждого эксперимента был разработан тест. В тесте два потока с одинаковым приоритетом выполняются параллельно разными ядрами процессора. Один поток осуществляет посылку сообщения, а второй неограниченно ожидает прихода сообщения. Тесты были разработаны таким образом, чтобы две инструкции доступа к памяти, образующие гонку по данным, реально выполнялись.

Эксперимент 1. *Объявить глобальную неатомарную переменную целочисленного типа. В реализации функции посылки сообщения записать в переменную значение. В реализации функции приёма сообщения прочитать значение переменной. Доступ к переменной осуществлять без организации взаимного исключения.* ***Результат:*** *обнаружена гонка по данным.*

Эксперимент 2. *В реализации функции приёма сообщения переместить операцию захвата примитива синхронизации так, чтобы чтение одной из разделяемых переменных выполнялось без взаимного исключения.* ***Результат:*** *обнаружена гонка по данным.*

Эксперимент 3. *В реализации функции посылки сообщения переместить операцию захвата примитива синхронизации так, чтобы чтение одной из разделяемых переменных выполнялось без взаимного исключения.* ***Результат:*** *обнаружена гонка по данным.*

В табл. 1 приведены основные показатели статистики верификации при помощи RaceHunter, полученной по результатам экспериментов. Операционная система выполнялась в эмуляторе qemu [17]. Эмулировалась система на кристалле с 4 ядрами по 100 МГц и 128 Мбайт оперативной памяти.

Табл. 1. Статистические показатели верификации (средние значенния)
Table 1. Some statistical indicators of verification (average values)

Величина	**Значение**
Количество потоков в тесте	4
Количество доступов к памяти	696
Количество вызовов функций	129
Количество конфликтующих пар доступов к памяти	50
Объём памяти, потреблённой RaceHunter	$\sim$ 14 Кб
Время выполнения теста без RaceHunter	$\sim$ 2 сек.
Время верификации с RaceHunter	$\sim$ 93 сек.

Отметим, текущая редакция RaceHunter создавалась с целью апробировать новый подход к поиску гонок по данным. Поэтому её можно считать рабочим прототипом. От индустриального инструмента она отличается неоптимальным потреблением памяти и алгоритмами, при разработке которых важнейшим критерием была простота реализации.

6. Сравнение с существующими подходами

RaceHunter - это инструмент динамического анализа, который создавался с целью обнаруживать гонки по-данным, упускаемые существующими breakpoint-watchpoint подходами. Оказалось, RaceHunter способен обнаруживать гонки по данным, упускаемые lock-set и happens-before подходами.

Утверждение 2. *Существуют гонки по данным, которые может пропустить lock-set подход, но обнаруживает RaceHunter.*

Доказательство. Для доказательства достаточно привести пример программы и соответствующей гонки по данным в ней. Программа из листинга 1 содержит гонку по данным по адресу переменной a.

```
int a = 0, b = 0;
mutex ma, mb;

void t1(void) {                    void t2(void) {
    lock(&ma);                         lock(&mb);
    a = 1;                             if(b == 1) {
    unlock(&ma);                           unlock(&mb);
    lock(&mb);                             lock(&ma);
    b = 1;                                 a = 2;
    unlock(&mb);                           unlock(&ma);
}                                      }
                                       else {
                                           unlock(&mb);
                                           a = 3;
                                       }
                                   }
```

Листинг 1. Псевдокод программы с гонкой по данным
Listing 1. Pseudocode of a program with a data race

Lock-set подход пропускает эту гонку по данным на последовательном выполнении потоков

t_1, t_2, так как в этом случае t_2 выполняет ветку if, в которой доступ к a выполняется с захватом блокировки.

RaceHunter обнаруживает эту гонку на последовательном выполнении t_1, t_2, потому что видит доступ к a из двух потоков на этапе мониторинга и ожидает перед $t_1 : a = 1$ на этапе провокации гонки по данным, что приводит к выполнению в t_2 ветки $else$, в которой доступ к a выполняется без захвата блокировки. □

Утверждение 3. *Существуют гонки по данным, которые может пропустить happens-before подход, но обнаруживает RaceHunter.*

Доказательство. Для доказательства достаточно привести пример программы и соответствующей гонки по данным в ней. Программа из листинга 2 содержит гонку по данным по адресу переменной a.

```
int a = 0;
atomic int b = 0;
atomic int c = 2;

void t1(void) {                    void t2(void) {
    a = 1;                             if(b != c)
    b = 1;                                 a = 2;
}                                  }
```

Листинг 2. Псевдокод программы с гонкой по данным
Listing 2. Pseudocode of a program with a data race

Happens-before подход пропускает эту гонку по данным на последовательном выполнении потоков t_1, t_2. На этом выполнении пара операторов $(t_1 : b = 1,\ t_2 : b \neq c) \in happens{-}before$. Также в соответствии с программным порядком $(t_1 : a = 1,\ t_1 : b = 1) \in happens - before \wedge (t_2 : b \neq c,\ t_2 : a = 2) \in happens - before$. Следовательно, на последовательном выполнении потоков t_1, t_2 по транзитивности happens-before $(t_1 : a = 1,\ t_2 : a = 2) \in happens - before$.

RaceHunter обнаруживает эту гонку на последовательном выполнении t_1, t_2, потому что перед $t_1 : a = 1$ успешно дожидается $t_2 : a = 2$. □

Усиление способности выявлять гонки по данным в подходе RaceHunter достигается за счёт увеличения времени верификации (в основном из-за повторных запусков) и неограниченного потребления памяти на этапе мониторинга. Табл. 2 содержит результаты сравнения ключевых характеристик динамических подходов для поиска гонок по данным. Обозначения, используемые в таблице: RH - RaceHunter, HB - happens-before, LS - lock-set, BW - breakpoint-watchpoint.

Табл. 2. Сравнение RaceHunter с существующими динамическими подходами к поиску гонок по данным
Table 2. RaceHunter vs. existing dynamic data race detection techniques

Характеристика	**RH**	**HB**	**LS**	**BW**
Не выдаёт ложных гонок из-за не наблюдаемой синхронизации	+	-	-	+
Не упускает гонки из-за случайных проверок доступов к памяти	+	+	+	-
Потребляет фиксированный объём памяти	-	+	+	+
Не тратит время на повторные выполнения программы	-	+	+	+
Находит гонки, упускаемые другими подходами	+	+	+	+

Можно провести некоторую аналогию между подходом RaceHunter и фаззингом [18—20]:

- Фаззер начинает свою работу с некоторого начального множества входных данных программы. RaceHunter в свою очередь применяется к некоторому множеству выполнений многопоточной программы.
- Фаззер мутирует входные данные программы. RaceHunter в свою очередь провоцирует новые выполнения многопоточной программы.

Однако в отличие от RaceHunter фаззер добавляет некоторые мутированные входные данные к начальному множеству входных данных для получения новых входных данных из них.

Идея двухэтапного динамического метода поиска гонок по данным, в котором в первом выполнении определяются возможные гонки по данным, а в последующих выполнениях эти гонки по данным подтверждаются путём воспроизведения, была ранее сформулирована и реализована в работе [21]. Алгоритм второго этапа в методе [21]:

- Организует некоторый последовательный порядок выполнения инструкций в программе, случайным образом выбирая поток для выполнения очередной инструкции в управляемом планировщике выполнения потоков.
- Идентифицирует доступы к памяти исключительно по оператору программы, который его выполняет. Поэтому приостанавливает каждый поток перед выполнением любого из двух целевых операторов программы (на которых обнаружена гонка по данным на первом этапе метода) до тех пор, пока гонка по данным не будет подтверждена (конфликт с одним из ранее приостановленных доступов к памяти), либо все потоки не остановятся.

В результате действий алгоритма второго этапа, которые не нацелены на воспроизведение выполнения программы, полученного по результатам первого этапа, может получиться новое выполнение программы, на котором гонка по данным, обнаруженная на первом этапе, не воспроизведётся. Однако проверка всех доступов к памяти, выполняемых целевыми операторами программы, может показать большую эффективность, чем подход RaceHunter, в случаях, когда описатель доступа к памяти по определению 11 (или иному) не идентифицирует целевой доступ к памяти в повторном выполнении программы из-за её недерминированного выполнения.

Дополнительно, управление планировщиком потоков на втором этапе метода [21] существенно ухудшает масштабирование метода на длительные выполнения программы либо на большое количество выполнений программы.

Метод [21] изначально разрабатывался для параллельных Java программ с общей памятью (shared memory concurrent Java programs). Позже в работе [22] авторы реализовали схожий метод для параллельных программ с распределённой общей памятью (distributed shared memory concurrent programs).

В методе [22] доступы к памяти также идентифицируются исключительно по оператору программы, который их выполняет. Алгоритм второго этапа отличается от метода [21]. Задачи на вычислительных узлах выполняются свободно за исключением обработки двух целевых операторов доступа к памяти. Обработчик последовательно:

1. Выполняет проверку на гонку по данным с одной из ранее приостановленных задач.
2. Если проверка не подтвердила гонку по данным, то приостанавливает текущую задачу на небольшой период времени. Точнее задача приостанавливается с некоторой вероятностью. Приостанавливая очередную задачу, обработчик понижает вероятность приостановки последующих задач.

Алгоритм второго этапа в методе [22] также может не подтверждать гонки по данным, обнаруженные первым этапом, из-за того, что целевые доступы к памяти идентифицируются не уникально, в выполнение программы вносятся задержки, не нацеленные на воспроизведение выполнения, полученного на первом этапе, и задачи приостанавливаются на целевых

операторах не всегда, а с некоторой вероятностью.

По аналогии с базовым методом [21] метод [22] может показать большую эффективность, чем подход RaceHunter в случае, когда описатель доступа к памяти по определению 11 не идентифицирует целевой доступ к памяти на этапе провокации гонки по данным ввиду недетерминированного выполнения программы. А уменьшение вероятности проверки доступов к памяти на гонки по данным на втором этапе метода [22] нацелено на улучшение масштабируемости метода.

7. Выводы

В настоящей работе предложен новый метод динамического поиска гонок по данным RaceHunter. RaceHunter развивает идею о непосредственном обнаружении гонок по данным при помощи точек останова и наблюдения, ранее реализованную в инструментах DataCollider [6] и KCSAN [7], идеей систематически выявлять и проверять конфликтующие доступы к памяти на гонки по данным.

Этап проверки пар конфликтующих доступов к памяти на гонки по данным (этап провокации гонки по данным) отличается от методов [21] и [22] попыткой идентифицировать целевую пару доступов к памяти в повторном выполнении программы уникальным образом, учитывая однако возможное недетерминированное выполнение многопоточной программы при помощи описателя доступа к памяти.

RaceHunter не выдаёт ложных предупреждений о гонках по данным при наличии не наблюдаемых событий синхронизации, поэтому может применяться к программному обеспечению, в котором используются любые способы синхронизации потоков. Это особенно актуально для системного программного обеспечения, такого как операционные системы и виртуальные машины. В частности, уже имеется опыт применения RaceHunter к операционной системе реального времени [16].

Усиление способности обнаруживать гонки по данным в подходе RaceHunter достигается в том числе за счёт повторных выполнений программы. Следствием этого является увеличение времени верификации.

Кроме того, подход RaceHunter не обеспечивает фиксированное потребление памяти, поэтому не нацелен на очень длительный (в пределе бесконечный) мониторинг выполнения программы. По этой причине RaceHunter более подходит для использования в системах непрерывной интеграции и разработки (Continous Integration and Development) совместно с тестами, которые задействуют параллельное выполнение потоков в программе.

8. Направления будущих исследований

Актуальным представляется экспериментальное исследование эффективности инструмента RaceHunter в сравнении с существующими динамическими подходами к поиску гонок по данным, в особенности с [7] и [22].

В текущей редакции RaceHunter конфликтующие пары доступов к памяти ищутся по определению 10. Применение анализа потоков, подобного [23], анализа блокировок, подобного [5], и др. потенциально способно сократить количество целей для этапа провокации гонок по данным и, следовательно, ускорить верификацию. Для этих видов анализа актуально оценить ускорение верификации и исследовать их влияние на эффективность выявления гонок по данным.

Подходу RaceHunter необходимо повторно выполнять программу на этапе провокации гонок по данным. Актуальным видится исследование применимости в RaceHunter существующих методов воспроизведения выполнения многопоточных программ [24—27].

Список литературы / References

[1]. M. Naik, A. Aiken и J. Whaley. Effective static race detection for java. В *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, страницы 308—319, 2006.

[2]. P. Andrianov и V. Mutilin. Scalable thread-modular approach for data race detection. В *International Workshop on Frontiers in Software Engineering Education*, страницы 371—385. Springer, 2019.

[3]. P. Pratikakis, J. S. Foster и M. Hicks. Locksmith: practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):1—55, 2011.

[4]. K. Serebryany и T. Iskhodzhanov. Threadsanitizer: data race detection in practice. В *Proceedings of the Workshop on Binary Instrumentation and Applications*, страницы 62—71, 2009. DOI: 10.1145/1791194.1791203.

[5]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro и T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391—411, 1997. DOI: 10.1145/265924.265927.

[6]. J. Erickson, M. Musuvathi, S. Burckhardt и K. Olynyk. Effective data-race detection for the kernel. В *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[7]. The kernel concurrency sanitizer, The Linux Kernel Organization. URL: https://docs.kernel.org/dev-tools/kcsan.html (дата обращения 26.10.2023).

[8]. L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. 2019, страницы 179—196. DOI: 10.1145/3335772.3335934.

[9]. LLVM language reference manual. URL: https://llvm.org/docs/LangRef.html (дата обращения 20.11.2023).

[10]. Clang: a c language family frontend for llvm. URL: https://clang.llvm.org (дата обращения 20.11.2023).

[11]. K. Serebryany, A. Potapenko, T. Iskhodzhanov и D. Vyukov. Dynamic race detection with llvm compiler: compile-time instrumentation for threadsanitizer. В *International Conference on Runtime Verification*, страницы 110—114. Springer, 2011.

[12]. D. Marino, M. Musuvathi и S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. В *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, страницы 134—143, 2009.

[13]. Information technology Portable Operating System Interface (POSIX ®) Base specifications. Standard, International Organization for Standardization, Geneva, CH, сент. 2009.

[14]. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690—691, 1979.

[15]. L. Lamport. The pluscal algorithm language. В *International Colloquium on Theoretical Aspects of Computing*, страницы 36—60. Springer, 2009.

[16]. V. Cheptsov и A. Khoroshilov. Robust resource partitioning approach for arinc 653 rtos.

[17]. F. Bellard. Qemu, a fast and portable dynamic translator. В *USENIX annual technical conference, FREENIX Track*, том 41, страница 46. Califor-nia, USA, 2005.

[18]. C. Holler, K. Herzig и A. Zeller. Fuzzing with code fragments. В *21st USENIX Security Symposium (USENIX Security 12)*, страницы 445—458, 2012.

[19]. Libfuzzer – a library for coverage-guided fuzz testing. URL: https://llvm.org/docs/LibFuzzer.html (дата обращения 26.10.2023).

[20]. S. Sargsyan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin и S. Kurmangaleev. Isp-fuzzer: extendable fuzzing framework. В *2019 Ivannikov Memorial Workshop (IVMEM)*, страницы 68—71, 2019. DOI: 10.1109/IVMEM.2019.00017.

[21]. K. Sen. Race directed random testing of concurrent programs. В *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, страницы 11—21, 2008.

[22]. C.-S. Park, K. Sen, P. Hargrove и C. Iancu. Efficient data race detection for distributed memory parallel programs. В *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, страницы 1—12, 2011.

[23]. J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. В *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, страницы 24—33, 1991.

[24]. Leblanc. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 100(4):471—482, 1987.

[25]. P. Dovgalyuk. Deterministic replay of system's execution with multi-target qemu simulator for dynamic analysis and reverse debugging. В *CSMR*, страницы 553—556, 2012.

[26]. R. H. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. В *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, страницы 1—11, 1993.

[27]. D. F. Bacon и S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. *ACM SIGPLAN Notices*, 26(12):194—206, 1991.

Информация об авторе / Information about author

Евгений Анатольевич ГЕРЛИЦ — научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: методы контроля и обеспечения качества программного обеспечения, методы динамической и статической верификации и анализа программ, формальные методы.

Evgeny GERLITS — researcher at the Software Engineering Department of ISP RAS. Main research interests: software quality control and assurance, dynamic and static software verification and analysis, formal methods.

A. Вспомогательные алгоритмы

Алгоритм 5. Проверка, конфликтуют ли два доступа к памяти, в предположении, что они выполняются из разных потоков

Вход: $a_1, a_2 \in \Lambda$

Выход: истина, если доступы к памяти конфликтуют, иначе ложь

function conflict(a_1, a_2)

return $(a_1.W = 1 \vee a_2.W = 1)$
$\wedge (a_1.A = 0 \vee a_2.A = 0)$
$\wedge (a_1.D = a_2.D \vee (a_1.D < a_2.D \wedge a_2.D < a_1.D + a_1.L)$
$\vee (a_2.D < a_1.D \wedge a_1.D < a_2.D + a_2.L))$

end function

Алгоритм 6. Добавление элемента в конец последовательности

Вход: $s : \mathbb{N} \to \Theta$, где Θ - произвольное множество
$v \in \Theta$ - добавляемый элемент

Выход: новая последовательность с добавленным элементом

function push(s, v)

$s_2 : Dom(s) \cup \{|Dom(s)| + 1\} \to \Theta$

$$s_2(k) = \begin{cases} \forall k \in Dom(s) & s_2(k) = s(k) \\ k = |Dom(s)| + 1 & v \end{cases}$$

return s_2

end function

Алгоритм 7. Удаление конечного элемента последовательности

Вход: $s : \mathbb{N} \to \Theta$, где Θ - произвольное множество

Выход: новая последовательность с удалённым элементом

function pop(s)

$s_2 : Dom(s) \setminus \{|Dom(s)|\} \to \Theta, \forall k \in Dom(s) \setminus \{|Dom(s)|\} : s_2(k) = s(k)$

return s_2

end function

Алгоритм 8. Равенство конечных последовательностей натуральных чисел

Вход: $a : \mathbb{N} \to \mathbb{N}$, $b : \mathbb{N} \to \mathbb{N}$

Выход: истина, если последовательности равны, иначе ложь

function equal(a, b)

return $Dom(a) \subset Dom(b) \wedge Dom(b) \subset Dom(a) \wedge \forall i \in Dom(a) : a(i) = b(i)$

end function

Алгоритм 9. Восстановление стека вызовов функций в момент доступа к памяти по трассе программы

Вход: U - идентификатор потока, из которого выполняется доступ к памяти
J - порядковый номер доступа к памяти в трассе потока с идентификатором U
$Trace$ - трасса программы по определению 6
Выход: стек вызовов функций $\mathbb{N} \to \Gamma$

function stack(U, J, Trace)
  $T_U \in Trace : equal(T_U.U, U)$
  $s : \emptyset \to \Gamma$
  return $rstack(T_U.S, J, 1, s)$
end function
function rstack(e, i, j, s)
  if j = i **then**
    return s
  else if e(j).K = 2 {вызов функции} **then**
    return $rstack(e, i, j+1, push(s, e(j).V))$
  else if e(j).K = 3 {завершение функции} **then**
    $s_2 : Dom(s) \setminus \{|Dom(s)|\} \to \Gamma, \forall k \in Dom(s) \setminus \{|Dom(s)|\} : s_2(k) = s(k)$
    return $rstack(e, i, j+1, s_2)$
  else
    return $rstack(e, i, j+1, s)$
  end if
end function

Алгоритм 10. Проверка, является ли данная последовательность вызовов функций верхней частью стека вызовов функций

Вход: $s_1 : \mathbb{N} \to \Gamma$ - последовательность вызовов функций
$s_2 : \mathbb{N} \to \Gamma$ - стек вызовов функций
Выход: истина, если последовательность вызовов функций является верхней частью стека вызовов функций, иначе ложь

function isTop(s_1, s_2)
  return $Dom(s_1) \subset Dom(s_2) \land \forall k \in Dom(s_1)$:
  $s_1(k).I = s_2(|Dom(s_2)| - k + 1).I \land s_1(k).F = s_2(|Dom(s_2)| - k + 1).F$
end function

Алгоритм 11. Нахождение верхней части стека вызовов функций, которая отличает данный стек от заданного множества стеков

Вход: $s : \mathbb{N} \to \Gamma$ - стек вызовов функций
C - множество стеков вызовов функций
Выход: последовательность вызовов функций (верхняя часть стека вызовов s в обратном поряке)

function top(s, C)
  $s_2 : \emptyset \to \Gamma$
  return $rtop(s, C, s_2, |Dom(s)|)$
end function
function rtop(s_1, C, s_2, i)
  if $i > 0 \land \exists s \in C : s(i) = s_1(i)$ **then**
    $s_3 : Dom(s_2) \cup \{|Dom(s_2)| + 1\} \to \Gamma$

$$s_3(k) = \begin{cases} \forall k \in Dom(s_2) & s_2(k) \\ k = |Dom(s_2)| + 1 & s_1(i) \end{cases}$$

    return $rtop(s_1, \{s \in C : s(i) = s_1(i)\}, s_3, i - 1)$
  else
    return s_2
  end if
end function