УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
март–апрель 2019 Том 19 № 2 ISSN 2226-1494 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
March–April 2019 Vol. 19 No 2 ISSN 2226-1494 http://ntv.ifmo.ru/en

УДК 004.416.2

ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА БИБЛИОТЕКИ RELACY RACE DETECTOR

О.В. Доронин[a,b], К.И. Дергун[b,c], А.М. Дергачев[b], А.О. Ключев[b]

[a] ITIVITI, Санкт-Петербург, 190000, Российская Федерация
[b] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
[c] ПАО Сбербанк, Санкт-Петербург, 195112, Российская Федерация
Адрес для переписки: dam600@gmail.com

Информация о статье
Поступила в редакцию 15.01.19, принята к печати 25.02.19
doi: 10.17586/2226-1494-2019-19-2-380-385
Язык статьи – русский

Ссылка для цитирования: Доронин О.В., Дергун К.И., Дергачев А.М., Ключев А.О. Оптимизация программного кода библиотеки Relacy Race Detector // Научно-технический вестник информационных технологий, механики и оптики. 2019. Т. 19. № 2. С. 380–385. doi: 10.17586/2226-1494-2019-19-2-380-385

Аннотация
Приведены результаты исследования библиотеки Relacy Race Detector (RRD) применительно к задаче тестирования многопоточного кода. В ходе исследования выявлен ряд недостатков, которыми обладает библиотека RRD. Это отсутствие возможности динамического изменения числа потоков, сложная структура проекта, наличие ошибок в реализации, отсутствие поддержки снимков памяти. В работе исправлены вышеперечисленные недостатки и представлен новый подход для получения атомарных снимков памяти нескольких потоков с использованием механизмов fork и fiber. Использование полученных результатов и внесение изменений в библиотеку RRD позволяет упростить процедуру тестирования многопоточных приложений.

Ключевые слова
многопоточность, гонки данных, Relacy Race Detector, планировщик потоков, lock-free алгоритмы, тестирование приложений, операционная система, fiber

Благодарности
Персональная благодарность Д. Вьюкову (разработчик библиотеки Relacy Race Detector), М. Хижинскому (разработчик библиотеки libcds) и Е. Калишенко за обсуждение, критику и предложения в процессе выполнения работы.

PROGRAM CODE OPTIMIZATION OF RELACY RACE DETECTOR LIBRARY

O.V. Doronin[a,b], K.I. Dergun[b,c], A.M. Dergachev[b], A.O. Kluchev[b]

[a]ITIVITI, Saint Petersburg, 190000, Russian Federation
[b]ITMO University, Saint Petersburg, 197101, Russian Federation
[c]PAO Sberbank, Saint Petersburg, 195112, Russian Federation
Corresponding author: dam600@gmail.com

Article info
Received 15.01.19, accepted 25.02.19
doi: 10.17586/2226-1494-2019-19-2-380-385
Article in Russian

For citation: Doronin O.V., Dergun K.I., Dergachev A.M., Kluchev A.O. Program code optimization of Relacy Race Detector library. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2019, vol. 19, no. 2, pp. 380–385 (in Russian). doi: 10.17586/2226-1494-2019-19-2-380-385

Abstract
The paper presents the results of Relacy Race Detector (RRD) library research as applied to the problem of multithreaded code testing. The study revealed several shortcomings of the RRD library. They are: a static number of threads, complex project structure, errors in implementation and lack of support for snapshots. The work has corrected the shortcomings described above and presented a new approach for the atomic snapshot of multiple threads using fork and fiber mechanisms. With the application of these results and implemented changes it is now easier to use the RRD library for multithreaded applications testing.

Keywords
multithreading, data races, Relacy Race Detector, thread scheduler, lock-free algorithms, testing applications, operating system, fiber

Acknowledgements
The authors express personal thanks to D. Vyukov (developer of the Relacy Race Detector library), M. Khizhinsky (developer of the libcds library) and E. Kalishenko for discussion, critique and suggestions in the process of research.

Основой многопоточных программных приложений являются алгоритмы, разработка и отладка которых является сложной задачей. От порядка выполнения потоков зависит результат работы как алгоритма, так и приложения в целом. При удачном результате тестирования исполняемого кода в одном программном окружении ошибки могут проявиться в другом. Если в случае lock-based-алгоритмов критическая секция, как правило, «большая», и при тестировании даже планировщиком операционной системы возможно найти большинство ошибок, то в случае lock-free-алгоритмов найти ошибки из-за «маленькой» критической секции гораздо сложнее. Планировщик потоков операционной системы может быть одной из причин пропуска ошибок в lock-free-алгоритмах при тестировании и возникновении ошибок при исполнении многопоточных приложений. Проблему может решить инструмент, который будет автоматически перебирать комбинации исполнения потоков и выявлять ошибки в lock-free-алгоритмах на стадии тестирования и отладки приложений.

В библиотеке Relacy Race Detector (RRD) заложена возможность управления порядком выполнения потоков. Помимо планирования потоков в библиотеке реализован поиск следующих типов ошибок: взаимоблокировки, динамические блокировки, утечки памяти, гонки данных с учетом модели памяти C++0x [1], доступ к освобожденной памяти и др. Подробное описание алгоритмов поиска взаимоблокировок и гонок данных можно найти в [2, 3]. К недостаткам существующей реализации библиотеки RRD можно отнести: отсутствие возможности изменения количества потоков в момент исполнения программы; ограничение по использованию в готовых программных продуктах; структурные проблемы в организации проекта; некорректная работа с TLS [4]; отсутствие учета изменений глобальной памяти и др. Настоящая работа направлена на устранение указанных недостатков.

Для ускорения процедуры моделирования алгоритмов управления потоками и минимизации накладных расходов в библиотеке RRD применяются различные варианты оптимизации. В частности, такой оптимизацией является выделение массивов в момент компиляции вместо использования new, malloc и других инструментов управления памятью. Поскольку размер массива должен быть известен во время компиляции, то изменение его во время работы невозможно. Пример программного кода для создания потоков в исходной версии приведен в листинге 1.

Листинг 1. Пример использования RRD до изменений

```
struct stack_test : rl::test_suite<stack_test, 4>;
rl::simulate<stack_test>();
```

Для интерактивного задания числа потоков, например через консоль, необходимо множество реализаций stack_test и оберток над кодом. Во избежание избыточного кода реализована возможность динамического задания числа потоков. Для этого значительная часть кода библиотеки RRD переписана. Пример программного кода для создания потоков после внесенных в библиотеку изменений приведен в листинге 2.

Листинг 2. Пример использования RRD после изменений

```
struct stack_test;
rl::test_params p { .static_thread_count = std::atoi(argv[1]) };
rl::simulate<stack_test>(p);
```

Динамическое изменение числа потоков позволило уменьшить размер бинарных файлов по сравнению с использованием существующих тестов RRD на 9 %. Этот параметр может оказаться критическим при тестировании встроенных систем с ограниченным объемом памяти. В таблице приведены размеры бинарных файлов до и после внесения изменений в библиотеку.

Таблица. Размер бинарных файлов

Название/Версия	До внесения изменений, Б	После внесения изменений, Б
Базовые тесты	8461336	7597856
jtest	1213688	845624
ntest	1067496	702088

Тестирование lock-free-структуры данных stack, расположенной в директории example библиотеки RRD, происходило в несколько этапов – запускалось тестирование для числа потоков от 1 до n и производились измерения: времени компиляции (рис. 1), размера исполняемого файла (рис. 2), времени работы (рис. 3).

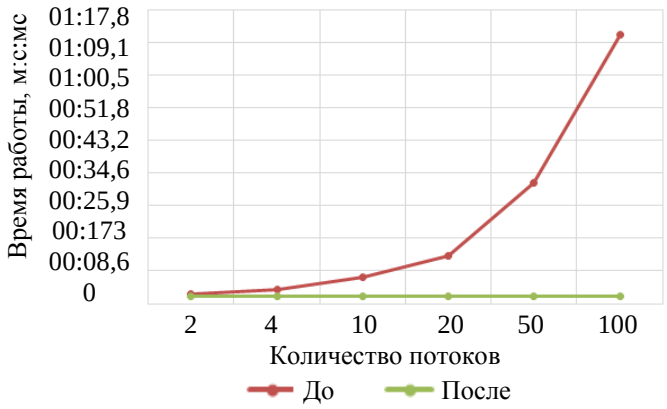

Рис. 1. Время компиляции

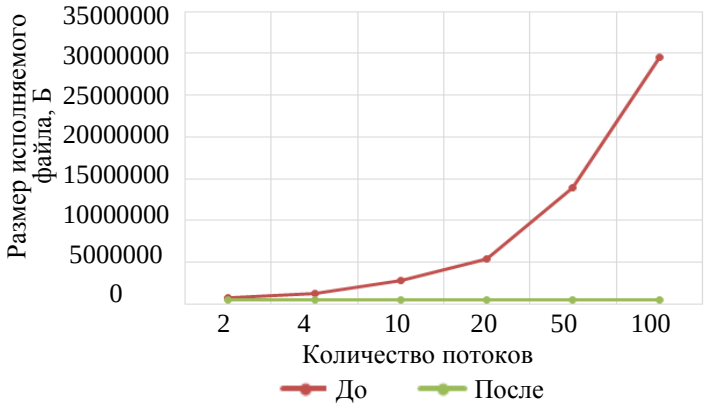

Рис. 2. Размер исполняемого файла

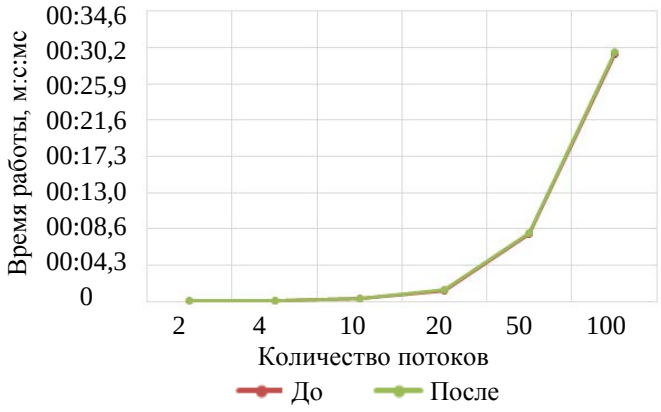

Рис. 3. Время работы программы

По полученным графикам можно сделать вывод о том, что увеличение времени работы алгоритмов несущественно, но при этом наблюдается значительное ускорение компиляции и также уменьшение размеров бинарных файлов.

Изменение структуры проекта позволило выделить тесты, примеры и библиотеку в отдельные структурные модули, а также использовать систему сборки проекта – перспективный кроссплатформенный инструмент CMake (cross-platform make), что упрощает первое использование библиотеки.

Выделены два типа ошибок в реализации библиотеки RRD:

1) предупреждения компилятора (их немного) – неиспользуемые переменные, некорректная работа с ссылками и др.;
2) критические ошибки, приводящие к неработоспособности. Была найдена ошибка, связанная с тем, что нулевой указатель преобразовывался к ссылочному типу, а по стандарту языка C/C++ разыменовывание нулевого указателя является неопределенным поведением. Например, при использовании опции O2 для компиляторов clang/gcc код становился неработоспособным.

Снимки памяти важны при тестировании и отладке многопоточных приложений, в частности, они нужны для перехода к старым состояниям программы с целью отладки; выделения границ тестируемой части кода; ускорения работы старых алгоритмов планирования, значительную часть кода которых занимают алгоритмически сложные однопоточные вычисления. Существуют различные реализации планировщиков потоков: основанные на приоритетах [5], основанные на задержках [6], контекстно-зависимые [7], рандомизированные [8, 9] и др. В библиотеке RRD реализованы: случайный планировщик, полный перебор вариантов, планировщик с обработкой граничных случаев. Но ни один из них не позволяет делать снимки памяти в момент исполнения программы. Для решения этой задачи были исследованы способы создания копии адресного пространства с помощью: виртуальной машины [10]; ручного копирования адресного пространства; механизма fork. Первые два подхода требуют много времени для реализации, второй сложен. Механизм fork позволяет создать копию адресного пространства, но обладает одним очень важным недостатком – в случае использования pthread-потоков (которые применяются в большинстве многопоточных приложений) при копировании адресного пространства и ресурсов остается лишь один поток. В библиотеке RRD все pthread-потоки заменяются на fiber, что позволяет получать снимки памяти при работе нескольких потоков.

Покажем, за счет чего снимки памяти могут давать ускорение при работе алгоритмов планирования. В библиотеке RRD код исполняется последовательно. В каждый момент времени физически может исполняться только один поток. На основании этого можно представить программу как линейную последовательность инструкций. Обозначим время выполнения программы на интервале между переключениями потоков t_i, а число таких переключений через N, тогда время выполнения программы на одном запуске можно записать в виде:

$$T_u = \sum_{i=0}^{N} t_i .$$

Посмотрим, сколько времени понадобится на выполнение программы, если при каждом переключении потоков будет создаваться снимок памяти, и программа будет запускаться сначала. Обозначим через c_i время на создание снимка памяти в момент времени i, тогда все время исполнения программы можно записать следующим образом:

$$T_s = \sum_{i=0}^{N} t_i + c_i .$$

Если предположить, что существует хотя бы одно $c_i > 0$, то алгоритм планирования потоков, основанный на снимках памяти, очевидно проигрывает. Но с помощью снимков памяти можно выполнять откат не в начало программы, а например, в снимок памяти под номером j , в этом случае:

$$T_s = \sum_{i=j}^{N} t_i + c_i .$$

Если неравенство: $\sum_{i=j}^{N} c_i < \sum_{i=0}^{j-1} t_i$ выполняется, то время работы алгоритма на новой итерации сокращается.

Сравнение времени работы алгоритмов затруднено большим количеством параметров. Приведем некоторый частный случай, демонстрирующий количественную оценку производительности, основанную на снимках памяти алгоритма планирования.

Пусть на каждом запуске можно выбрать один из двух потоков, и на каждом запуске обеспечивается N шагов алгоритма. Время работы алгоритма между переключениями потоков гарантированно равно t, а время на снимок памяти c. Тогда время работы алгоритма без снимков памяти, направленных на перебор всевозможных комбинаций исполнения потоков:

$$S_u = 2^{N+1}\left(N+1\right)t ,$$

где 2^{N+1} – число всех возможных комбинаций, а $(N + 1)t$ – время исполнения программы на одной итерации.

Время работы алгоритма, основанного на снимках памяти с переходом не в начало:

$$S_s = \sum_{i=0}^{N} 2^i \left(t+c\right) = \left(2^{N+1}-1\right)\left(t+c\right),$$

где 2^i – число итераций отрезка i программы, а $(t + c)$ – время прохождения этого отрезка.

Таким образом, основанный на снимках памяти алгоритм в рассмотренном случае работает практически в N раз быстрее.

В результате создания инструментария планирования и поиска ошибок для lock-free-алгоритмов проект RRD был существенно переработан: реализовано динамическое изменение числа потоков; изменена структура проекта; исправлены критические ошибки; добавлена функциональная возможность планирования потоков с использованием снимков памяти. Добавленная система сборки, основанная на

CMake, позволила упростить использование библиотеки сторонними разработчиками, а также выделить тесты, примеры и библиотеку в отдельные модули. В библиотеку внесены изменения[1], которые позволили упростить изменение числа потоков в момент исполнения программы, а также уменьшить размер бинарных файлов и значительно сократить время компиляции. Новый подход к планированию потоков, основанный на снимках памяти, позволил ускорить некоторые алгоритмы планирования потоков, а также получить новые функциональные возможности: возращение к предыдущим состояниям, создание копии глобального адресного пространства, выделение участков кода для тестирования и оптимизация алгоритмов планирования.

Основным недостатком[2] библиотеки RRD остается ее неприменимость для больших масштабируемых приложений. В ней отсутствует поддержка сохранения состояния глобальной памяти между запусками, некорректно организована работа с TLS. Дальнейшая работа будет направлена на внедрение алгоритмов планирования в Google Thread Sanitizer [11] для применения их на больших масштабируемых приложениях.

Литература

1. Batty M., Owens S., Sarkar S., Sewell P., Weber T., Tjark W. Mathematizing C++ concurrency // ACM SIGPLAN Notices. 2011. V. 46. N 1. P. 55–66. doi: 10.1145/1925844.1926394
2. Lidbury C., Donaldson A.F. Dynamic race detection for C++11 // ACM SIGPLAN Notices. 2016. V. 52. N 1. P. 443–457. doi: 10.1145/3093333.3009857
3. Jula H., Tralamazza D., Zamfir C., Candea G. Deadlock immunity: enabling systems to defend against deadlocks // Proc. of Operating System Design and Implementation (OSDI). 2008. P. 295–308.
4. Drepper U. ELF Handling for Thread-Local Storage. Red Hat Inc, 2005. 79 p.
5. Nagarakatte S., Burckhardt S., Martin M., Musuvathi M. Multicore acceleration of priority-based schedulers for concurrency bug detection // ACM SIGPLAN Notices. 2012. V. 47. N 6. P. 543–554. doi: 10.1145/2345156.2254128
6. Atig M.F., Bouajjani A., Qadeer S. Context-bounded analysis for concurrent programs with dynamic creation of threads // Lecture Notes in Computer Science. 2009. V. 5505. P. 107–123. doi: 10.1007/978-3-642-00768-2_11
7. Emmi M., Qadeer S., Rakamaric Z. Delay-bounded scheduling // ACM SIGPLAN Notices. 2011. V. 46. N 1. P. 411–422. doi: 10.1145/1925844.1926432
8. Burckhardt S., Kothari P., Musuvathi M., Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs // ACM SIGARCH Computer Architecture News. 2010. V. 38. P. 167–178. doi: 10.1145/1735970.1736040
9. Sen K. Effective random testing of concurrent programs // Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE). 2007. P. 323–332. doi: 10.1145/1321631.1321679
10. Coetzee L. Combining Reverse Debugging and Live Programming towards Visual Thinking in Computer Programming. Stellenbosch University, 2015. 94 p.
11. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice // Proc. Workshop on Binary Instrumentation and Applications (WBIA). 2009. P. 62–71. doi: 10.1145/1791194.1791203

References

1. Batty M., Owens S., Sarkar S., Sewell P., Weber T., Tjark W. Mathematizing C++ concurrency. *ACM SIGPLAN Notices*, 2011, vol. 46, no. 1, pp. 55–66. doi: 10.1145/1925844.1926394
2. Lidbury C., Donaldson A.F. Dynamic race detection for C++11. *ACM SIGPLAN Notices*, 2016, vol. 52, no. 1, pp. 443–457. doi: 10.1145/3093333.3009857
3. Jula H., Tralamazza D., Zamfir C., Candea G. Deadlock immunity: enabling systems to defend against deadlocks. *Proc. of Operating System Design and Implementation, OSDI*, 2008, pp. 295–308.
4. Drepper U. *ELF Handling for Thread-Local Storage*. Red Hat Inc, 2005, 79 p.
5. Nagarakatte S., Burckhardt S., Martin M., Musuvathi M. Multicore acceleration of priority-based schedulers for concurrency bug detection. *ACM SIGPLAN Notices*, 2012, vol. 47, no. 6, pp. 543–554. doi: 10.1145/2345156.2254128
6. Atig M.F., Bouajjani A., Qadeer S. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Lecture Notes in Computer Science*, 2009, vol. 5505, pp. 107–123. doi: 10.1007/978-3-642-00768-2_11
7. Emmi M., Qadeer S., Rakamaric Z. Delay-bounded scheduling. *ACM SIGPLAN Notices*, 2011, vol. 46, no. 1, pp. 411–422. doi: 10.1145/1925844.1926432
8. Burckhardt S., Kothari P., Musuvathi M., Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News*, 2010, vol. 38, pp. 167–178. doi: 10.1145/1735970.1736040
9. Sen K. Effective random testing of concurrent programs. *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering, ASE*, 2007, pp. 323–332. doi: 10.1145/1321631.1321679
10. Coetzee L. *Combining Reverse Debugging and Live Programming towards Visual Thinking in Computer Programming*. Stellenbosch University, 2015, 94 p.
11. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice. *Proc. Workshop on Binary Instrumentation and Applications, WBIA*, 2009, pp. 62–71. doi: 10.1145/1791194.1791203

Авторы

Доронин Олег Владимирович – инженер-математик, ITIVITI, Санкт-Петербург, 190000, Российская Федерация; аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, ORCID ID: 0000-0003-4209-8440, dorooleg@yandex.ru

Дергун Карина Ильдаровна – старший ИТ-инженер, ПАО Сбербанк, Санкт-Петербург, 195112, Российская Федерация; студент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, ORCID ID: 0000-0001-7350-6780, dergun_karina@mail.ru

Authors

Oleg V. Doronin – applied mathematician, ITIVITI, Saint Petersburg, 190000, Russian Federation; postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, ORCID ID: 0000-0003-4209-8440, dorooleg@yandex.ru

Karina I. Dergun – Senior IT-engineer, PAO Sberbank, Saint Petersburg, 195112, Russian Federation; student, ITMO University, Saint Petersburg, 197101, Russian Federation, ORCID ID: 0000-0001-7350-6780, dergun_karina@mail.ru

[1] Изменения можно найти по адресу: https://github.com/dorooleg/relacy/tree/development.

[2] С полным списком проблем и идей для модернизации библиотеки RRD можно ознакомиться по адресу https://github.com/dorooleg/relacy/issues.

Дергачев Андрей Михайлович – кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com

Ключев Аркадий Олегович – кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 56429748500, ORCID ID: 0000-0002-3892-8424, kluchev@gmail.com

Andrey M. Dergachev – PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com

Arkady O. Kluchev – PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 56429748500, ORCID ID: 0000-0002-3892-8424, kluchev@gmail.com