Метод инструментирования кода на этапе компиляции для направленной отладки оптимизирующих преобразований

Д. А. Максименков <shark@mcst.ru>
ПАО «МЦСТ», РФ, г. Москва, Ленинский пр-т, д. 51

Аннотация. В статье рассматривается проблемы отладки оптимизирующих компиляторов. В качестве эффективного способа повышения надежности производимых компилятором оптимизирующих преобразований автором предлагается новый метод инструментирования кода программы на этапе компиляции. Особенностью описываемого в статье метода является то, что он предназначен в первую очередь для отладки конкретных проблемных оптимизаций, а не самих тестов, и позволяет верифицировать корректность формируемого оптимизацией кода для произвольных входных данных запускаемой задачи. Предлагаемый метод был успешно использован для поиска и выявления нерегулярно проявляющихся ошибок в программах с асинхронно работающим кодом.

Ключевые слова: тестирование, инструментирование кода, отладка оптимизирующих компиляторов.

DOI: 10.15514/ISPRAS-2015-27(6)-1

Для цитирования: Максименков Д.А. Метод инструментирования кода на этапе компиляции для направленной отладки оптимизирующих преобразований. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 7-20. DOI: 10.15514/ISPRAS-2015-27(6)-1.

1. Введение

Одной из проблем при отладке оптимизирующего компилятора является выявление факта того, что в процессе работы скомпилированной задачи была допущена ошибка. Так, некоторые ошибки, допущенные компилятором, могут не приводить к аварийному завершению задачи, а влиять лишь на вычисляемое конечное значение в программе. Если получаемый результат вычислений отличается от эталонного незначительно в пределах допустимой погрешности, то задача считается отработавшей корректно. Таким образом, например, устроены пакеты тестовых задач spec[1]. В этом случае, обнаружить, что в процессе работы компилятора была допущена ошибка, оказывается довольно затруднительно. Зачастую, преимущественно в

больших задачах некоторые неправильно произведенные вычисления могут вообще не влиять на конечный результат работы программы. С точки зрения пользователя, такие ошибки компилятора не являются критичными, т.к. не оказывают какого-либо влияния на конечный результат вычисления конкретной задачи. Но для целей отладки оптимизирующего компилятора выявление некорректных преобразований, произведенных компилятором, являются важной задачей. Ведь такие оптимизации при компиляции других программ могут привести к их неправильной работе или даже слому. Из-за ограниченности набора тестовой базы, вычислительных мощностей, человеческих и временных ресурсов возможность выявления любых ошибок оптимизатора (а не только тех, которые приводят к явному слому или неправильной работе задачи) является эффективным средством повышения надежности оптимизирующего компилятора.

В случае, когда ошибка, допущенная компилятором, выявляется самой задачей в процессе ее работы (падение или различие в печатаемых результатах вычислений), также возникает необходимость в локализации момента появления ошибки в коде. Как правило, неправильно произведенные вычисления приводят к падению задачи далеко от момента самой ошибки или отражаются только в конечном подсчитанном значении после завершения работы всей задачи, что сильно затруднит анализ проблемы.

Одним из широко используемых методов поиска ошибок в компиляторе является метод инструментирования кода. Например, проекты Valgrind[2], AddressSanitizer[3] предназначены для отладки программ. В качестве отлаживаемой программы можно использовать, в том числе, и сам оптимизирующий компилятор. При этом в бинарный код оптимизирующего компилятора встраиваются различные проверки, предназначенные для выявления ошибок определенного класса, таких как выход за границу массива, использование неинициализированных данных, обращение по невалидному адресу в коде самого компилятора. Подавая такому компилятору различные входные данные – т.е. разнообразные тесты, можно находить контексты для работы большинства анализов и запуска целого ряда оптимизаций, проверяя тем самым корректность работы написанного кода компилятора. Т.е. данные методы позволяют отлаживать компилятор как пользовательскую задачу. При этом не проверяется логика работы самих оптимизаций, проводимых над кодом задачи. Существуют методы [4], которые позволяют на уровне исходных кодов инструментировать тесты, используемые в качестве входных данных для оптимизирующего компилятора. Тесты с такими встроенными внутренними самопроверками во время своей работы в определенных местах программы проверяют корректность производимых промежуточных вычислений на соответствие эталонным, заранее подсчитанным константам. Инструментированные таким образом задачи способны эффективно выявлять ошибки в различных оптимизациях компилятора, если код теста в результате произведенных

компилятором преобразований оказался неэквивалентным исходному и стал работать по-другому. Однако в результате такого инструментирования изменяется сам исходный код задачи, что в конечном итоге может повлиять на результаты анализов и на применение различных оптимизаций. В результате возможна ситуация, когда ошибка проявляется при компиляции исходного кода теста и перестает проявляться после инструментирования этого же кода дополнительными проверками. Помимо этого, встроенные проверки рассчитаны на фиксированный набор входных данных. Если запускаемый тест предполагает возможность использования различных исходных данных (например, [1]), то для каждого входного набора данных инструментирование кода придется проводить заново.

Предлагаемый в статье метод производит инструментирование кода задачи в момент применения оптимизирующего преобразования, т.е. на этапе компиляции теста. Поэтому исходный код задачи никак не модифицируется, а встраиваемые проверки не зависят от подаваемых входных для теста данных. Особенностью описываемого в статье метода является то, что он предназначен для отладки отдельных проблемных оптимизаций и позволяет верифицировать корректность формируемого оптимизацией кода для произвольных входных данных запускаемой задачи. Встраиваемые в код самопроверки не влияют на результаты анализов и, как следствие, на применимость отлаживаемой оптимизации. Иначе говоря, они не вносят возмущение в работу компилятора, т.к. реализуются непосредственно после формирования кода производимой оптимизации и лишь дополняют его. Так же, как и другие методы, данный метод позволяет оперативно выявлять ошибки и локализовывать место и контекст их проявления в работающем коде программы. Метод применим для широкого класса оптимизаций и способен находить ошибки в оптимизирующих преобразованиях даже в тех случаях, когда сам тест проблем не выявляет, а известные способы инструментирования программ не способны их обнаруживать в силу ограниченности класса выявляемых ими проблем.

2. Ошибки, возникающие при работе оптимизаций

Для возможности создания эффективно исполняемого кода компилятор преобразует задачи, написанные на разных высокоуровневых языках программирования, в команды, представляющие собой внутренний абстрактный язык компилятора — промежуточное представление, удобное для дальнейшей работы оптимизаций. Оптимизации последовательно применяются к операциям промежуточного представления, тем самым постепенно модифицируя код программы в терминах внутреннего языка, делая его более эффективным. В конечном итоге после применения всей цепочки оптимизаций полученное финальное промежуточное представление оптимизированной программы преобразуется в ассемблер целевой

архитектуры. Далее, уже с помощью ассемблера и линковщика, формируется объектный и исполняемый код.

Количество применяемых оптимизаций над командами промежуточного представления может быть довольно большим и зависеть как от режима компиляции (от подданных программистом опций компиляции), так и от самой задачи (компилятор может динамически, в процессе своей работы принимать решение о использовании тех или иных оптимизаций). Оптимизации применяются пофазно: от одного промежуточного состояния представления к другому. Одна фаза может содержать в себе запуск сразу несколько оптимизаций, работающих сообща. Количество таких последовательно выполняемых фаз в оптимизирующем компиляторе, как правило, достигает нескольких сотен.

Оптимизирующие преобразования производятся над одной или несколькими операциями промежуточного представления. Компилятор в процессе оптимизаций заменяет их на другие, более эффективные команды. Получаемая последовательность операций должна быть эквивалентна преобразуемой, изначальной группе команд. В противном случае оптимизированный код будет содержать в себе ошибки и может работать неверно.

Ошибки в оптимизациях, проявляющиеся на широком диапазоне входных данных задачи, как правило, находятся довольно легко. Например, если выражение

```
a = b * c;
```

в процессе оптимизаций компилятором было ошибочно заменено на

```
a = b + c;
```

то практически при любых входных значениях переменных b и c результат преобразования окажется неверным. Такую ошибку в программе будет обнаружить несложно — выражение будет получать неверный результат практически при любых входных данных. Причем вычисляемое значение будет отличаться от ожидаемого значительно.

Однако существует другой класс ошибок, проявляющийся только при определенном наборе входных данных. Для получения более эффективный код часть оптимизаций производит преобразование, которое будет эквивалентным лишь при выполнении определенного ряда условий, например, при ограниченном наборе входных аргументов операций, к которым применяется данная оптимизация.

Так, выражение

```
f2 = f1 * 0.0;
```

где f1 и f2 — это вещественные числа,

можно заменить на эквивалентное выражение

```
f2 = 0.0;
```

Однако такое преобразование будет эквивалентным не для любых входных значений f1. Например, при f1 = nan (не число), согласно стандарту работы вещественной арифметики IEEE-754 [5] значение f2 также должно принимать значение nan. Производить описанное выше преобразование можно лишь в том случае, когда компилятору достоверно известно, что в вещественных вычислениях участвуют только нормальные числа (т.е. среди них нет nan, qnan, inf и т.п.). В данном случае такую информацию может сообщить компилятору программист с помощью специализированной опции компиляции, указав оптимизациям, что такие преобразования разрешены.

Большинство оптимизаций применяются как раз в предположении выполнения ряда условий, определяемых компилятором статически. Т.е. решение о выполнение условий, для которых применяемое преобразование будет эквивалентным, принимается в процессе работы компилятора. И по результатам этого решения одни операции заменяются на другие, эквивалентные изначальным, но при выполнении определенных условий. Выявлять ошибки в таких преобразованиях бывает довольно сложно, т.к. для их проявления необходимо появление контекста, не учтенного оптимизацией. Т.е. для некоторых входных данных задачи построенный оптимизатором код будет некорректным. И чем меньше диапазон входных данных, для которых сформированный оптимизацией код работает неверно, тем реже проявляется ошибка и тем сложнее будет ее обнаружить.

3. Описание метода инструментирования кода при работе оптимизаций

Покажем суть метода на простом примере. Допустим, выражение

```
b = a / 2;
```

в процессе работы одной из оптимизаций заменили на

```
b = a >> 1;
```

Данная замена будет эквивалентной для всех положительных значений a, а также для a = 0 и для ряда отрицательных значений a (а именно, четных). Для нечетных отрицательных a преобразование даст другой результат, отличающийся на 1. В данном случае диапазон значений a, при которых оптимизация работает корректно, оказывается в несколько раз больше, чем тот набор значений, при которых будет возникать ошибка. Поэтому очень важным является гарантия выполнения условий применимости оптимизации перед использованием их компилятором. В данном примере нужно гарантировать, что a — неотрицательное или четное число.

Рассматриваемый в работе метод позволяет обнаруживать преобразования, некорректно произведенные оптимизирующим компилятором, в т.ч. и для тех

случаев, когда ошибка проявляется на ограниченном контексте входных условий, а значит, имеет редкое (а в некоторых случаях и недетерминированное) проявление, т.е. когда компилятор на основе статического анализа производит замену одной группы операций на другую (эквивалентную, например, только для определенного контекста). В таких случаях возможна ошибка как в самом преобразовании, так и в корректности определении контекста для применяемого преобразования.

Для проверки эквивалентности замены одной группы операций на другую было предложено не удалять первую группу операций, а оставить ее в коде программы и использовать для получения эталонного значения с целью последующего сравнения его с результатом, получаемым от новой группы операций. В случае несличения подсчитанных значений от двух групп операций работа теста аварийно завершается, что оперативно сигнализирует о найденной ошибке. Попутно с этим, предложено сохранять информацию о контексте (например, значение аргумента для модифицируемой операции), который привел к проявлению ошибки. Т.е. на рассмотренном выше примере, если до применения оптимизации имелось выражение

```
b = a / 2;
```

то после преобразования изначальный код дополняется динамической проверкой:

```
b1 = a / 2;
b = a >> 1;
if(b != b1) Error(a);
```

Наравне с новым, более эффективным способом вычисления переменной b (с помощью побитового сдвига вправо) в коде сохраняется оригинальный способ вычисления выражения (с помощью операции целочисленного деления) с последующим сохранением результата во временную переменную b1. Далее идет сличение значений b и b1 и в случае их неравенства происходит аварийное завершение работы теста. Если случится, что переменная a примет отрицательное нечетное значение, при котором производимая оптимизация является некорректной, то об этом сразу же станет известно из-за аварийного завершения программы. Досрочное, аварийное завершение программы позволяет выявлять и локализовать ошибки, допущенные компилятором при построении оптимизированного кода программы, даже если они не влияют на конечный результат работы теста.

С помощью описанного метода можно проверять корректность работы широкого класса простых правил замены одних операций на другие (оптимизации peephole [6]), которых в оптимизирующем компиляторе, как правило, насчитывается несколько сотен. Сохранение изначальной цепочки операций для получения эталонного значения и сравнение его с результатом

вычисления сформированных оптимизацией операций гарантирует выявление ошибки в случае, если компилятор построил неэквивалентный оптимизированный код для подаваемых входных данных.

Еще одной областью применения данного метода в оптимизирующем компиляторе является проверка корректности работы анализов разрыва зависимости между операциями обращения в память. Например, это могут быть операции записи (store) и чтения (load) данных или операции, неявно работающие с памятью, например, команды вызова функций (call) и системных вызовов (syscall). К сожалению, далеко не всегда удается корректно определить независимость операций обращения в память. Адресные аргументы операций load и store, как правило, статически неизвестны или динамически изменяются, например, в цикловых конструкциях. А возможность модификации функциями конкретных ячеек памяти зачастую неочевидна. В компиляторе используются несколько десятков интерфейсов определения независимости для заданных пар операций обращения в память, использующих различные анализы (например, [7], [8], [9]). Для определения независимости операций в коде программы могут также применяться различные подсказки (pragma, restrict), добавляемые, как правило, автором задачи для возможности более эффективной работы оптимизатора, и даже пользовательские опции (-frestrict-params, -frestrict-all, -fstrict-aliasing). Ошибка в любой из перечисленных компонент приведет к ложному признанию независимости для операций чтения и записи в память с совпадающими или пересекающимися адресами.

Цель оптимизирующего компилятора – сформировать как можно более быстро работающий код. Для этого операции в коде переставляются таким образом, чтобы загрузить данные из памяти на внутренние регистры процессора как можно раньше. Операции load могут работать довольно долго в зависимости от того, где именно находятся запрашиваемые данные (в L1, L2, L3-кэш памяти процессора или в ОЗУ). Именно поэтому многие анализаторы стремятся доказать независимость операции load с другими командами (store, call), чтобы разорвать зависимости между конфликтующими операциями. При отсутствии таких зависимостей команды чтения данных из памяти поднимаются и планируются выше по коду, чем операции store и call.

Поясним вышесказанное на примере (рис.1).

Операции в коде до применения оптимизации:	Операции в коде после применения оптимизации:
store [mem1] ← var1	load [mem2] → var2
…	…
load [mem2] → var2	store [mem1] ← var1
	…
	load’ [mem2] → var3
	if (var2 != var3) Error(mem2);

Рис.1

До применения оптимизации, использующей статический анализ адресов mem1 и mem2, в коде промежуточного представления была операция store, пишущая по адресу mem1 значение переменной var1, и стоящая ниже по коду операция load, читающая в переменную var2 данные из памяти mem2. Для случая, когда один из анализов дал положительный ответ о различии адресов mem1 и mem2, оптимизация поднимает операцию load выше store. Чтобы проконтролировать корректность принятого анализом решения, в коде на месте изначальной операции load создают другую, похожую операцию load', работающую по тому же адресу mem2, но сохраняющую значение во временной переменной var3. Такую операцию помечают во внутреннем представлении как заведомо конфликтующую с операцией store, чтобы впоследствии к ней уже нельзя было применить похожее преобразование повторно. Далее результаты прочитанных значений в переменных var2 и var3 сравнивается, и в случае их несоответствия происходит аварийное завершение работы программы с фиксацией факта ошибки в работе оптимизации для адреса mem2.

По имеющейся многолетней статистике ошибок надежности, выявленных в процессе отладки оптимизирующих компиляторов, ложное признание независимости конфликтующих операций load и store — одна из самых часто встречающихся ошибок оптимизирующего компилятора. Применение описываемого метода позволяет выявить ошибки, связанные с некорректным разрывом зависимости между конфликтующими операциями обращения к пересекающимся ячейкам памяти и последующей их перестановкой.

4. Применение метода на практике

Предлагаемый в работе метод был использован для поиска ошибок и отладки оптимизирующего компилятора для процессора Эльбрус [10] при работе с асинхронным механизмом предварительной подкачки данных из массивов, находящихся в памяти. Асинхронность работы предварительной подкачки данных из памяти приводила к случайному проявлению ошибок в разных местах программы (например, на разных итерациях цикла). Для такой ситуации было крайне важным в случае проявления ошибки идентифицировать момент возникновения проблемы и контекст, при котором проявлялась ошибка. Асинхронная программа запускается в определенный момент работы основного кода программы и далее работает параллельно ему. Ошибка могла возникать (или не проявляться) в зависимости от того, в какой именно момент происходит считывание данных из памяти. Предварительная подкачка данных для массивов из памяти устроена следующим образом. Вначале компилятором определяется место в коде основной программы, начиная с которого возможен запуск асинхронной подпрограммы — момент начала предварительной подкачки (операция bap). Перед запуском самой подпрограммы предварительной подкачки настраиваются специализированные регистры, определяющие параметры считываемых

данных: начальный адрес памяти, по которому будут загружаться данные, расстояние между последовательными чтениями, направление изменение адреса считывания и т.д. После запуска асинхронной подпрограммы в отдельном потоке считываемые данные из памяти попадают в буфер предварительной подкачки. Все это происходит параллельно работе основного кода. В самом коде программы чтения данных из памяти с помощью обычных операций load заменяются на специализированные операции mova – пересылки уже прочитанных ранее значений из буфера предварительной подкачки в переменные программы. Время доступа при обращении к такому буферу — минимально и постоянно в отличие от операций load. Рассматриваемая оптимизация apb – Array Prefetch Buffer[11] состоит в выявлении регулярных цикловых чтений, построении асинхронного кода и синхронного кода подготовки и запуска асинхронной программы предварительной подкачки; при этом операции load заменяются на mova. Однако далеко не любая операция load может быть заменена на mova. Определение того, какие именно данные можно прочитать заранее, т.е. для каких ячеек памяти нет конфликтующих операций записи (store), и определение момента в коде программы, начиная с которого можно запускать асинхронную подпрограмму, является условием применимости данной оптимизации. Кроме того, ошибка в настройке специализированных регистров, может влиять на корректность работы считываемых значений программой предварительной подкачки.

В данном случае эти условия определяются статически, на этапе компиляции. Т.е. компилятор в процессе своей работы определяет те операции load, которые не конфликтуют с другими операциями, и заменяет их на операции mova. Если независимость операций была определена некорректно (между моментом запуска bap асинхронной программы и операцией mova имеется модификация читаемой ячейки памяти), то в зависимости от того, когда именно асинхронная подпрограмма прочтет данные из памяти (до (область A) или после (область B) конфликтующей операции store) зависит корректность почитаемого значения (см рис.2). Отсюда и появляется недетерминированное поведение оптимизированной программы: от запуска к запуску данные, получаемые с помощью операции mova, могут содержать в себе корректные или неверные значения.

Код до применения оптимизации apb:

```
store [mem1] ← %reg1
…
load [mem2] → %reg2
```

Код после применения оптимизации apb:

```
bap
// область A
store [mem1] ← %reg1
// область B
mova → %reg2
```

Рис.2

При равенстве адресов mem1 и mem2 (или даже их частичном пересечении, с учетом формата операций load и store) получим некорректный код, который может загружать различные значение в регистр %reg2.

Чтобы можно было оперативно выявлять данный класс ошибок и идентифицировать условия, при которых оптимизация apb отрабатывает некорректно, в компиляторе, на фазе apb был реализован отладочный режим генерации кода с динамической проверкой результата. По отладочной опции компилятора оптимизация apb не удаляет операцию load, к которой применяется преобразование, и в дополнение к новой созданной операции mova добавляет код самопроверки (см рис.3).

Код до применения оптимизации apb:	Код после применения оптимизации apb:

store [mem1] ← %reg1
…
load [mem2] → %reg2
	bap
	// область A
	store [mem1] ← %reg1
	// область B
	mova → %reg2
	load [mem2] → %reg3
	cmp %reg2, %reg3 → %reg3
	div mem2, %reg3

Рис.3

В данном случае, результаты операций load и mova сохраняются в регистрах %reg2 и %reg3 соответственно. Регистр %reg3 при этом используется как временный регистр для хранения промежуточных значений. Если асинхронная подпрограмма предварительной подкачки загрузила данные в буфер до модификации памяти операцией store, то в случае mem1 = mem2 результат операции сравнения cmp двух регистров %reg2 и %reg3, содержащих разные значения, будет ложным (т.е. равным нулю). Далее нулевой %reg3 используется в качестве делителя целочисленной операции div. Деление на ноль приведет к исключительной ситуации и аварийному завершению работы программы. В случае, когда прочитанные значения операциями load и mova совпадают, результат операции cmp будет истинным (отличным от нуля). В таком случае, операция целочисленного деления завершится без побочных эффектов. В качестве первого аргумента у операции div используется значение адреса mem2. При возникновении целочисленного деления на ноль доступность (например, под отладчиком) информации о первом аргументе сломавшейся операции даст информацию о условиях проявления ошибки. В данном случае это значение адреса mem2, обращение к которому компилятором было ошибочно признано независимым с другими операциями программы, находящимися после команды bap.

Возможны различные вариации реализации данного алгоритма. Вместо адреса mem2 в качестве делимого можно сохранять другую информацию, например, счетчик цикла, чтобы узнать на какой именно итерации цикла произошла ошибка в программе. Вместо операции целочисленного деления можно использовать другие команды, например, операцию обращения в память, формируя в ней при помощи регистра %reg3 нулевой или корректный адрес[4].

Также нужно отметить компактность встраиваемой динамической проверки, линейность полученного кода и отсутствие операций вызова различных функций (exit, abort и т.д), используемых для аварийного завершения работы программы. В этом случае вносится минимальное возмущение в оптимизируемый код, что благоприятно способствует дальнейшему применению других оптимизаций к модифицируемой цепочки операций. Так, например, операции вызова функций являются полюсом для применения целого ряда оптимизирующих преобразований с ограничением возможности переноса других команд через точку вызова. Код, построенный в виде динамической самопроверки, не препятствует использованию результата из регистра %reg2 в дальнейших вычислениях программы и может работать параллельно с ним.

5. Ограничения метода

Данный метод встраивания в код динамической проверки корректности оптимизаций можно использовать при отладке довольно широкого класса оптимизаций, таких как peephole и перестроение операций, на основе различных анализов зависимостей. Однако для некоторых видов преобразований описанный выше алгоритм неприменим или применим с дополнительными ограничениями. Метод нельзя применять, если среди модифицируемых операций присутствуют вызовы некоторого класса функций, меняющих глобальный контекст, например, при наличии функций печати (printf). Если такой код исполнить два раза (оптимизированный и оригинальный вариант), то функция печати отработает дважды, что нарушит логику программы. В некоторых случаях, данный метод можно применять, но с дополнительными модификациями. Если входные данные для фрагмента кода изменяются в процессе его же работы, то повторное их использование уже будет некорректным. В таком случае нужно предварительно скопировать модифицируемые входные во временные регистры для возможности их повторного использования.

6. Результаты

Как уже было сказано, описанный метод был успешно реализован в оптимизирующем компиляторе для процессоров Эльбрус и использовался при поиске и выявления нерегулярно проявляющихся ошибок в оптимизации apb. По результатам тестирования в течение 8 месяцев отладки было выявлено

порядка двух десятков ошибок надежности в данной оптимизации и несколько ошибок в других оптимизациях, также работающих с операциями предварительной подкачки. Время жизни большинства найденных ошибок оказалось довольно большим (составило несколько лет). После исправления всех выявленных проблем нерегулярные падения оптимизированных кодов больше проявлялись. Помимо этого, тесты со встроенными динамическими самопроверками для операций предварительной подкачки позволили выявить и исправить аппаратные ошибки в механизме предварительной подкачки в процессоре Эльбрус во время его верификации на прототипе, а также успешно используются в настоящее время для отладки и поиска ошибок на прототипах новых процессоров.

Список литературы

[1]. Standart Perfomance Evaluation Corporation. The SPEC Benchmark Suites http://www.spec.org
[2]. Valgrind http://www.valgrind.org
[3]. AddressSanitizer (ASan) http://www.chromium.org/developers/testing/addresssanitizer
[4]. Максименков Д.А., Рогов Р.Ю. Применение метода инструментирования тестовых программ при отладке оптимизирующих компиляторов. Вопросы радиоэлектроники, 2010, вып. 3, стр. 50-61
[5]. Standard for Binary Floating-Point Arithmetic IEEE-754. http://www.ieee.org
[6]. D.A. Lamb, Construction of a Peephole Optimizer. Software – Practice & Experience, 11/ 639-647 – 1981.
[7]. Michael Jind Pointer analysis: Haven't we solved this proplem yet http://www.cs.cornell.edu/courses/cs711/2005fa/papers/hind-paste01.pdf
[8]. Amer Divan, Kathryn S.McKinley, J.Eliot B.Moss. Type-Based Alias Analysis http://web.cs.ucla.edu/~palsberg/tba/papers/diwan-mckinley-moss-pldi98.pdf
[9]. Дроздов А.Ю, Владиславлев В.Е. Межпроцедурный анализ указателей. Информационные технологии, 2005, приложение No 2, стр 35-42
[10]. Волконский В.Ю. Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. Информационные технологии и вычислительные системы - 2004 — Вып.3
[11]. Галазин А.Б, Степаненков, А.М., Ступаченко, Е.В. Программная предварительная подкачка кода для микропроцессора Эльбрус-3М. Информационные технологии, 2007, вып. 11.

Compile the Code Instrumentation Technique for Selective Debugging of Optimizing Transformations

D. Maksimenkov <shark@mcst.ru>
PAO «MCST», 51 Leninsky, Moscow, Russia

Abstract. The paper addresses the problem of an optimizing compiler debugging. A new method for compile-time instrumentation is presented as an efficient approach to improve reliability of optimizing transformations implemented in the compiler. The principal feature of the method is that it aims at debugging the transformations itself rather than debugging the tests, and therefore it allows correctness of the resulting code to be verified on any input data fed into the executed program, i.e. the proposed self-checking instrumentation is orthogonal to particular input data in that sense it is able to detect the bugs on arbitrary data flow the program exhibits during its invocation.

The method can be applied to a wide set of optimizing transormations. And the cases are known when it reveals faults in transformations while the non-instrumented test itself remains fully functional (and other somewhat similar but more limited instrumentations reveal no bugs, too). Among other its features are the compactness of the embedded dynamic checkers, the linearity of code bloat, and also no assumptions on any standard libraries availability is made (eg. no functions like exit(), abort() etc are used to terminate the program if any checker is triggered). It all ensures that the optimized code is influenced minimally so the original sequence of optimizing transormations implemented in the compiler remains applicable (if compared to non-instrumented code).

The described method has demonstrated successfully its usability for detecting and identifying volatile bugs in the optimizing compilers of the Elbrus microprocessor family when they were used for building software with asynchronous control flow. Moreover it allowed to increase the reliability of not only the compiler itself but also the software being compiled as well as it shown its use in complex testing of the microprocessor prototypes.

Keywords: testing, instrumentation code, debugging optimizing compilers

DOI: 10.15514/ISPRAS-2015-27(6)-1

For citation: D. Maksimenkov. Compile the Code Instrumentation Technique for Selective Debugging of Optimizing Transformations. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-1

References

[1]. Standart Perfomance Evaluation Corporation. The SPEC Benchmark Suites http://www.spec.org
[2]. Valgrind http://www.valgrind.org
[3]. AddressSanitizer (ASan) http://www.chromium.org/developers/testing/addresssanitizer
[4]. Maksimenkov D.A., Rogov R.Y. Primenenie metoda instrumentirovaniya testovyh program pri otladke optimiziruyushcih kompilyatorov [Application of instrumentation

test program debugging optimizing compilers]. Voprosy radioelektroniki [Questions electronics], 2010, no. 3, pp. 50-61 (in Russian).

[5]. Standard for Binary Floating-Point Arithmetic IEEE-754. http://www.ieee.org

[6]. D.A. Lamb, Construction of a Peephole Optimizer. Software – Practice & Experience, 11/ 639-647 – 1981.

[7]. Michael Jind Pointer analysis: Haven't we solved this proplem yet http://www.cs.cornell.edu/courses/cs711/2005fa/papers/hind-paste01.pdf

[8]. Amer Divan, Kathryn S.McKinley, J.Eliot B.Moss. Type-Based Alias Analysis http://web.cs.ucla.edu/~palsberg/tba/papers/diwan-mckinley-moss-pldi98.pdf

[9]. Drozdov A.Y., Vladislavlev V.E. Mezhprocedurny analiz ukazatelej [Interprocedural analysis pointer]. Informacionnie tehnologii [Information technology], 2005. att No 2, pp 35-42 (in Russian).

[10]. Volkonsky V.Y. Optimiziruyuschie kompilyatory dly arhitektur s yavnim parallelizmom komand I apparatnoj podderzhkoj dvoichnoj sovmestimosti [Optimizing compilers for architectures with explicitly parallel instruction and hardware support for binary compatibility] Informacionnie tehnologii i vychislitelnie sistemy [Information technology and computer systems], 2004, no..3 (in Russian).

[11]. Galazin A.B., Stepanenkov A.M., Stupachenko E.V. Programmnaya predvaritelnaya podkachka koda dlya mikroprocessora Elbrus-3M [Spooling software code for the microprocessor Elbrus-3M]. Informacionnie tehnologii [Information technology], 2007, no. 11 (in Russian).