УДК 004.4'422
DOI: 10.25559/SITITO.019.202302.355-364

Оригинальная статья

Статический и динамический подходы к преобразованию косвенных переходов

В. В. Черноног[1*], И. Л. Дьячков[2], А. Д. Добров[1]
[1] ООО «Техкомпания Хуавэй», г. Москва, Российская Федерация
Адрес: 121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2
[2] ООО «Коулмэн Сервисиз», г. Москва, Российская Федерация
Адрес: 117218, Российская Федерация, г. Москва, ул. Кржижановского, д. 29, к. 2
[*] norrilsk@gmail.com

Аннотация

С ростом популярности модульной парадигмы программирования количество косвенных переходов в продуцируемом коде значительно возросло. Аппаратные способы уменьшить задержки, связанные с такими переходами, требуют сложной логики непосредственно на кристалле, которую зачастую невыгодно реализовывать из-за дополнительного энергопотребления процессора. Существующие в компиляторе GCC-подходы преобразования косвенных переходов позволяют существенно увеличить производительность программ, однако без сбора статистики и перекомпиляции кода все еще остается большое количество непреобразованных переходов, которые приводят к уменьшению производительности программ. В этой статьи предлагается два метода повышения производительности, связанных с оптимизацией косвенных переходов. Статический метод позволяет расширить существующие оптимизационные возможности компилятора. Динамический метод является новым подходом подстановки целевых адресов переходов во время выполнения программы. Наше исследование показывает, что эти подходы способны увеличить производительность отдельных участков программ, содержащих косвенные переходы, до 2,3 раза. Применение статического метода позволило улучшить производительность отдельных тестов из пакета CPUBench на 15 % при увеличении затраченного на компиляцию времени не более чем на 1 %.

Ключевые слова: компиляторы, оптимизации, косвенные переходы, вызовы функций, сигнатуры функций, девиртуализация

Конфликт интересов: авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Черноног В. В., Дьячков И. Л., Добров А. Д. Статический и динамический подходы к преобразованию косвенных переходов // Современные информационные технологии и ИТ-образование. 2023. Т. 19, № 2. С. 355-364. doi: https://doi.org/10.25559/SITITO.019.202302.355-364

© Черноног В. В., Дьячков И. Л., Добров А. Д., 2023

Контент доступен под лицензией Creative Commons Attribution 4.0 License.
The content is available under Creative Commons Attribution 4.0 License.

Original article

Static and Dynamic Approaches for Indirect Branch Optimization

V. V. Chernonog[a*], I. L. Diachkov[b], A. D. Dobrov[a]
[a] Huawei Technologies Co., Ltd, Moscow, Russian Federation
Address: 17 building 2 Krylatskaya St., Moscow 121614, Russian Federation
[b] Coleman Services, Moscow, Russian Federation
Address: 29 building 2 Krzhizhanovsky St., 117218, Russian Federation
[*] norrilsk@gmail.com

Abstract
With the growing popularity of the modular programming paradigm, the number of indirect branches has increased significantly in the produced code. Hardware approaches to reduce the delays associated with such branches require complex logic directly on the chip, which is often unprofitable to implement due to the additional power consumption. In the GCC compiler, existing approaches of converting indirect branches can significantly increase program performance. However, without collecting statistics and recompiling the code, there are still numerous unhandled branches that lead to a decrease in program performance. In this article, two methods are proposed to reduce the cost of indirect branches. A static method that is a continuation of existing solutions, which allows you to expand the existing optimization capabilities of the compiler. Dynamic method, an innovative approach to substitution of target transition addresses in real time. Our measurements show that these approaches are able to increase the performance of individual indirect branches up to 2.3 times. The application of the static method to the CpuBench test suite allowed improving the performance of individual tests of the package by 15 %, while the compilation time increased by no more than 1 %.

Keywords: compilers, optimizations, indirect branches, function calls, function signatures, devirtualization

Conflict of interests: The authors declares no conflict of interest.

For citation: Chernonog V.V., Diachkov I.L., Dobrov A.D. Static and Dynamic Approaches for Indirect Branch Optimization. *Modern Information Technologies and IT-Education*. 2023;19(2):355-364. doi: https://doi.org/10.25559/SITITO.019.202302.355-364

Введение

Несмотря на активно развивающиеся направления графических и тензорных ускорителей, центральное процессорное устройство остается очень распространенным вычислителем, поскольку является пригодным для выполнения широкого спектра вычислительных задач [1, 2]. Основным условием применимости центрального процессора является скорость выполнения целевой программы, которая определяется такими факторами, как выбор алгоритма, производительность аппаратуры и оптимизация написанного кода. Если задача программиста состоит в написании быстрого и качественного кода, то на эффективность его работы существенное влияние оказывает используемый набор средств разработки ПО, в частности компилятор, который призван облегчить эту задачу настолько, насколько возможно, особенно в отношении применения низкоуровневых оптимизаций, требующих специализированные знания целевой архитектуры [3, 4].

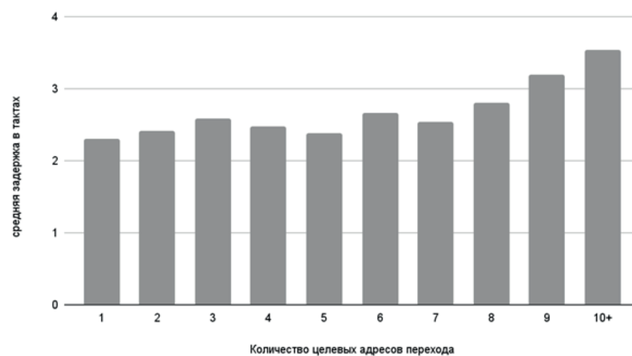
Современные высокоуровневые объектно-ориентированные языки программирования, такие как Java, C++ и C#, реализуют парадигму полиморфизма с помощью динамических таблиц вызовов функций [5, 6, 7]. В такой схеме адрес функции, которая будет вызвана, известен только во время исполнения программы. Вызовы виртуальных функций в объектно-ориентированных языках программирования являются значительным, но не единственным источником косвенных переходов в программах. Например, в языках C/C++ можно использовать указатели на функции, а в расширении GCC доступны указатели на метки внутри программы[1]. В такой парадигме программирования пользователь сам создает косвенные вызовы и переходы [8]. Исполнение нетривиальных инструкций управления зачастую замедляет работу центрального процессора, так как для эффективной обработки инструкций управления суперскалярному процессору необходимо предсказывать целевой адрес перехода. Для этого используются специальные устройства-предсказатели[2] [9]. Однако предсказание косвенного перехода отличается от предсказания условного, поскольку в первом случае конечных адресов может быть множество, и их необходимо держать в памяти предсказателя. Это оказывается достаточно затратным, так как приводит к использованию лишней площади на кристалле [10]. Современные методы разработки аппаратуры позволяют применять принцип виртуального счетчика инструкций, но такой подход все равно требует дополнительной логики непосредственно в аппаратуре [11].

Текущие подходы к решению данной задачи в основном ориентированы на разрешение вызовов виртуальных функций для объектно-ориентированного программирования [12, 13]. Отсутствие информации о целевых функциях в графе вызова (call graph) затрудняет межпроцедурные оптимизации, делает невозможным подстановку функций непосредственно в код (inlinng), что существенно ограничивает оптимизационные возможности компилятора [14, 15].

Мотивация исследования

В работе [11], посвященной использованию в аппаратуре виртуального счетчика инструкций для косвенных переходов, для приложений операционной системы Windows было подсчитано среднее количество ошибок предсказаний на тысячу инструкций. Согласно результатам, оно составляет приблизительно 6 для всех переходов и 1,9 для косвенных. Таким образом, косвенные переходы становятся причиной остановки конвейера из-за непредсказуемого изменения счетчика команд в одной трети случаев. Авторы другого исследования аппаратного улучшения предсказания косвенных переходов [16] выяснили, что доля косвенных переходов по сравнению с другими типами переходов составляет менее 10 %.

Наше собственное исследование для сервера на базе исследуемого процессора ARM показало следующую зависимость средней стоимости исполнения косвенных переходов от количества целевых адресов (рис. 1). Для измерения задержки были использованы нарезанные с помощью технологии SimPoint [17, 18] участки трасс исполнения приложений из пакетов SPEC CPU2017 [19] и CPUBench [20]. Под задержкой мы понимаем расстояние в тактах процессора между удалением инструкций (retirement) из буфера исполненных инструкций (retirement buffer).



Р и с. 1. Зависимость задержки исполнения от количества целевых адресов косвенных переходов

F i g. 1. Dependence of the execution latency on the number of target addresses of indirect branches

Источник: здесь и далее в статье все рисунки и таблицы составлены авторами.

Source: Hereinafter in the article, all tables and figures are compiled by the authors.

Еще одно предметное исследование в области косвенных переходов[3] показывает, что их преобразование с использовани-

[1] Labels as Values [Электронный ресурс] // GCC, the GNU Compiler collection online documentation, 2023. URL: https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html (дата обращения: 16.03.2023).

[2] McFarling S. Combining Branch Predictors: WRL Technical Note TN-36. Digital Western Research Laboratory, 1993 [Электронный ресурс]. URL: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf (дата обращения: 16.03.2023).

[3] Baev I., Center Q. I. Profile-based Indirect Call Promotion [Электронный ресурс] // LLVM Developers Meeting, 2015. URL: https://www.llvm.org/devmtg/2015-10/slides/Baev-IndirectCallPromotion.pdf (дата обращения: 16.03.2023).

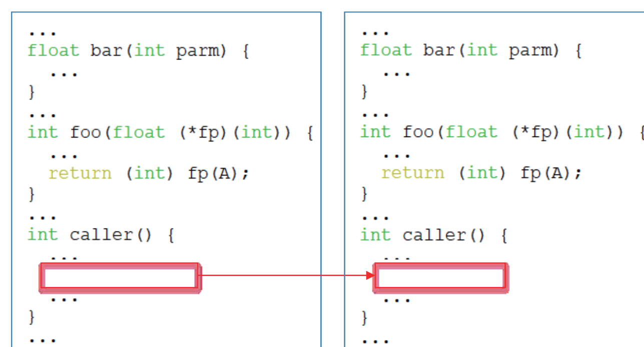
ем профиля исполнения программы (PGO) позволяет добиться среднего ускорения в 4 % тестов из пакетов SPEC CPU2000/CPU2006. Однако использование профиля является не всегда приемлемым и требует значительное количество дополнительной логики со стороны компилятора и аппаратуры. Например, запускать процесс рекомпиляции при изменении входных данных, поведения программы или иметь
несколько версий кода для своевременного использования[4] [21]. В данной статье рассматриваются два метода, которые могут быть эффективными, невзирая на изменяющийся профиль программы. Статический метод является развитием классического подхода оптимизации косвенных переходов, в том числе для функциональных языков программирования. Динамический — позволяет разрешать переходы в отсутствие статического предсказания непосредственно во время исполнения.

Целевая платформа

В качестве основной платформы для исследований был использован распространенный на рынке сервер ARM64 с архитектурой ARM V8.2-A [22], ближайшим конкурентом которого является Ampere Altra Server [23]. Исследуемая модель изготовлена с использованием 7-нанометрового техпроцесса, а ее тактовая частота составляет 2,6 ГГц. Важной характеристикой исследуемого процессора является возможность считывать 4 инструкции за такт. Размер кэша 1, 2 и 3-го уровня у данного процессора составляет 64, 512 Кб и 64 Мб на одно ядро соответственно. Стоит отметить, что кэш-линия третьего уровня исследуемого процессора составляет 128 байт, что в 2 раза больше размера кэш-линий более низкого уровня или кэш-линий процессоров конкурентов. Существенной особенностью данной модели процессора является большое количество вычислительных ядер, однако наше исследование направлено на улучшение производительности одноядерных приложений.

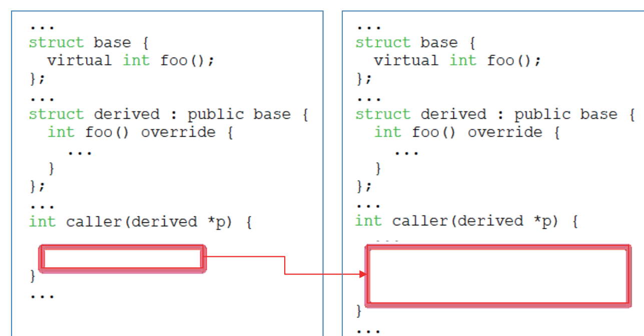
Статический подход

Естественным подходом для решения задачи удаления косвенных вызовов процедур без использования профилирования является определение адресов переходов на основе статического анализа программы. Обычно для этого применяется анализ потока данных или анализ типов, а также существуют реализации, совмещающие оба вида анализа [15], [24, 25].
В компиляторе GCC реализовано преобразование косвенных вызов на основе обоих статических методов. Локальный анализ указателей (вместе с распространением констант) позволяет оптимизировать простые случаи косвенных вызовов, когда взятие указателя на процедуру и косвенный вызов находятся в пределах одной компилируемой процедуры, первоначально или после выполнения оптимизации подстановки (inlining). Псевдокод, иллюстрирующий последний вариант оптимизации, приведен на рисунке 2. Затраты времени компиляции на такой анализ незначительны, однако в силу локальности область его применения достаточно ограничена. Расширить ее можно было бы за счет выполнения глобального анализа указателей. В GCC существует его реализация [26, 27], но по умолчанию он выключен даже на высоких уровнях оптимизации. При компиляции достаточно больших программ время работы анализа может быть велико, например, он занимает более 5 % времени при компиляции некоторых тестов из пакета SPEC CPU2017 [19]. При сборке современных версии GCC на уровне оптимизации «-O3 -flto» время работы компилятора возрастает в несколько раз, например, для GCC версии 9.0 авторы получили почти десятикратное замедление. Более того, этот анализ не реализован для указателей на процедуры в смысле функциональных языков программирования, поэтому в настоящий момент он неприменим для преобразования косвенных вызовов.



Р и с. 2. Удаление косвенного вызова после подстановки процедуры

F i g. 2. Indirect call promotion after procedure inlining



Р и с. 3. Условная девиртуализация

F i g. 3. Speculative devirtualization

Другой статический метод основан на анализе иерархии классов языка C++. На его базе выполняется девиртуализация (devirtualization) — замена вызова виртуальной функции на прямой вызов (рис. 3). Очевидно, что данный подход применим только для оптимизации программ, написанных на языке C++ или других объектно-ориентированных языках. Влияние девиртуализации на улучшение производительности можно оценить на примере оптимизации тестов из пакета SPEC CPU2017. На двух из 9 тестов, написанных с использованием языка C++, авторами работы определены следующие ускоре-

[4] Saxena R. Profile merging and code versioning for automated profile guided optimization systems: Dissertation for a Master of Science degree University of Colorado at Boulder, 2007. URL: https://www.proquest.com/openview/91ad9e73f680586023e84c4e34da06e5 (дата обращения: 16.03.2023).

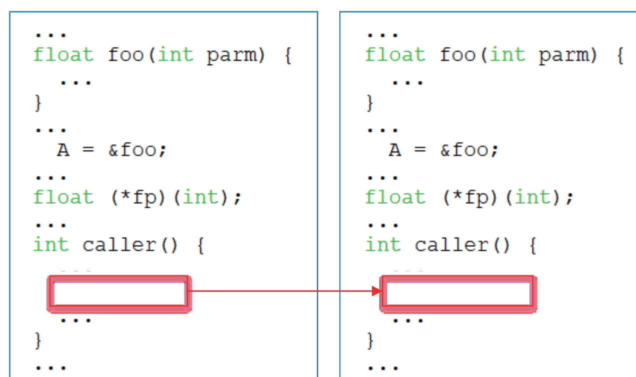
ния: 5 % на 520.omnetpp_r и 2.5 % на 523.xalancbmk_r. Таким образом, существующие методы дают заметный прирост производительности, однако не позволяют оптимизировать все существующие случаи косвенных переходов.

При компиляции целой программы есть возможность собрать информацию обо всех используемых в программе процедурах и обо всех косвенных вызовах. Сопоставляя типы аргументов и возвращаемых значений процедур, можно определить, какие из пользовательских процедур подходят для каждого косвенного вызова. На этой идее основан глобальный анализ сигнатур процедур, реализованный авторами в данной работе.

Каждой процедуре во внутреннем представлении компилятора GCC соответствует ее тип, включающий информацию о типах аргументов, возвращаемого значения и некоторую другую. Следует заметить, что процедуры, имеющие одну и ту же сигнатуру вызова с точки зрения пользователя, не обязательно получают одинаковый тип во внутреннем представлении компилятора. Это же справедливо и для пользовательских составных типов, таких как структуры и объединения (unions). Такая особенность связана с реализацией поддержки информации о типах в GCC в режиме работы LTO (link time optimization [28]), который используется для межпроцедурного анализа и оптимизации программ, состоящих из нескольких модулей. В разных модулях одни и те же типы могут иметь полное или неполное определение, что при наличии составных типов с полями-указателями на другие составные типы делает сопоставление таких типов из разных модулей без поддержки со стороны языка (такой, например, как правило одного определения в C++) или специального межпроцедурного анализа затруднительным. В режиме LTO при объединении информации из разных модулей выполняется сопоставление типов для ряда случаев (вводятся так называемые канонические типы), но не гарантируется, что будут сопоставлены все варианты одного и того же пользовательского типа.

Таким образом, для выполнения анализа сигнатур требуется предварительно определить, какие типы являются псевдонимами (как для составных, так и для типов процедур). Для этого на первой стадии необходимо обойти все процедуры программы и составить базовую таблицу псевдонимов типов, анализируя инструкции с преобразованием типа операндов, такие как инструкции вызова процедур и присваивания. На второй стадии для всех составных типов и типов процедур на основе их полей и аргументов соответственно вычисляются хеш-значения, используя которые можно найти окончательный список псевдонимов для каждой процедуры. Также во время поиска псевдонимов составляется список небезопасных для оптимизации процедур, указатели на типы которых преобразуются к указателям на типы данных или void, и наоборот.

Анализ сигнатур вызовов собирает типы всех процедур в программе и находит для каждого из них список подходящих процедур со взятым адресом с учетом собранной информации о типах-псевдонимах. Результаты анализа используются для преобразования косвенных вызовов в тех случаях, когда для подстановки найдена одна процедура. В качестве демонстрации реализованного подхода рассмотрим псевдокод, приведенный на рисунке 4. Если анализ определяет, что в программе существует только одна процедура со взятым адресом foo, имеющая такую же сигнатуру, как указатель fp, то косвенный вызов по указателю fp в процедуре caller может быть оптимизирован.



Р и с. 4. Замена косвенного вызова на основе анализа сигнатур
F i g. 4. Indirect call promotion based on signature analysis

Данный анализ и оптимизация запускаются в компиляторе GCC как межпроцедурный проход (inter-procedural optimization pass) после выполнения межпроцедурной девиртуализации (IPA devirtualization pass). В процессе своей работы оптимизация обновляет граф вызовов, что делает дальнейшие стадии межпроцедурного анализа более точными, а также позволяет компилятору выполнить подстановку (inlining) или специализацию процедур (cloning) в преобразованных местах вызова.

Динамический подход

В этом подходе авторы данной статьи хотели задействовать возможность трансформации косвенных переходов на основе информации, собранной во время исполнения программы[5] [29], но при этом обойтись без повторной компиляции программы, как в случае использования профилирования, и без системы JIT-компиляции (Just-in-Time compilation). Для решения этой задачи предлагается встроить средства трансформации кода непосредственно в исполняемую программу. В системах типа JIT заново сгенерированный код отдельного метода или функции обычно помещается во вновь выделенный участок памяти [30]. Однако такое изменение слишком глобально и потенциально разносит точку перехода и точку входа на большое адресное расстояние либо требует перемещения и повторной компиляции всех модулей. В нашем случае для решения поставленной задачи создана небольшая библиотека-транслятор (DDL — dynamic devirtualization library), которая способна трансформировать код в определенных рамках, заданных во время компиляции пользовательской программы.

Взаимодействие компилятора, целевой программы и библиотеки DDL похоже на взаимодействие с обычными динамическими библиотеками, за исключением того, что в нашем случае компилятор может сам вставлять необходимые вызовы и оберточную логику.

[5] Baev I., Center Q. I. Profile-based Indirect Call Promotion [Электронный ресурс] // LLVM Developers Meeting, 2015. URL: https://www.llvm.org/devmtg/2015-10/slides/Baev-IndirectCallPromotion.pdf (дата обращения: 16.03.2023).

Алгоритм

Шаг 0. Выбор косвенных переходов. Использование библиотеки DDL подразумевает два режима выбора переходов для оптимизации: по указанию пользователя и автоматический. В первом варианте пользователь сам решает, какие случаи нужно оптимизировать. Для этого в исходном коде программы ему необходимо обернуть косвенный вызов или переход типа goto специальными библиотечными макроопределениями DDL_GOTO или DDL_CALL. При использовании второго режима решение о применении данной оптимизации для конкретного косвенного перехода остается за компилятором. Для каждого косвенного перехода создается модуль инициализации, который запускается в начале исполнения программы и подготавливает к работе все внутренние структуры библиотеки DDL, а также добавляет права на запись для страницы памяти, содержащей косвенный переход.

Шаг 1. Трансформация косвенных переходов. Каждый косвенный переход заменяется следующей последовательностью команд:

1) k инструкций nop;
2) вызов функции сбора статистики (ddl_collect_stat);
3) вызов функции изменения кода при условии, что сбор статистики завершен (выполнено N итераций);
4) оригинальный косвенный переход;

```
INT entry_count = 0;
entry_count++;
START_REWRITE_POINT:
NOP
NOP
...
NOP // k-times reapeat
BOOL collect_stat_mode = FALSE;
INT miss_count = 0;
miss_count++;
if ( entry_count == N)
    && (miss_count > entry_count >> 4) {
  if (!collect_stat_mode){
    collect_stat_mode = TRUE;
    CALL DDL_DISABLE_OPTIMIZATON();
  } else {
    miss_count = 0;
    entry_count = 1;
    CALL DDL_REWRITE_INDIRECT();
    collect_stat_mode = FALSE;
    }
}
if (collect_stat_mode) {
  CALL DDL_UPATE_STATISTICS();
}
goto *addr; // original indirect branch
```

Р и с. 5. Псевдокод преобразованного косвенного перехода
F i g. 5. Pseudocode of the transformed indirect jump

Занятое инструкциями nop место зарезервировано для вставки оптимизированных переходов. На этом этапе вводятся параметры N и k, которые должны быть подобраны экспериментальным путем. Основываясь на опыте других исследований[6] [11], [29], можно предположить, что оптимальными будут значения $k \approx 20$, $N \approx 512$. Очевидно, что если количество часто встречающихся адресов переходов окажется велико, то выделенного для вставки прямых переходов места может не хватить (на подстановку одного прямого условного перехода требуется 3-4 инструкции: 1-2 mov, 1 cmp, 1 jmp.eq). На этот случай алгоритм предусматривает автоматическую отмену оптимизации с заменой первого nop на оригинальный косвенный переход.

Шаг 2. Запуск программы и сбор статистики. Этот и последующие этапы выполняются непосредственно во время работы программы (и не подразумевают каких-либо действий со стороны пользователя или компилятора). Каждый раз, когда в программе достигнут модифицированный косвенный переход, происходит вызов библиотечной функции обновления статистики целевых адресов для этого перехода. Статистика в библиотеке DDL накапливается отдельно для каждого перехода, который идентифицируется по уникальному номеру, присвоенному ему на этапе инициализации.

Шаг 3. Трансформация в реальном времени. Когда собранно достаточное количество статистической информации (которое задается параметром N), запускается функция модификации кода программы. Целевые адреса сортируются в порядке уменьшения количества совершенных по ним переходов. Если 95 % переходов происходит не более чем по k адресам, то выполняется замена цепочки nop инструкций на условные переходы (рис. 6). В противном случае происходит вставка исходного косвенного перехода.

```
mov     w9 #0xe20
movk    w9 #0x40, lsl #16
cmp     x28, x9
b.eq    0x400e20
movk    w9, #0xe60
cmp     x28, x9
b.eq    0x400e60
movk    w9, #0xe30
cmp     x28, x9
b.eq    0x400e30
movk    w9, #0xe40
cmp     x28, x9
b.eq    0x400e40
movk    w9, #0xe50
cmp     x28, x9
b.eq    0x400e50
movk    w9, #0xdd4
cmp     x28, x9
b.eq    0x400dd4
movk    w9, #0xe10
cmp     x28, x9
b.eq    0x400e10
nop
nop
...
```

Р и с. 6. Пример преобразованного на ходу косвенного перехода
F i g. 6. An example of translated on-the-fly indirect jump

Шаг 4. Работа оптимизированного перехода. После преобразования перехода режим сбора статистики целевых адресов выключается. Однако продолжается накопление

[6] Там же.

информации о количестве исполнений данного перехода и о числе случаев, когда в преобразованном коде отсутствует прямой переход по полученному целевому адресу и должен сработать первоначальный косвенный переход (он находится дальше по коду), т. е. пропущена оптимизация для полученного адреса. Если значение счетчика пропущенных переходов станет достаточно большим по сравнению с числом входов в данный участок кода, то цепочка прямых условных переходов будет заменена на первоначальный блок nop инструкций и опять будет запущен сбор статистики. Иными словами, совершится переход к шагу 2. При этом собранная во время предыдущего выполнения шага 2 статистика адресов учитывается с коэффициентом 0,5, что наделяет систему памятью.

Результаты

Статический подход

Результаты работы анализа и оптимизации для тестов с косвенными вызовами из пакета SPEC CPU2017 представлены в таблице 1. Благодаря реализованному анализу сигнатур компилятор в значительном числе случаев (до 100 %) определяет множество процедур, которые являются кандидатами для косвенных вызовов, и оптимизирует до 100 % этих вызовов. Согласно полученным данным, имеет смысл реализация оптимизации для случаев, когда анализ находит две или три возможные процедуры для косвенного вызова. При этом требуется выполнять условную (спекулятивную) подстановку. Также в настоящей реализации не поддерживается оптимизация методов в программах на языке C++, поэтому в соответствующих тестах относительное количество оптимизированных вызовов невелико.

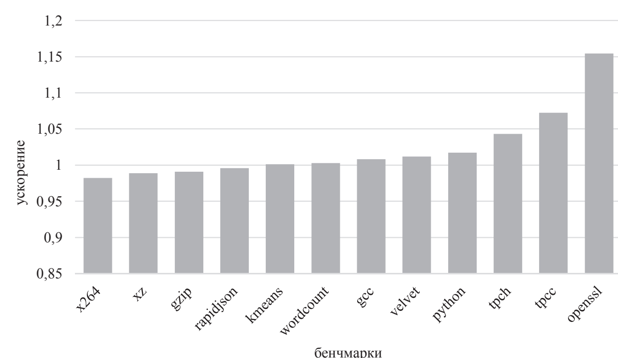
Т а б л и ц а 1. **Результаты работы анализа сигнатур и оптимизации на пакете SPEC CPU2017**

T a b l e 1. **Results of the analysis and the optimization on SPEC CPU2017**

Тест	Количество вызовов процедур								ΔTc, %
	всего	косвенных	косвенных с найденными N кандидатами для подстановки				косвенных с кандидатами, %	оптимизировано, %	
			N=1	N=2	N=3	N>3			
500.perlbench_r	301206	237	5	27	81	117	97,0	1,7	0,0
502.gcc_r	20909534	3213	384	183	172	2314	95,0	12,0	0,1
505.mcf_r	529	14	0	14	0	0	100,0	0,0	0,0
520.omnetpp_r	26571017	7091	1328	291	196	553	33,4	0,3	0,2
523.xalancbmk_r	183042435	26807	1629	688	637	1879	18,0	0,0	0,9
525.x264_r	228151	491	153	41	22	268	98,6	27,5	0,0
541.leela_r	1416	1	0	0	0	0	0,0	0,0	0,1
557.xz_r	5643	44	7	8	0	28	97,7	15,9	0,3
507.cactuBSSN_r	781791	1060	20	238	593	158	95,2	1,4	0,0
508.namd_r	3353	12	0	0	0	0	0,0	0,0	0,0
510.parest_r	16672711	4655	1170	92	159	2372	81,5	0,8	0,3
511.povray_r	232313	185	8	1	47	65	65,4	1,1	0,0
526.blender_r	179576873	11410	321	451	344	9666	94,5	2,5	1,2
538.imagick_r	74999	60	23	4	0	33	100,0	23,3	0,0
544.nab_r	1169	3	3	0	0	0	100,0	100,0	0,0

Стоит отметить, что реализованные анализ и оптимизация выполняются относительно быстро: при компиляции тестов из пакета SPEC CPU2017 на уровне оптимизации «-O3 -flto» они увеличивают время работы стадии LTO не более чем на 1,2 %. Если учесть, что в полное время сборки программ следует включить предварительную компиляцию отдельных модулей, то оптимизация оказывает еще меньшее влияние на скорость работы компилятора.

На пакете SPEC CPU2017 не было замечено улучшения в скорости работы программ. Однако авторы работы обнаружили значительное повышение производительности некоторых тестов пакета CPUBench. Эти результаты представлены на рисунке 7.

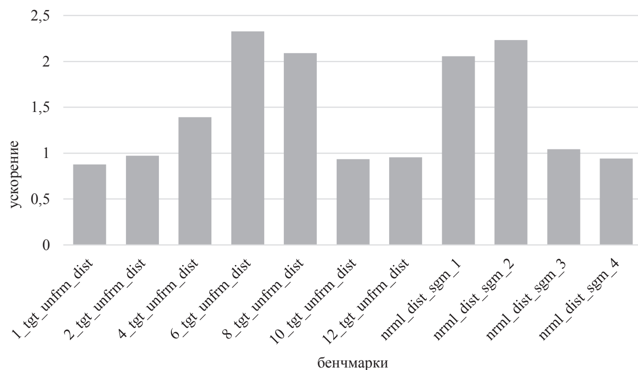


Р и с. 7. Результаты замеров производительности статического подхода для тестов пакета CPUBench

F i g. 7. Results of measuring the performance of the static approach for CPUBench test suite

Динамический подход

Для демонстрации динамического подхода был разработан набор тестов, состоящих из переходов с различными распределениями целевых адресов. Результаты замеров времени исполнения показывают (рис. 8), что данный подход дает улучшение производительности до 200 %, когда количество целевых адресов находится в диапазоне от 4 до 8. Если же целевых адресов слишком мало или слишком много, на тестах наблюдается небольшая деградация.



Р и с. 8. Результаты замеров производительности динамического подхода в зависимости от распределения количества целевых адресов

F i g. 8. Results of measuring the performance of the dynamic approach depending on the distribution of the number of target addresses

В случае компиляции сложных приложений получается несколько иная картина. Например, при оптимизации интерпретатора python (входит в тестовый пакет CPUBench) наблюдается замедление его работы на 10 %. На тестах пакета SPEC CPU2017 улучшение производительности не обнаружено. Такой результат можно объяснить значительным увеличением размера кода программ в результате работы оптимизации. Часто вставить условный переход не представляется возможным из-за ограниченного размера смещения адреса (offset). В такой ситуации компилятору приходится вставлять косвенный переход или использовать вызов-трамплин.

Выводы

В работе предложены два подхода к снижению количества косвенных переходов внутри программного кода. Статический метод является продолжением классического подхода в борьбе с косвенными вызовами, тем не менее с его помощью на тестах пакета CPUBench достигается ускорение до 15 % при увеличении времени компиляции программ менее чем на 1 %. В качестве дальнейшего развития этого метода планируется реализовать условную замену косвенных вызовов в случае нахождения нескольких процедур, подходящих для вызова.
Иные результаты получены для динамического метода: он позволяет сократить время исполнения отдельных косвенных переходов до двух раз. В дальнейшем предполагается улучшить селективность применения динамического метода, поскольку текущий вариант сильно увеличивает размер кода программы, что может приводить к замедлению исполнения реальных приложений.

References

[1] Shalf J. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*. 2020;378(2166):20190061. https://doi.org/10.1098/rsta.2019.0061

[2] Lee V.W., Kim C., Chhugani J., Deisher M., Kim D., Nguyen A.D., Satish N., Smelyanskiy M., Chennupaty S., Hammarlund P., Singhal R., Dubey P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). New York, NY, USA: Association for Computing Machinery; 2010. p. 451-460. https://doi.org/10.1145/1815961.1816021

[3] Kandemir M., Vijaykrishnan N., Irwin M.J., Ye W. Influence of compiler optimizations on system power. In Proceedings of the 37th Annual Design Automation Conference (DAC '00). New York, NY, USA: Association for Computing Machinery; 2000. p. 304-307. https://doi.org/10.1145/337292.337425

[4] Devkota S., Aschwanden P., Kunen A., Legendre M., Isaacs K.E. CcNav: Understanding Compiler Optimizations in Binary Code. *IEEE Transactions on Visualization and Computer Graphics*. 2021;27(2):667-677. https://doi.org/10.1109/TVCG.2020.3030357

[5] Calder B., Grunwald D., Zorn B. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming languages.* 1994;2(4):313-351. Available at: https://cseweb.ucsd.edu/~calder/abstracts/C++Study.html (accessed 16.03.2023).

[6] Suganuma T., et al. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*. 2000;39(1):175-193. https://doi.org/10.1147/sj.391.0175

[7] Bauer M., Rossow C. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P). Vienna, Austria: IEEE Computer Society; 2021. p. 650-666. https://doi.org/10.1109/EuroSP51992.2021.00049

[8] Shah A., Ryder B.G. Function Pointers in C – An Empirical Study. Rutgers University; 1995. https://doi.org/10.7282/T3X92FRK

[9] Mittal S. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience*. 2019;31(1):e4666. https://doi.org/10.1002/cpe.4666

[10] Driesen K., Hölzle U. Accurate indirect branch prediction. *ACM SIGARCH Computer Architecture News*. 1998;26(3):167-178. https://doi.org/10.1145/279361.279380

[11] Kim H., Joao J.A., Mutlu O., Lee C.J., Patt Y.N., Cohn R. VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. *ACM SIGARCH Computer Architecture News*. 2007;35(2):424-435. https://doi.org/10.1145/1273440.1250715

[12] Namolaru M. Devirtualization in GCC. In: Proceedings of the GCC Developers' Summit. Ottawa, Ontario Canada; 2006. p. 125-133. Available at: https://gcc.gnu.org/pub/gcc/summit/2006-GCC-Summit-Proceedings.pdf (accessed 16.03.2023).

[13] Padlewski P. Devirtualization in LLVM. In: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017). New York, NY, USA: Association for Computing Machinery; 2017. p. 42-44. https://doi.org/10.1145/3135932.3135947

[14] Li D.X., Ashok R., Hundt R. Lightweight feedback-directed cross-module optimization. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10). New York, NY, USA: Association for Computing Machinery; 2010. p. 53-61. https://doi.org/10.1145/1772954.1772964

[15] Pande H.D., Ryder B.G. Data-flow-based virtual function resolution. In: Cousot R., Schmidt D.A. (eds.) Static Analysis. SAS 1996. *Lecture Notes in Computer Science*. Vol. 1145. Berlin, Heidelberg: Springer; 1996. p. 238-254. https://doi.org/10.1007/3-540-61739-6_45

[16] Garza E., Mirbagher-Ajorpaz S., Khan T.A., Jiménez D.A. Bit-level perceptron prediction for indirect branches. In: Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19). New York, NY, USA: Association for Computing Machinery; 2019. p. 27-38. https://doi.org/10.1145/3307650.3322217

[17] Hamerly G., Perelman E., Lau J., Calder B. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction-Level Parallelism*. 2005;7(4):1-28. Available at: https://jilp.org/vol7/v7paper14.pdf (accessed 16.03.2023).

[18] Perelman E., Hamerly G., Van Biesbrouck M., Sherwood T., Calder B. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*. 2003;31(1):318-319. https://doi.org/10.1145/885651.781076

[19] Bucek J., Lange K.D., v. Kistowski J. SPEC CPU2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). New York, NY, USA: Association for Computing Machinery; 2018. p. 41-42. https://doi.org/10.1145/3185768.3185771

[20] Shi H.-k., Wang Ze-s., Zhang Shi-z., Gao X., Zhao Y-j. Performance Evaluation Benchmark of General-Purpose CPUA Survey. *ACTA ELECTONICA SINICA.* 2023;51(1):246-256. (In Chi., abstract in Eng.) https://doi.org/10.12263/DZXB.20220169

[21] Stephenson M., Rangan R., Keckler S.W. Cooperative Profile Guided Optimizations. *Computer Graphics Forum*. 2021;40(8):71-83. https://doi.org/10.1111/cgf.14382

[22] Stephens N. Armv8-a next-generation vector architecture for HPC. In: 2016 IEEE Hot Chips 28 Symposium (HCS). Cupertino, CA, USA: IEEE Computer Society; 2016. p. 1-31. https://doi.org/10.1109/HOTCHIPS.2016.7936203

[23] Fenton T., Kennedy P. The Future of ESXi on Arm. In: Running ESXi on a Raspberry Pi. Berkeley, CA: Apress; 2021. p. 355-368. https://doi.org/10.1007/978-1-4842-7465-1_14

[24] Bacon D.F., Sweeney P.F. Fast static analysis of C++ virtual function calls. In: OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96). New York, NY, USA: Association for Computing Machinery; 1996. p. 324-341. https://doi.org/10.1145/236337.236371

[25] Lu K., Hu H. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). New York, NY, USA: Association for Computing Machinery; 2019. p. 1867-1881. https://doi.org/10.1145/3319535.3354244

[26] Berlin D. Structure aliasing in GCC. In: Proceedings of the GCC Developers' Summit. Ottawa, Ontario Canada; 2005. p. 25-35. Available at: https://gcc.gnu.org/wiki/HomePage?action=AttachFile&do=get&target=2005-GCC-Summit-Proceedings.pdf (accessed 16.03.2023).

[27] Pearce D.J., Kelly P.H.J., Hankin C. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems.* 2007;30(1):4-es. https://doi.org/10.1145/1290520.1290524

[28] Glek T., Hubicka J. Optimizing real world applications with GCC Link Time Optimization. In: Proceedings of the 2010 GCC Developers' Summit. Ottawa, Ontario, Canada; 2010. p. 25-45. Available at: https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=get&target=2010-GCC-Summit-Proceedings.pdf (accessed 16.03.2023).

[29] Ishizaki K., Kawahito M., Yasue T., Komatsu H., Nakatani T. A study of devirtualization techniques for a Java Just-In-Time compiler. *ACM SIGPLAN NOTICES*. 2000;35(10):294-310. https://doi.org/10.1145/354222.353191

[30] Cravford K.M., Blackstone Jr J.H., Cox J.F. A study of JIT implementation and operating problems. *The International Journal Of Production Research*. 1988;26(9):1561-1568. https://doi.org/10.1080/00207548808947966

Поступила 16.03.2023; одобрена после рецензирования 12.05.2023; принята к публикации 25.05.2023.
Submitted 16.03.2023; approved after reviewing 12.05.2023; accepted for publication 25.05.2023.

Об авторах:

Черноног Вячеслав Викторович, сотрудник, ООО «Техкомпания Хуавэй» (121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2), **ORCID: https://orcid.org/0009-0007-8407-1082,** norrilsk@gmail.com

Дьячков Илья Леонидович, сотрудник, ООО «Коулмэн Сервисиз» (117218, Российская Федерация, г. Москва, ул. Кржижановского, д. 29, к. 2), **ORCID: https://orcid.org/0009-0002-9740-7861,** ilia.diachkov@gmail.com

Добров Андрей Дмитриевич, сотрудник, ООО «Техкомпания Хуавэй» (121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2), кандидат технических наук, **ORCID: https://orcid.org/0009-0007-5074-7873,** adobrov1954@gmail.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Viacheslav V. Chernonog, researcher, Huawei Technologies Co., Ltd (17 building 2 Krylatskaya St., Moscow 121614, Russian Federation), **ORCID: https://orcid.org/0009-0007-8407-1082,** norrilsk@gmail.com
Ilia L. Diachkov, researcher, Coleman Services (29 building 2 Krzhizhanovsky St., 117218, Russian Federation), **ORCID: https://orcid.org/0009-0002-9740-7861,** ilia.diachkov@gmail.com
Andrey D. Dobrov, researcher, Huawei Technologies Co., Ltd (17 building 2 Krylatskaya St., Moscow 121614, Russian Federation), Cand. Sci. (Eng.), **ORCID: https://orcid.org/0009-0007-5074-7873,** adobrov1954@gmail.com

All authors have read and approved the final manuscript.