электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.432.2

Обобщённый метод обнаружения оптимизационно-нестабильного поведения в коде на языке C

***Красиков А.С.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
кафедра «Программное обеспечение ЭВМ и информационные технологии»

Научный руководитель: Ломовской И.В., старший преподаватель
«Программное обеспечение ЭВМ и информационные технологии»
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Неопределённое поведение (англ. undefined behavior) – такое поведение, которые может возникнуть при использовании ошибочных или некорректных программных конструкций или данных, на которое Международный Стандарт не накладывает никаких ограничений [1].

Так как ограничений никаких стандартом не накладывается, то неопределённое поведение полностью зависит лишь от реализации компилятора и случайных параметров наподобие состояния памяти или сработавшего прерывания. Оно может варьироваться от полного игнорирования, выдачи предупреждающих сообщений и завершении программы. Стандарт языка C явно выделяет случаи неопределённого поведения.

Неопределённое поведение используется для того, чтобы упростить структуры компиляторов языка и ускорить целевой код. Например, стандарт языка C C99 не определяет результат деления на 0, численное переполнение, обращение по нулевому указателю и выход за пределы массивов. Таким образом, в программах на языке C, деление на 0 иногда может не приводить к ошибкам времени выполнения, как, например, происходит на PowerPC.

Идея неопределённого поведения заключается в том, что оно должно отлавливаться на уровне логики программистом, а не на уровне скомпилированного кода.

С точки зрения оптимизирующего компилятора, ему на вход подан детерминированный (*англ.* well-formed) код, значит в нём нет и неопределённого поведения, поэтому он может удалить этот блок, т.к. условие никогда не сработает. Код с неопределённым поведением после прогона через оптимизирующий компилятор становится потенциально оптимизационно-нестабильным.

Принимая во внимания недостатки недавних подходов поиска неопределённого и оптимизационно-нестабильного поведения [2,3], а именно невозможность применения в общем случае для любых видов оптимизаций, была сделана попытка решить проблему появления оптимизационно-нестабильного поведения с меньшей эффективностью, но для общего случая.

Проблему оптимизационно-нестабильного поведения можно рассматривать как преобразование оптимизирующим компилятором кода с неопределённым поведением, что выливается в задачу сравнения случаев использования конструкций и данных с неопределённым поведением до оптимизации кода и после. При рассмотрении оптимизационно-нестабильного поведения используют исходный код на языке C, но исходный код после сразу первого прогона оптимизирующего компилятора как-минимум крайне сложно привести в состояние, хотя бы похожее на первоначальное.

Сравнение случаев использования конструкций и данных с неопределённым поведением можно рассматривать как выделение состояния характеристик у сравниваемых множества использования неопределённого поведения и их последующее сопоставление. Эти же ключевые состояния можно выделять и на внутреннем представлении компилятора. Можно говорить о том, что исходный код преобразуется один в один без изменений во внутреннее представление компилятора после семантического прогона компилятора, кроме случаев примитивной оптимизации, как удаление кода следующего после exit вызова. Тогда следующим шагом в полученном внутреннем представлении мы должны выделить неопределённые поведения. Косвенно отловить оптимизационно-нестабильное поведение в данном случае можно разметив множества неопределённых поведений и сравнив их.

Мы можем говорить о том, что оптимизирующий компилятор не добавил оптимизационно-нестабильного поведения если код до оптимизирующих прогонов и после обладает сопоставимыми множествами неопределённых поведений. Ключевые этапы и их последовательность предложенного способа изображены на рисунке 1.

Обобщённый способ поиска оптимизационно-нестабильного поведения также требует поддержания в актуальном состоянии механизма определения неопределённого поведения, но в отличии от остальных подходов не требует поддержания в актуальном состоянии стандарту языка C.

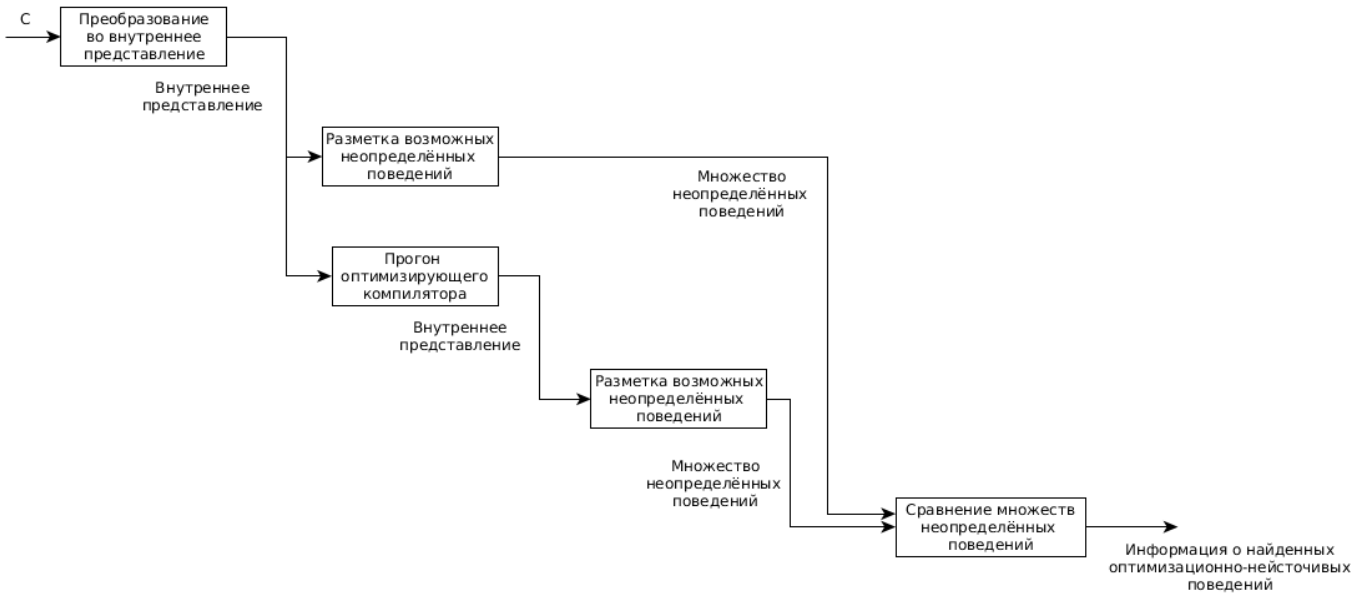

Рис. 1. Обобщённый поиск оптимизационно-нестабильного поведения

Преобразование во внутреннее представление

Очевидным требованием к внутреннему представлению является линейность отображения кода на языке C в него. Так как целью анализа является именно исходный код, модификации исходного кода, происходящие во время преобразования во внутреннее представление, такие как примитивные оптимизации, сильно затрудняют анализ исходного кода.

Дополнительно встаёт проблема того, что транслятор во внутреннее представление тоже может доопределять неопределённое поведение, или скрывать существующее неопределённое поведение. Простейшей иллюстрацией этой проблемой является неопределённое поведение при неоднократном чтении/записи значения переменной в одном выражении. Стандарт не гарантирует порядок выполнения выражения, но распространённые внутренние представления основываются на триплетах (*например* https://gcc.gnu.org/onlinedocs/gccint/RTL.html, http://llvm.org/docs/LangRef.html), таким образом упорядочивая порядок операций. Очевидной проблемой также является платформозависимость внутреннего представления, так как в этих случаях не всегда можно определить неопределённое поведение исходного кода на языке C от его внутреннего представления с учётом особенностей конечной архитектуры.

Тем не менее, распространённые средства компиляции предоставляют (*например* https://gcc.gnu.org/onlinedocs/gccint/RTL.html, http://llvm.org/docs/LangRef.html) форматы внутренних представлений, а также средства трансляции в них лишённые

вышеперечисленных недостатков.

Разметка неопределённого поведения

Очевидно, что статический поиск всех неопределённых поведений в языке С стандарта С99 является неразрешимой задачей, поэтому, в зависимости от конечных требований, искать неопределённые поведения можно с ошибкой первого или второго рода, но выбранный приоритетный тип допущения должен совпадать с допущениями на весь метод целиком. Далее будет рассмотрена разметка неопределённого поведения с допустимой ошибкой первого рода.

Так как оптимизирующие компиляторы во время своей работы опираются на потоки данных и ограничений, то их можно использовать и при разметке неопределённого поведения. Таким образом, если на сущность накладывается ограничение об отсутствии неопределённого поведения в коде, например `x != 0`, а потом идёт работа с `x`, выходящей за рамки этого значения, то результат этого использования может обладать неопределённым поведением.

Можно выделить 3 уровня видов ограничения:

- не ограничено;
- ограничено по предположению об отсутствии неопределённого поведения;
- ограничено по проверке.

При этом, существовать могут переходы только из уровней ниже на уровни выше, т.е. из 1 в {2,3} и из 2 в {3}.

Сравнение множеств неопределённых поведений

Сравнение множеств неопределённых поведений сложно. Для решений обработки переупорядоченного кода встаёт необходимость проверки потоков данных, поэтому одним из инструментов для сравнения является символьное выполнение для проверки возможности перевода начального множества неопределённых поведений в конечное с помощью операций, не уменьшающих итоговую мощность замкнутого множества. Иными словами, задачу сравнения можно формализовать как поиск отображения множества начальных неопределённых поведений в конечное множество неопределённых поведений. Следует заметить, что отображение в общем случае сюръективно и не всегда может быть биективным, так как неопределённые поведения в следствии перестановок и объединения вычислений могут объединяться, но не разделяться, например в листинге 1 приведён пример при котором начальное множество неопределённых поведений однозначно

свернётся.

Листинг 1. Код с неуникальным потенциальным неопределённым поведением

```
1  int p = *l;
2  int d = *l;
3  return p * d;
```

Рассматриваемые классы оптимизаций

Данный подход рассматривается только для случая локальных оптимизаций внутри отдельных функций. Именно благодаря этому ограничению мы можем упростить сравнивать множества неопределённых поведений приведённым выше способом. При наличии кросс-процедурных оптимизаций конечные множества неопределённых поведений может оказываться больше начального. В данном случае можно учитывать факт кросс-процедурных оптимизаций, чтобы они помечали все добавленные операции как «внешние», но для этого придётся изменять код оптимизирующих методов, который не всегда может быть доступен. Но, тем не менее для рассмотрения остаётся большой класс оптимизирующих методов, порождаемое неопределённое поведение которых становились целями исследований в последнее время [3,4], такие как удаление недостижимого кода, удаление лишних вычислений.

Встраивание в компиляторы

Стоит так же рассмотреть возможные способы реализации описанного подхода поиска оптимизационно-нестабильного поведения. Одним из естественных современных способов встраивания является реализация его как как модуля для компилятора, например, такую функциональность поддерживают компиляторы Clang и GCC. При подключения внешнего модуля в качестве прогона компилятора в них можно указывать порядок их запуска, таким образом можно реализовать метод в качестве двух прогонов, первый из которых будет запускаться до всех прогонов оптимизирующего компилятора, а второй – после.

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

Список литературы

1. Internation Standard Organization C Standard 1999 / ISO, ISO/IEC 9899:1999 draft. WG14

ed., 1999. Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, accessed: 27.11.2014.

2. Ellison, C. Defining the undefinedness of C. University of Illinois, 2012. Available at: http://hdl.handle.net/2142/30780, accessed: 27.11.2014.
3. Xi Wang, Zeldovich N., Frans Kaashoek M., Armando Solar-Lezama Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. MIT CSAIL, 2013. Available at: http://dl.acm.org/citation.cfm?doid=2517349.2522728, accessed: 27.11.2014.
4. Will Dietz, Peng Li, Regehr J., Vikram Adve Understanding Integer Overflow in C/C++. 34th International Conference on Software Engineering (ICSE), proceedings: Zurich, Switzerland, June 2-9, 2012. P. 760-770.