УДК 004.415.53

Использование системы LLVM при динамическом поиске состояний гонок в программах

Д.Н. Ковега [1], В.А. Крищенко [1]

[1] МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

В многопоточных программах может возникать состояние гонок ввиду отсутствия синхронизации при доступе к памяти. Для сбора информации о событиях динамическими методами выявления таких гонок требуется либо виртуальная машина, либо инструментирование исполняемого кода. В работе предложен метод динамического поиска гонок, использующий отношение предшествования и ограничение истории обращений. Метод реализован для анализа программ на языке C, для инструментирования исходного кода на этапе трансляции используется система LLVM. Как показывают проведенные эксперименты, применение предложенного метода позволяет сохранить накладные расходы на поиск гонок на приемлемом уровне.

E-mail: arhibot@gmail.com, kva@bmstu.ru

Ключевые слова: *состояния гонок, динамический поиск гонок, инструментирование кода, тестирование многопоточных программ, LLVM.*

При разработке параллельных приложений часто возникают трудные в локализации ошибки, вызванные нарушениями в синхронизации доступа потоков в общей памяти, которые приводят к состоянию гонки между потоками. Для поиска состояний гонок при работе с разделяемой памятью в многопоточных программах существуют инструменты нескольких классов, наиболее применимым являются средства динамического поиска гонок. Такие средства во время выполнения программы ведут историю обращений к памяти и операций синхронизации для дальнейшего анализа.

Генерация потока событий при динамическом поиске гонок может реализоваться либо при исполнении программы в виртуальной машине, либо посредством модификации кода программы для сбора информации о доступе к памяти и к функциям синхронизации. Существующие реализации на базе виртуальной машины, такие как Helgrind [1] и Thread Checker, осуществляют интерпретацию исполняемого файла, что позволяет проводить анализ программы и используемых библиотек без необходимости повторной компиляции или внесения изменений в бинарные файлы. Основной проблемой таких инструментов являются высокие накладные расходы на интерпретацию анализируемого машинного кода: так, детекторы Thread Checker и Helgrind снижают производительность анализируемой программы в 10—200 раз [2].

Подход на основе инструментирования кода заключается в добавлении в исполняемый файл кода для генерации событий при до-

ступе к памяти. Кроме того, для этого подхода требуется переопределение системных функций синхронизации. Использование инструментирования кода для динамического поиска гонок могло бы сократить накладные расходы. В качестве средства инструментирования кода можно применять Low Level Virtual Machine (LLVM) — универсальную систему анализа, трансформации и оптимизации программ, в которой используется промежуточный код с RISC-подобными инструкциями [3].

Метод динамического поиска гонок. Динамические средства поиска гонок проводят анализ программы в ходе ее выполнения, проверяя журналы обращений по всем адресам памяти и операций с примитивами синхронизации (рис. 1). Результатом анализа является сообщение о найденных гонках, включающее в себя стеки вызовов потоков выполнения, которые привели к состоянию гонки при обращении к какой-либо области памяти.

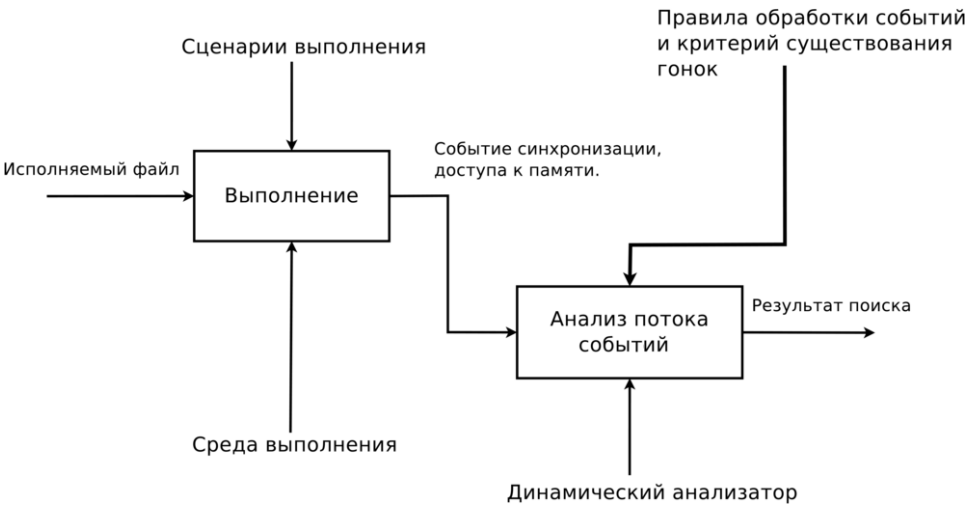

Рис. 1. Динамический поиск гонок

Динамический анализатор основан на анализе потока событий, генерируемых программой во время выполнения. Программа является «черным ящиком», подающим поток событий в регистратор событий, который обрабатывает полученные событий и сохраняет их в журнал выполнения потока. Далее анализатор по этому журналу проверяет наличие гонок, если текущее событие может быть источником гонок (рис. 2).

Выделяют следующие типы событий, необходимых анализатору [4]:

– события доступа к памяти на чтение или запись;

– захват или освобождение блокировок семафоров или мьютексов;

– прием или генерация событий, упорядочивающих выполнение программы.

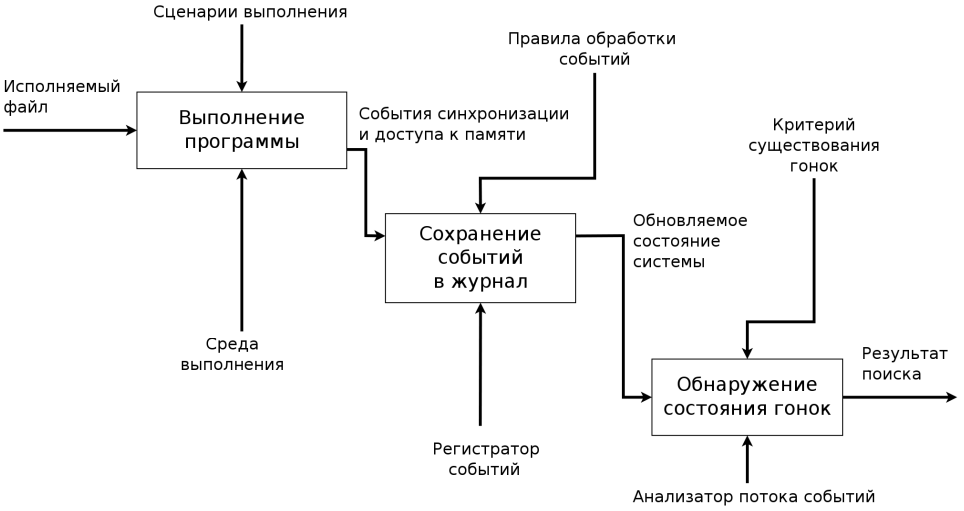

Рис. 2. Схема генерации и обработки событий

Средства динамического поиска гонок в настоящее время обычно строят на основе анализа отношения предшествования. Подход на базе отношения предшествования сообщает о найденной гонке, если два потока или более используют общую память, причем доступы к памяти причинно неупорядочены по определению из [5]. Выполнение программы разбивается на сегменты, которые определяют области потока выполнения, содержащие только обращения к памяти.

Динамические анализаторы требуют поддержки журнала обращений для каждой области памяти, к которой выполнялся доступ. С увеличением размера журналов это создает дополнительные накладные расходы как памяти, так и при анализе существования гонки. Пусть R — множество сегментов, в которых осуществлялось чтение некоторой области памяти m, а W — множество сегментов, в которых осуществлялось обращение на запись к этой же области памяти m. Для уменьшения размеров хранимой истории в списке сегментов R хранятся только сегменты, отвечающие условию: любой сегмент из множества R произошел после любого сегмента из множества W или параллелен с ними, а в списке сегментов присутствуют W только попарно неупорядоченные сегменты [6].

Анализатор гонок с использованием системы LLVM. Разработанный анализатор состоит из следующих компонентов (рис. 3):

– модуль компилятора для модификации генерируемого машинного кода;

– библиотека переопределенных функций POSIX для работы с потоками и примитивами синхронизации;

– библиотека регистрации событий;

– библиотека, содержащая реализацию гибридного алгоритма поиска гонок.

Рис. 3. Структура динамического анализатора

Модуль компилятора генерирует модифицированный исполняемый файл, который во время выполнения проводит генерацию событий. Библиотека, содержащая переопределенные функции, необходима для генерации событий, упорядочивающих выполнение программы. Библиотека регистрации событий предоставляет интерфейс, через который события фиксируются в журнале выполнения потоков. Библиотека, содержащая реализацию гибридного алгоритма поиска гонок на основе журнала выполнения потоков, осуществляет поиск гонок и формирование отчета на основе данных журнала выполнения.

Разработанный анализатор проводит анализ наличия гонок непосредственно во время выполнения программы и работает в том же адресном пространстве, что и анализируемая программа.

Компиляция из языка программирования в промежуточное представление реализуется в соответствующем трансляторе LLVM: для языка С в LLVM используется компилятор CLang. После преобразования транслятором исходного кода в промежуточный код (IR-код) происходит наложение цепочки оптимизаторов и получение объектного файла для заданной архитектуры. Далее для получения исполняемого файла выполняется компоновка объектных файлов с необходимыми библиотеками (рис. 4).

Система LLVM предоставляет возможность включения собственных обработчиков AST-деревьев в процесс компиляции IR-кода в машинный код и предлагает API для них. Обработчики проводят преобразования и оптимизацию кода, сбор аналитических данных во время компиляции. Благодаря тому, что обработчики являются разделяемыми библиотеками, для реализации собственного обработчика необязательно вносить изменения в дерево исходных кодов LLVM.

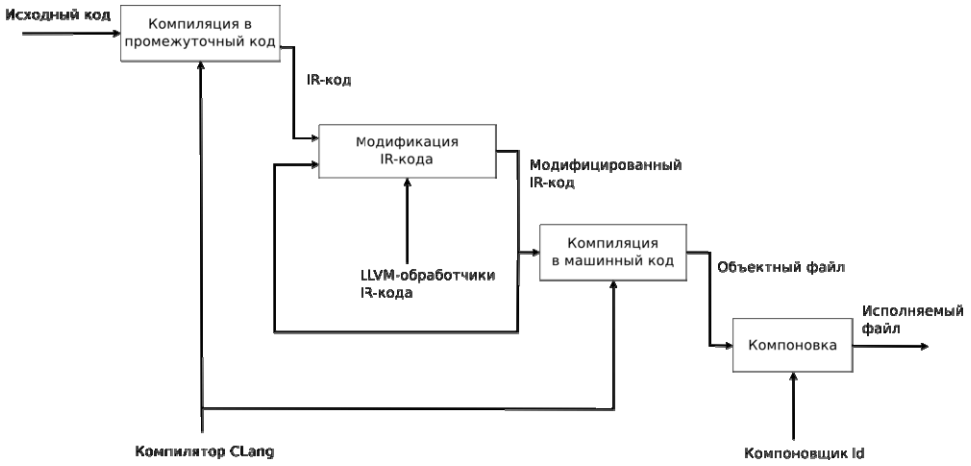

Рис. 4. Использование обработчиков промежуточного кода

В данной работе для генерации модифицированного исполняемого файла реализован обработчик AST-деревьев в рамках платформы LLVM. В зависимости от цели оптимизатора LLVM представляет несколько типов базовых проходов. Класс `ModulePass` является наиболее общим проходом, позволяющим добавлять, удалять функции или изменять код любой функции. Для определения корректного прохода уровня модуля необходимо создать его класс-наследник и переопределить функцию `runOnModule`.

Разработанный обработчик проводит анализ на уровне модулей программы, поэтому он является наследником класса `ModulePass`. Модификация модуля программы проходит в два этапа:

1) добавление деклараций функций, вызовы которых будут вставлены в код;

2) проход по всем функциям модуля для вставки необходимых инструкций.

Созданный обработчик осуществляет добавление вызовов следующих функций библиотеки регистрации событий:

`void new_segment(int line, char* file, char* dir)` — генерация события входа в новый сегмент выполнения программы;

`void mem_access(int* addr, short is_write)` — генерация события обращения к памяти на чтение или запись;

`void enter_function(char* name)` — генерация события входа в функцию;

`void leave_function()` — генерация события о выходе из функции.

Таким образом, разработанное устройство анализирует существование гонок и поддерживает журналы обращений к памяти с помощью модификации процессов выполнения потоков, проводящих вызов специальных функций во время выполнения. События входа и

выхода из функций нужны для построения стека вызовов, приведшего к гонкам.

Эксперимент с созданным анализатором. Для определения накладных расходов созданного анализатора сравним потребление памяти и процессорного времени при выполнении одной и той же исходной тестовой программы:

– запуск программы без анализатора;

– запуск программы с динамическим анализатором с ограничением глубины истории обращений к памяти;

– запуск программы с динамическим анализатором, но без ограничения глубины истории обращений к памяти.

Тестовая программа запускает 20 потоков выполнения, в каждом потоке выполнения выделяется 20 Мб памяти. Далее по 1000 случайным адресам, которые принадлежат областям памяти, выделенным в каждом потоке, происходит запись или чтение. Каждые 500 итераций происходит выделение дополнительных 20 Мб памяти или уменьшение выделенной памяти на 20 Мб. На рис. 5 показано использование памяти при работе тестовых программ. Видно, что тестовая программа без ограничения глубины истории практически линейно увеличивает потребляемую память. В то же время программа с ограничением глубины истории имеет практически постоянное потребление памяти и превышает в среднем на 20 Мб потребление памяти программы, запущенной без анализатора.

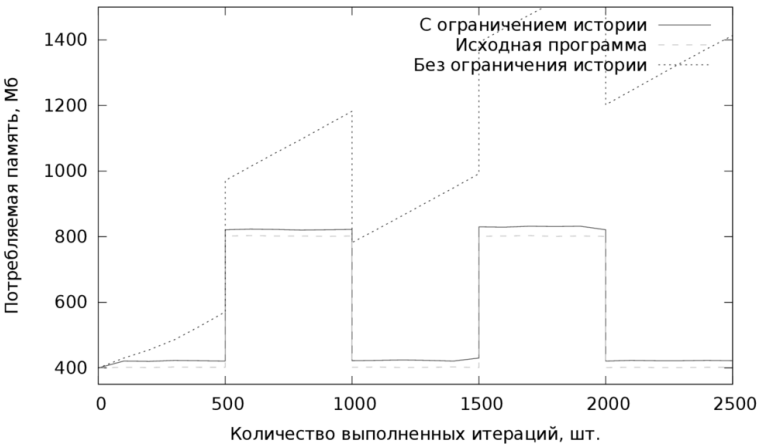

Рис. 5. Потребление памяти анализируемой программой

Рассмотрим использование процессорного времени тестовыми программами (рис. 6). Эксперимент проводился на компьютере с четырьмя ядрами. Поскольку тестовая программа является многопоточной, то можно ожидать, что во время выполнения будут востребованы все доступные ядра. На рис. 6 видно, что программа без использования анализатора занимает все ядра и загружает процессор

практически на 100 %. Программы, запущенные с добавленным в них анализатором, используют не более 55 % процессорного времени. Это связано с применением глобальных блокировок между потоками в анализаторе гонок для того, чтобы безопасно добавлять события в журнал выполнения программы и осуществлять поиск гонок. Среднее использование процессора программой (при ограничении истории) составляет 52 % и практически не изменяется со временем. В то же время на графике видно, что использование процессора программой, когда история не ограничивается, падает с 55 до 27 %. Это обусловлено тем, что процедура анализа журнала выполнения происходит в единственном потоке, а так как размер журнала выполнения в программе без использования ограничений истории растет линейно от числа обращений к памяти, то большее время потоки ожидают получение доступа к журналу.

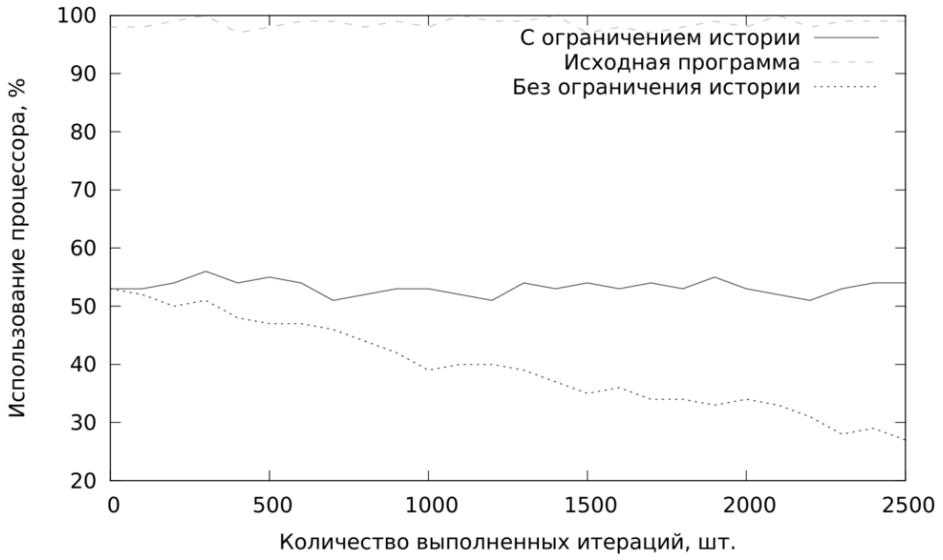

Рис. 6. Потребление процессорного времени анализируемой программой

В результате проделанной работы разработан динамический анализатор поиска гонок в программах, реализованных на языке C. Анализатор использует систему LLVM для модификации исполняемого кода на этапе компиляции. Для этого разработан обработчик AST-деревьев для платформы LLVM и переопределены системные функции POSIX по работе с примитивами синхронизации и управления потоками выполнения. Использование анализатора показало, что при его применении накладные расходы остаются незначительными.

Работа выполнена при частичной поддержке Российского фонда фундаментальных исследований (грант № 13-07-00918).

СПИСОК ЛИТЕРАТУРЫ

1. Nethercote N., Seward J. Valgrind: A framework for heavy-weight dynamic binary instrumentation // Proc. of the 2007 Programming Language Design and Implementation Conf. 2007. Vol. 26. No 6. P. 89–100.
2. Marino D., Musuvathi M., Narayanasamy S. LiteRace: effective sampling for lightweight data-race detection // PLDI. 2009.
3. LLVM Programmer's Manual. URL: http://llvm.org/docs/ProgrammersManual.html
4. A theory of data race detection / U. Banerjee, B. Bliss, Z. Ma, P. Petersen // Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. PADTAD'06. N. Y.: ACM, 2006. P. 69–78.
5. Lamport L. Time, Clocks and the Ordering of Events in a Distributed System // Communications of the ACM. 1978. Vol. 21. No 7. P. 558–565.
6. Serebryany K., Iskhodzhanov T. ThreadSanitizer: Data race detection in practice // Proc. of the Workshop on Binary Instrumentation and Applications. WBIA'09. N. Y.: ACM, 2009. P. 62–71.

Статья поступила в редакцию 25.10.2012