УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
июль–август 2022 Том 22 № 4 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
Jule–August 2022 Vol. 22 No 4 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2022-22-4-734-741
УДК 004.416.2

Организация фаззинг-тестирования многопоточных приложений на основе метода распараллеливания независимых переходов

Олег Владимирович Доронин✉

Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
dorooleg@yandex.ru✉, https://orcid.org/0000-0003-4209-8440

Аннотация

Предмет исследования. Современные информационные системы сложно представить без использования многопоточности. Многопоточность может как повышать производительность системы в целом, так и замедлять выполнение приложений за счет возникновения ошибок многопоточного программирования. Для нахождения таких ошибок на языках C/C++ существует модуль компилятора Google Thread Sanitizer. Но порядок выполнения потоков может каждый раз меняться при запуске программы на выполнение и влиять на появление подобных ошибок. Для многократного изменения порядка выполнения потоков за время работы программы в Google Thread Sanitizer применен модуль фаззинг-тестирования, который повысил вероятность нахождения ошибок. Все алгоритмы планирования потоков в фаззинг-модуле предназначены для последовательного выполнения потоков, что приводит к значительному замедлению работы Google Thread Sanitizer. Также происходит влияние на тестирование приложений, которое зависит от асинхронных взаимодействий (ожидания сетевых событий, ограничения времени выполнения операций). **Метод.** Для ускорения работы фаззинг-планировщиков предложен метод распараллеливания независимых переходов. Ошибки многопоточного программирования возникают при изменении разделяемого состояния между потоками, при этом локальные вычисления не влияют на воспроизведение многопоточных ошибок. Изменение разделяемых состояний происходит в точках синхронизаций, где выполняется переключение потоков по принципу кооперативной многозадачности. Предложено осуществлять управление последовательностями выполнения потоков только при изменении разделяемых состояний в точках синхронизаций, а локальные вычисления выполнять параллельно. Данное условие позволило сократить время тестирования без снижения результативности обнаружения ошибок многопоточного программирования. Для анализа теоретической сложности алгоритма планирования применен метод комбинаторного подсчета. **Основные результаты.** Предложен новый подход организации фаззинг-тестирования на основе метода распараллеливания независимых переходов, реализация которого по теоретическим и практическим оценкам показывает заметное ускорение работы фаззинг-планировщиков. Результаты эксперимента показали, что для алгоритма перебора всех вариантов выполнения приложения ускорение выполнения достигает 1,25 раза для двух потоков. Представлено соотношение для оценки ускорения в случае произвольного числа потоков. **Практическая значимость.** Предложенный подход позволяет покрывать фаззинг-тестами многопоточные приложения, для которых важно время выполнения — приложения с привязкой к асинхронным взаимодействиям.

Ключевые слова

инструменты поиска ошибок, многопоточность, фаззинг-тестирование, алгоритмы планирования, компиляторы

Благодарности

Персональная благодарность Дергачеву Андрею Михайловичу за оказанную поддержку и мотивацию при написании работы.

Ссылка для цитирования: Доронин О.В. Организация фаззинг-тестирования многопоточных приложений на основе метода распараллеливания независимых переходов // Научно-технический вестник информационных технологий, механики и оптики. 2022. Т. 22, № 4. С. 734–741. doi: 10.17586/2226-1494-2022-22-4-734-741

Improvement and comparison the performance of fuzzing testing algorithms for applications in Google Thread Sanitizer

Oleg V. Doronin✉

ITMO University, Saint Petersburg, 197101, Russian Federation

dorooleg@yandex.ru✉, https://orcid.org/0000-0003-4209-8440

Abstract

It is difficult to imagine modern information systems without the use of multithreading. The use of multithreading can both improve the performance of the system as in whole so as slow down the execution of multithreaded applications due to the occurrence of multithreaded programming errors. To find such errors in C/C++ languages, there exists a Google Thread Sanitizer compiler module. The order of execution of threads can change every time the program is started for execution and can affect the appearance of such errors. To repeatedly change the order of execution of threads during the execution of the program, Google Thread Sanitizer has a fuzzing testing module that allows you to increase the probability of finding errors. But all the thread scheduling algorithms in this module are presented in the form of sequential execution of threads which can lead to a significant slowdown in Google Thread Sanitizer as well as can affect the testing of applications that depends on timers (waiting for network events, deadline for operations, ...). To speed up the work of fuzzing schedulers, a method for parallelizing independent transitions is proposed. From the point of view of multithreaded programming errors, it is only important to change the shared state between threads, and local calculations do not affect the reproduction of multithreaded errors. The changes of shared states themselves occur at synchronization points (places in the code where threads are switched according to the principle of cooperative multitasking). The method suggests ordering only the change of shared states at synchronization points, and performing local calculations in parallel, due to which parallelization is achieved. For the analysis of theoretical complexity of the algorithm, the method of combinatorial counting is used. A new approach to the organization of fuzzing testing based on the method of parallelization of independent transitions is proposed the implementation of which, according to theoretical and practical estimates, shows a noticeable acceleration of the work of fuzzing schedulers. According to the results of the experiment, it was revealed that for the algorithm of iterating through all execution variants, the acceleration of execution reaches 1.25 times for two threads. For an arbitrary number of threads, an estimate is presented in the form of a formula. The proposed approach allows fuzzing tests to cover multithreaded applications for which execution time is important — applications with reference to timers which improve the quality of the software.

Keywords

error search tools, multithreading, fuzzing testing, scheduling algorithms, compilers

Acknowledgements

Personal thanks to Andrey Mikhailovich Dergachev for the support and motivation in writing the work.

For citation: Doronin O.V. Improvement and comparison the performance of fuzzing testing algorithms for applications in Google Thread Sanitizer. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2022, vol. 22, no. 4, pp. 734–741 (in Russian). doi: 10.17586/2226-1494-2022-22-4-734-741

Введение

Для инструментирования и выявления ошибок в программах, написанных на языках C/C++ [1], часто используется модуль компилятора Google Thread Sanitizers (GTSAN).

Для однопоточных приложений используются модули address, memory и др., для многопоточных — только GTSAN. Основные ошибки, которые обнаруживает модуль GTSAN[1], — гонки данных [2–4] и взаимоблокировки. Кроме основных ошибок, встречаются и такие редкие, как проблема ABA [5], которые характерны алгоритмам lock-free и wait-free [6, 7], но в GTSAN нет алгоритмов для их обнаружения.

Успешность нахождения основных ошибок зависит от порядка выполнения потоков. Для повышения вероятности нахождения ошибок используется фаззинг-модуль, интегрированный в GTSAN [8]. Фаззинг-модуль имеет два базовых варианта архитектуры управления потоками, основанные на: легковесных потоках (однопоточный режим, без возможности распараллеливания); потоках POSIX (Portable Operating System Interface), которые приостанавливаются в нужные фаззинг-планировщику моменты времени.

В дополнение к фаззинг-модулям существуют инструменты формальной верификации приложений — model checking [9–11]. Отметим, что данные инструменты позволяют проверить теоретическую корректность алгоритмов и, как правило, не встраиваются при верификации реальных приложений.

В модуле GTSAN, кроме архитектуры управления потоками, используются различные планировщики, дающие возможность управлять последовательностью выполнения потоков и повышать вероятность нахождения ошибок в многопоточном коде. Алгоритмы планирования GTSAN реализуются только последовательно, что замедляет выполнение одной итерации программы. В настоящей работе предложен метод распараллеливания алгоритмов фаззинг-планирования потоков в GTSAN. Проведено сравнение времени выполнения приложений с аналогами.

[1] ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests [Электронный ресурс]. URL: http://code.google.com/p/data-race-test, свободный. Яз. англ. (дата обращения: 19.02.2022).

Архитектура Google Thread Sanitizer

GTSAN — встроенный в компилятор модуль, который работает на одной из стадий оптимизации. На

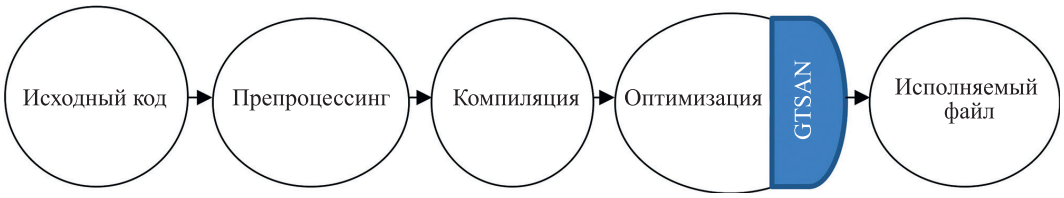

Рис. 1. Этапы компиляции программы
Fig. 1. Stages of program compilation

данном этапе происходит перехват вызовов для работы с атомарными переменными, мьютексами, условными переменными и другими примитивами синхронизации. На рис. 1 представлены этапы компиляции и место встраивания GTSAN.

Платформа управления потоками построена с использованием POSIX потоков [12] и алгоритма управления потоками на основе ожиданий. Когда исполнение потока доходит до точки синхронизации — SynchronizationPoint, поток может приостановить свою работу или передать ее другому потоку на выполнение [13]. Чтобы обеспечить корректную работу с примитивами синхронизации, для которых еще не поддержана работа в фаззинг-модуле, существует специальный поток — WatchDog. Задача WatchDog — следить за временем, которое тратит поток на выполнение, а в случае превышения порогового значения, переводить следующий поток в состояние выполнения. Для смены состояния потока применяется планировщик Scheduler (рис. 2).

В случае параллельных алгоритмов планирования требуется разделить SynchronizationPoint на два состояния — до (Before) и после (After), как показано на рис. 3. Работа в рамках SynchronizationPoint будет последовательной (с примитивом синхронизации), но она вносит минимальный вклад во время выполнения программы. Распараллеливание будет происходить на участках между SynchronizationPoint.

На рис. 3 в качестве примера, точки синхронизации изображены желтой окружностью со следующими операциями:

— load/store — чтение и запись в атомарную операцию:
— lock/unlock — захват и освобождения мьютекса;
— wait/notify — ожидание и пробуждение потока, взаимодействующего с условной переменной.

Параллельные алгоритмы планирования

В GTSAN существуют последовательные алгоритмы фаззинг-тестирования потоков, такие как: случайный, случайный с разными распределениями, полный перебор, полный перебор на фиксированном окне, полный перебор всех состояний. Все эти алгоритмы созданы для последовательного исполнения потоков

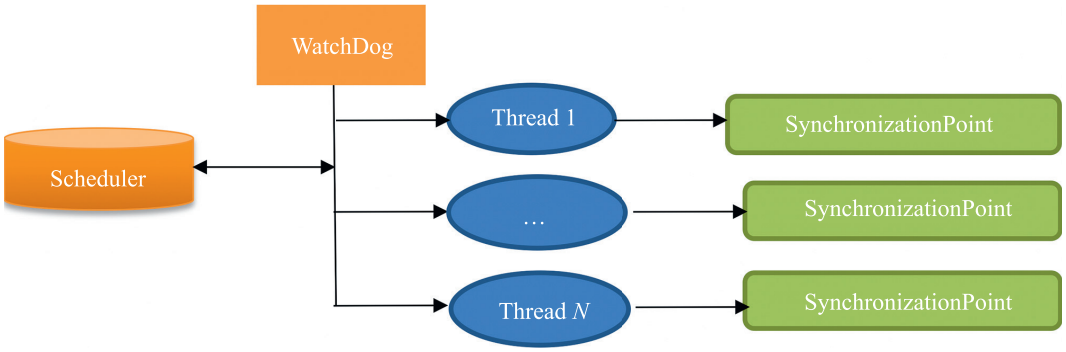

Рис. 2. Схема управления потоками с помощью алгоритма на основе ожиданий
Fig. 2. Thread control scheme using based on waiting algorithm

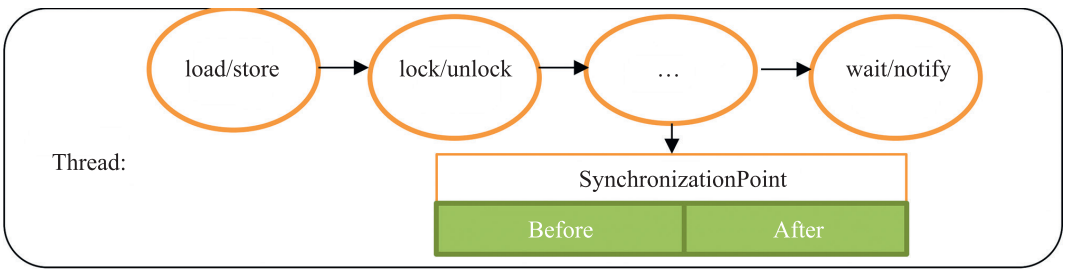

Рис. 3. Представление SynchronizationPoint как два участка Before/After
Fig. 3. Representation of SynchronizationPoint as two sections Before/After

[14, 15]. Рассмотрим процесс изменения каждого перечисленного алгоритма в параллельный.

Перебор всех вариантов. Это один из самых сложных алгоритмов для распараллеливания в модуле фаззинг-тестирования потоков. Создав параллельную реализацию данного алгоритма возможно автоматически распараллелить и все остальные алгоритмы. Идея последовательного алгоритма заключается в следующем: каждое исполнение можно представить в виде плавающей системы счисления [14], где максимальное число состояний разряда ограничено числом потоков, которые можно выбрать в этот логический момент времени. Перебирая все возможные числа в такой системе счисления, получим все варианты исполнения потоков. Под логическим временем имеем в виду дискретную величину, монотонно растущую на единицу при прохождении через точку синхронизации, начинающуюся с нуля. Данное время — общее для всех потоков. После прохождения логического времени выбирается следующий поток, продолжающий выполнение алгоритма, в это время все остальные потоки ожидают наступления логического момента, после которого они смогут начать свое выполнение. Понятие времени с точки зрения физики не подходит, так как это — непрерывная величин, и переключение потоков в произвольный момент физического времени может не давать новых вариантов обращений к разделяемым ресурсам и не приведет к воспроизведению новых ошибок многопоточного программирования.

Алгоритм перебора всех вариантов разбивается на две части: обработка точек синхронизации и выбор потока на исполнение.

Потоки между точками синхронизации можно выполнять в параллельном режиме, а логику точек синхронизации (load/store, lock/unlock, wait/notify) только в последовательном. Последовательное исполнение в этом случае оказывает минимальное влияние, так как эти части и в обычной программе исполняются с синхронизациями.

Выбор следующего потока на исполнение осуществляется в SynchronizationPointAfter(), GetNextTid() и в момент изменения состояния. Но если изменили состояние на тот же самый поток, то тогда в SynchronizationPointBefore() у потока добавляется проверка наличия состояния RUNNING вместо PARALLEL_RUNNING, и поток необходимо продолжить исполнять.

Когда вызывается функция SynchronizationPointAfter() — значит код обработки точки синхронизации завершен, и поэтому можно изменить логическое время. Определим идентификатор следующего на выполнение потока с помощью GetNextTid(), и если этот поток ожидает разрешение на выполнение, то переведем его в состояние RUNNING. GetNextTid() может вернуть тот же самый поток (tid == nextTid), тогда поток не может параллельно запустить сам себя. Другой вариант — когда GetNextTid() попадает в уже выполняющийся поток, который невозможно запустить параллельно с самим собой. Тогда необходимо возвратить обратно идентификатор следующего на исполнении потока через вызов PutNextTid(nextTid), и он уже будет подхвачен при достижении точки синхронизации одним из выполняющихся потоков. Если GetNextTid() возвращает поток, который находится в состоянии WAIT, то необходимо перевести его в PARALLEL_RUNNING, тогда поток пройдет в SynchronizationPointBefore(). При этом поток может находится в цикле while, если еще не зашел в SynchronizationPointBefore(). Это возможно при старте потока (все потоки создаются с состоянием WAIT). После этого поток пройдет SynchronizationPointBefore() и в SynchronizationPointAfter() изменит состояние на PARALLEL_RUNNING, из-за которого при следующем прохождении SynchronizationPointBefore() он приостановит свое выполнение, только если GetNextTid() не ожидал его на выполнение еще раз.

Псевдокод SynchronizationPointBefore имеет вид:

```
SynchronizationPointBefore():
   tid = GetTid();
   oldState = state[tid];
   if (oldState != RUNNING):
       state[tid] = WAIT
   while (state[tid] == WAIT) Yield();
```

Псевдокод SynchronizationPointAfter имеет вид:

```
SynchronizationPointAfter():
   tid = GetTid();
   state[tid] = PARALLEL_RUNNING;
   nextTid = GetNextTid();
   if state[nextTid] == WAIT:
      state[nextTid] = RUNNING;
   else:
     PutNextTid(nextTid)
```

Рассмотрим пример, в котором два потока пытаются изменить разделяемую переменную *value N* раз.

Пример кода для оценивания теоретического распараллеливания программы:

```
for (int j = 0; j < N; j++) {
   value++;
   // логика с интенсивным использованием
   ресурсов центрального процессора, T
   миллисекунд
}
```

Время вычисления, присваивания и сравнения переменных *value*, *j*, *N* гораздо меньше по сравнению со временем интенсивного использования ресурсов центрального процессора. Потому переменная *T* будет обозначать время выполнения одного цикла выполнения программы.

Число логических времен в этом примере равно $2 \times N$, и в каждой такой точке логического времени могут выполняться 0 или 1 поток. 0 и 1 потоки по N раз встречаются в такой последовательности логических времен. Число всех возможных последовательностей исполнения потоков равно $\binom{2 \times N}{N}$, каждая из которых содержит $2 \times N - 1$ переходов.

Оценим число переходов между логическим временем $i-1$ и i, которые занимают нулевое время исполнения (к этому времени они уже вычислены). Для перехода логического времени из 0 в 1 с уже вычисленными данными должна быть последовательность исполнения потоков 01 или 10, и далее все возможные комбинации на оставшихся последовательностях. В оставшейся последовательности логических времен будет $N-1$ точек с идентификатором потока 0 и $N-1$ точек с идентификатором потока 1, в сумме $2\times N-2$ точек последовательности. Число возможных сочетаний последовательности будет равно $\binom{2\times N-2}{N-1}$, умноженное на два для двух случаев 01 или 10. Далее преобразуем это число в другую форму для упрощения подсчета. Для этого воспользуемся рекурсивной формулой подсчета числа сочетаний $\binom{N}{k}=\binom{N-1}{k}+\binom{N-1}{k-1}$ и правилом симметрии $\binom{N}{k}=\binom{N}{N-k}$, где k — набор элементов выбираемых из N элементного множества, тогда получим:

$$2\times\binom{2\times N-2}{N-1}=2\times\left(\binom{2\times N-3}{N-1}+\binom{2\times N-3}{N-2}\right)=$$
$$=2\times\left(\binom{2\times N-3}{N-1}+\binom{2\times N-3}{N-1}\right)=4\times\binom{2\times N-3}{N-1}.$$

Выполним оценку остальных переходов между $i-1$ и i, где $i-1>0$. Выберем подпоследовательности исполнения потоков 001 или 110 в произвольном логическом времени, так как только они дают нулевое время выполнения программы. Получим всего комбинаций $2\times\binom{2\times N-3}{N-1}$ и последовательностей $2\times N-2$. В итоге имеем $4\times\binom{2\times N-3}{N-1}$ с использованием рекурсивной формулы и правила симметрии. Тогда доля распараллеливаемых участков равна отношению числа переходов с нулевым временем исполнения к числу всех возможных переходов для всех вариантов исполнения потоков. Для упрощения формулы воспользуемся выражением подсчета сочетаний через факториалы.

$$\binom{N}{k}=\frac{N!}{k\times!(N-k)!}\cdot\frac{4\times N\times\binom{2\times N-3}{N-1}}{(2\times N-1)\times\binom{2N}{N}}=$$
$$=\frac{N\times4\times\dfrac{(2\times N-3)!}{(N-1)!\times(N-2)!}}{(2\times N-1)\times\dfrac{(2\times N)!}{N!\times N!}}=$$
$$=\frac{N\times4\times(2\times N-3)!\times N!\times N!}{(2\times N-1)\times(N-1)!\times(N-2)!\times(2\times N)!}=$$
$$=\frac{N\times4\times N\times N\times(N-1)}{(2\times N-1)\times2\times N\times(2\times N-1)\times(2\times N-2)}=$$
$$=\frac{N\times N}{(2\times N-1)\times(2\times N-1)}=\left(\frac{N}{2\times N-1}\right)^2.$$

Если рассмотреть график функции зависимости числа шагов от распараллеливаемых участков кода, например, при $N=5$, то коэффициент параллельного выполнения равен 0,308642, а предел функции $\lim\limits_{N\to\infty}\left(\dfrac{N}{2\times N-1}\right)^2=0{,}25$. В среднем получим коэффициент распараллеливания не хуже, чем 0,25 для одной итерации выполнения программы.

Выведем формулу для вычисления распараллеливаемых участков с произвольным числом потоков T. Пронумеруем потоки от 1 до i и обозначим через n_i число шагов, которые проделывает поток (пусть для всех потоков оно будет одинаковым $n_1=n_2=\cdots=n_T=n$). Тогда $n\times T=N$ число всех шагов, а $N-1$ — число всех переходов. Вычислим число всех переходов между $k-1$ и k, оно будет равно $\binom{N}{n_1\ldots n_i\ldots n_T}$ числу всех путей (число всех возможных комбинаций исполнения потоков). Из полученного числа переходов рассчитаем число переходов с соседними повторяющимися потоками для переходов из $k-1$ и k. Такие переходы невозможно распараллелить, так как для продвижения логического времени тот же самый поток должен дойти до точки синхронизации, а это невозможно, так как один и тот же поток не может выполняться параллельно сам с собой. Для каждой подпоследовательности на логическом интервале времени от $k-T+1$ до k будем вести подсчет. Тогда вне этого интервала имеем $N-T-1$ переходов. Зафиксируем равные потоки на данном интервале при переходе от $k-1$ в k. Число шагов, в которых будет встречаться фиксированный поток вне этого интервала, будет не больше, чем $n-2$ раз, а для остальных потоков — в n раз. Для фиксированного набора потоков на подпоследовательности $k-T+1$ до k получим число комбинаций исполнения потоков: $\binom{N-T-1}{n_1-2-p_1\ldots n_i-p_i\ldots n_T-p_T}$, где $p_1, p_i\ldots p_T$ — число потоков с номером i, используемых на интервале от $k-T+1$ до k . Рассчитаем число всех комбинаций исполнения потоков для различных $p_1, p_i\ldots p_T$:

$$\sum_{\substack{p_1+\ldots+p_T=T-1\\0\le p_i\le T-1}}\binom{N-T-1}{n_1-2-p_1\ldots n_i-p_i\ldots n_T-p_T},$$

а для всех фиксированных T получим

$$T\times\sum_{\substack{p_1+\ldots+p_T=T-1\\0\le p_i\le T-1}}\binom{N-T-1}{n_1-2-p_1\ldots n_i-p_i\ldots n_T-p_T}.$$

Следующим шагом вычислим число исполнений с различными потоками перед k, и не совпадающими с k и $k-1$. Такие переходы нельзя распараллелить, так как необходим один дополнительный свободный поток, а все T потоки выполняются, и для продвижения логического времени следует дождаться освобождения потоков. Тогда в момент времени k возможно выполнение одного из T потоков, а в момент времени $k-1$ — любого, кроме исполняющегося в момент времени k (их число равно $T-1$). На оставшейся фиксированной под-

последовательности могут быть $(T-1)!$ перестановок различных потоков. Для конкретной фиксированной подпоследовательности получим число последовательностей исполнения потоков, равное числу сочетаний $\binom{N-T-1}{n_1-2\ \ldots\ n_i-1\ \ldots\ n_T-1}$.

Сложим все варианты вместе и получим итоговую формулу:

$$T\times(T-1)\times(T-1)!\times\binom{N-T-1}{n_1-2\ \ldots\ n_i-1\ \ldots\ n_T-1}.$$

Для вычисления числа последовательностей исполнений, которые приводят к распараллеливанию фиксированного перехода, необходимо отнять из всех последовательностей только те, которые не приведут к распараллеливанию. В итоге имеем

$$\binom{N}{n_1\ \ldots\ n_i\ \ldots\ n_T}-T\times$$
$$\times\sum_{\substack{p_1+\ldots+p_T=T-1\\ 0\le p_i\le T-1}}\binom{N-T-1}{n_1-2-p_1\ldots\ n_i-p_i\ \ldots\ n_T-p_T}-T\times$$
$$\times(T-1)\times(T-1)!\times\binom{N-T-1}{n_1-2\ \ldots\ n_i-1\ \ldots\ n_T-1}.$$

Случайный планировщик и планировщик с различными распределениями случайных величин. Случайный планировщик для выбора потоков использует равномерное распределение, формула плотности которого имеет вид

$$f(x)=\frac{1}{b-a},\ x\in[a,b],$$

где a, b — границы конечного интервала, в котором плотность сохраняет постоянное значение.

При использовании алгоритма планирования с различными случайными величинами, в циклическом буфере найдем генераторы случайных величин, и на каждой итерации выберем свой генератор (нормальное распределение, логнормальное, экспоненциальное, и др.) [14].

Для распараллеливания планировщиков воспользуемся подходом, который описан в планировщике перебора всех вариантов. GetNextTid() возвращает случайный поток, а SynchronizationPointBefore() и SynchronizationPointAfter() решают, можно ли его запустить параллельно с остальными. Отметим, что в алгоритме планирования есть некоторые ограничения на предел распараллеливания потоков в среднем.

Рассмотрим второй алгоритм, который заключается в следующем. В каждой точке синхронизации SyncronizationPoint (вместо SynchronizationPointBefore() и SynchronizationPointAfter()) выберем потоки, которые могут дальше выполняться, и исполним их. Причем выбор числа потоков осуществим с помощью генератора случайных величин. Изменяя число исполняемых потоков от максимального к минимальному, произведем проверку уровня параллелизма в системе. Для случайного планировщика строгой завязки на порядок исполнения потоков не требуется, и поэтому последовательный порядок исполнения между точками синхронизации можно не гарантировать.

Планировщик с перебором на фиксированном окне. Идея планировщика с фиксированным окном заключается в следующем: в качестве параметра задается размер окна, в котором происходит перебор всех вариантов, а окно двигается в соответствии с логическим временем [15].

Один алгоритм заключается в использовании подхода, который описан в переборе всех вариантов, что позволяет повысить уровень параллелизма. GetNextTid() возвращает случайный поток, а SynchronizationPointBefore() и SynchronizationPointAfter() решают, можно ли его запустить параллельно с остальными.

Другой подход заключается в достижении нужного логического времени путем использования случайного планировщика с максимальным уровнем параллелизма. В этом случае максимальная скорость исполнения достигается в интервалах до и после окна. А обработка окна осуществляется параллельным алгоритмом полного перебора.

Планировщик с перебором всех состояний. Для планировщика с перебором всех состояний основная цель, чтобы в каждый логический момент времени поработал каждый из возможных потоков. Тогда число итераций для такого планировщика не превышает максимальное число потоков в системе [14]. Первая итерация в таком алгоритме производится без ограничений на порядок. А следующие итерации — с использованием подхода, описанного в полном переборе вариантов.

Результаты тестирования

Для проведения тестирования использованы: существующие unit тесты платформы GTSAN[1]; тестовые примеры, на которых успешно проверены последовательные алгоритмы.

Листинг 1. Пример гонки данных.

```
// std::atomic_int d;
// int a;
// thread

++d; ++a; ++d;
```

В *листинге* 1 присутствует гонка данных на переменной *a*. Два потока пытаются получить доступ к переменной без синхронизации. Планировщики всех параллельных алгоритмов успешно находят гонку данных.

Листинг 2. Пример с редкими случаями появления значений.

```
// std::atomic_int value { 0 };
// thread:
for (int j = 0; j < 5; j++) {
  auto r = value.load();
  r++;
  value.store(r);
}
```

[1] ThreadSanitizer unit tests [Электронный ресурс]. URL: https://github.com/llvm-mirror/compiler-rt/tree/master/lib/tsan/tests, свободный. Яз. англ. (дата обращения: 19.02.2022).

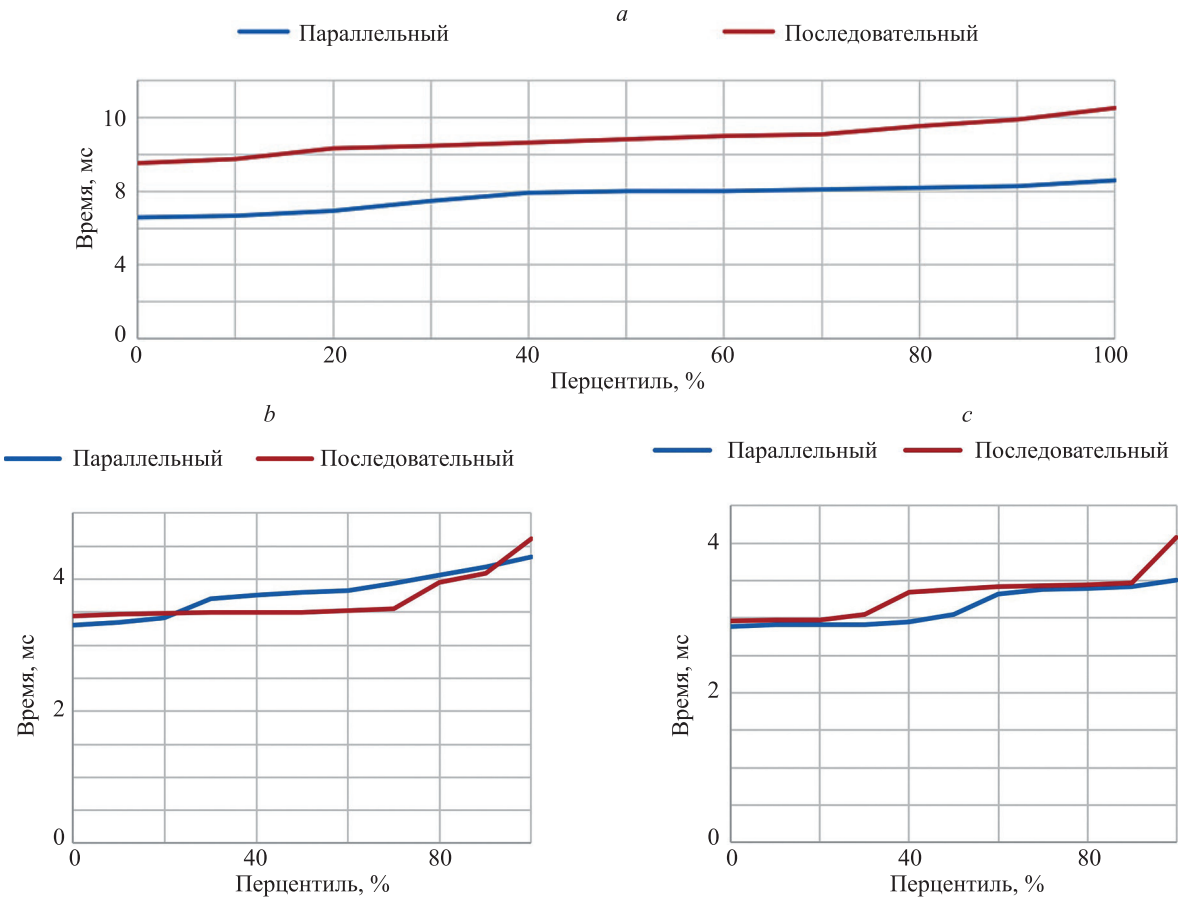

Рис. 4. Сравнение планировщиков на примере кода с редкими случаями проявления значений: переборы всех вариантов (*a*) и на фиксированном окне (*b*); случайный планировщик (*c*)

Fig. 4. Comparison of schedulers on the example of code with rare cases of manifestation of values: iteration of all options (*a*) and on a fixed window (*b*); random scheduler (*c*)

В *листинге* 2 приведен пример, который демонстрирует, насколько сложные случаи позволяют обнаруживать фаззинг-планировщики (планировщики операционной системы на более чем сто тысяч итераций не способны обнаружить все возможные значения value). Параллельные алгоритмы для полного перебора вариантов и планировщик с различными распределениями случайных величин позволяют получить все наборы результатов для переменной *r* от 2 до 10, как это было с последовательными алгоритмами.

Выполним сравнение параллельных и последовательных планировщиков на примере кода с редкими случаями проявления значений. Проведем тестирование на процессоре Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz (8 ядер).

На графиках рис. 4 показано распределение среднего времени выполнения одной итерации на ста запусках для каждого алгоритма планирования с последовательной и параллельной реализациями. В случаях планировщика с перебором всех вариантов (рис. 4, *a*) и случайного планировщика (рис. 4, *c*) заметно уменьшение времени работы в параллельной реализации. При фиксированном окне (рис. 4, *b*) сложно сказать об улучшении времени работы.

Заключение

Для перехода от последовательного исполнения потоков при тестировании многопоточных приложений к параллельному, разработаны алгоритмы распараллеливания для разных способов фаззинг-планирования потоков в модуле GTSAN. Для каждого из разработанных алгоритмов выполнена теоретическая оценка времени выполнения, подтвержденная практическим экспериментом.

Разработанные алгоритмы параллельного фаззинг-тестирования асимптотически показали лучшие результаты, чем существующие аналоги, а с точки зрения корректности обнаружения ошибок многопоточного программирования продемонстрировали результаты, аналогичные уже реализованным в GTSAN последовательным алгоритмам. Полученные результаты позволяют ускорить фаззинг-планирование потоков, а также сделать возможным фаззинг-тестирование приложений, для которых критически важно время выполнения одной итерации программы.

Литература

1. Stroustrup B. The C++ Programming Language. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. 1019 p.
2. Serebryany K., Iskhodzhanov T. ThreadSanitizer - Data race detection in practice // ACM International Conference Proceeding Series. 2009. P. 62–71. https://doi.org/10.1145/1791194.1791203
3. Netzer R.H.B., Miller B.P. What are race conditions?: Some issues and formalizations // ACM Letters on Programming Languages and Systems (LOPLAS). 1992. V. 1. N 1. P. 74–88. https://doi.org/10.1145/130616.130623
4. Banerjee U., Bliss B., Ma Z., Petersen P. A theory of data race detection // Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD). 2006. P. 69–78. https://doi.org/10.1145/1147403.1147416
5. Dechev D., Pirkelbauer P., Stroustrup B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs // Proc. of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). V. 1. 2010. P. 185–192. https://doi.org/10.1109/ISORC.2010.10
6. Anderson J., Ramamurthy S., Jeffay K. Real-time computing with lock-free shared objects // ACM Transactions on Computer Systems. 1997. V. 15. N 2. P. 134–165. https://doi.org/10.1145/253145.253159
7. Harris T.L., Fraser K., Pratt I.A. A practical multi-word compare-and-swap operation // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2002. V. 2508. P. 265–279. https://doi.org/10.1007/3-540-36108-1_18
8. Саттон М., Грин А., Амини П. Fuzzing: исследование уязвимостей методом грубой силы. Москва: Символ-Плюс, 2009. 555 с.
9. Clarke E., Grumberg O., Peled D. Model Checking. MIT Press, 1999. 314 p.
10. Meyers S., Alexandrescu A. C++ and the Perils of Double-Checked Locking: Part I // Dr. Dobb's Journal. 2004. V. 29. N 7. P. 46–49.
11. Gluck P., Holzmann G. Using SPIN model checking for flight software verification // IEEE Aerospace Conference Proceedings. 2002. V. 1. P. 105–113. https://doi.org/10.1109/AERO.2002.1036832
12. Garcia F., Fernandez J. Posix thread libraries // Linux Journal. 2000. V. 2000. N 70. P. 36.
13. Doronin O., Dergun K., Dergachev A., Ilina A. Fuzz testing of multithreaded applications based on waiting // CEUR Workshop Proceedings. 2020. V. 2590. P. 1–8.
14. Doronin O., Dergun K., Dergachev A. Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code // CEUR Workshop Proceedings. 2019. V. 2344. P. 1–12.
15. Дергун К.И., Доронин О.В. Фаззинг тестирование fine-grained алгоритмов // Сборник тезисов докладов VIII Конгресса молодых ученых. Электронное издание. СПб: Университет ИТМО, 2019.

References

1. Stroustrup B. *The C++ Programming Language*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 2000, 1019 p.
2. Serebryany K., Iskhodzhanov T. ThreadSanitizer — Data race detection in practice. *ACM International Conference Proceeding Series*, 2009, pp. 62–71. https://doi.org/10.1145/1791194.1791203
3. Netzer R.H.B., Miller B.P. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1992, vol. 1, no. 1, pp. 74–88. https://doi.org/10.1145/130616.130623
4. Banerjee U., Bliss B., Ma Z., Petersen P. A theory of data race detection. *Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006, pp. 69–78. https://doi.org/10.1145/1147403.1147416
5. Dechev D., Pirkelbauer P., Stroustrup B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. *Proc. of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). V. 1*, 2010, pp. 185–192. https://doi.org/10.1109/ISORC.2010.10
6. Anderson J., Ramamurthy S., Jeffay K. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 1997, vol. 15, no. 2, pp. 134–165. https://doi.org/10.1145/253145.253159
7. Harris T.L., Fraser K., Pratt I.A. A practical multi-word compare-and-swap operation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002, vol. 2508, pp. 265–279. https://doi.org/10.1007/3-540-36108-1_18
8. Sutton M., Greene A., Armini P. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007, 576 p.
9. Clarke E., Grumberg O., Peled D. *Model Checking*. MIT Press, 1999, 314 p.
10. Meyers S., Alexandrescu A. C++ and the Perils of Double-Checked Locking: Part I. *Dr. Dobb's Journal*, 2004, vol. 29, no. 7, pp. 46–49.
11. Gluck P., Holzmann G. Using SPIN model checking for flight software verification. *IEEE Aerospace Conference Proceedings*, 2002, vol. 1, pp. 105–113. https://doi.org/10.1109/AERO.2002.1036832
12. Garcia F., Fernandez J. Posix thread libraries. *Linux Journal*, 2000, vol. 2000, no. 70, pp. 36.
13. Doronin O., Dergun K., Dergachev A., Ilina A. Fuzz testing of multithreaded applications based on waiting. *CEUR Workshop Proceedings*, 2020, vol. 2590, pp. 1–8.
14. Doronin O., Dergun K., Dergachev A. Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code. *CEUR Workshop Proceedings*, 2019, vol. 2344, pp. 1–12.
15. Dergun K.I., Doronin O.V. Fuzzing testing of fine-grained algorithms. *Abstracts collection of VIII Young Scientists Congress*. St. Petersburg, ITMO University, 2019. (in Russian)

Автор

Доронин Олег Владимирович — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, SC 57208322052, https://orcid.org/0000-0003-4209-8440, dorooleg@yandex.ru

Author

Oleg V. Doronin — PhD Student, ITMO University, Saint Petersburg, 197101, Russian Federation, SC 57208322052, https://orcid.org/0000-0003-4209-8440, dorooleg@yandex.ru

Статья поступила в редакцию 26.03.2022
Одобрена после рецензирования 27.06.2022
Принята к печати 30.07.2022

Received 26.03.2022
Approved after reviewing 27.06.2022
Accepted 30.07.2022