УДК 004.4'416
DOI: 10.25559/SITITO.17.202102.369-378

Оригинальная статья

Сбор профильной информации с помощью трасс исполнения приложения для статической оптимизирующей бинарной трансляции

С. А. Лисицын[1,2]
[1] Российский исследовательский институт Huawei, г. Москва, Российская Федерация
121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, корп. 2
[2] ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация
141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9
lisitsyn.sergey@huawei.com

Аннотация
Статическая бинарная оптимизация – один из способов ускорить исполняемый файл без исходного кода. Данная технология применяется в бинарном оптимизаторе BOLT (Binary Optimization and Layout Tool), требующий для своей работы профиль с информацией о взятых и информацией о не предсказанных переходах, которую на x86 архитектуре можно получить из аппаратной очереди LBR (Last Branch Record). Эта информация необходима ему для перекомпоновки кода, благодаря которой уменьшается количество промахов по инструкционному кэшу и буферу ассоциативной трансляции инструкций. В результате проделанной оптимизации удалось ускорить работу серверных приложений на 8.0% работающих на x86 архитектурах. На применение данной оптимизации на ARM архитектуре есть ограничения, связанные с частым отсутствием возможности получить профильную информацию с помощью аппаратных средств. В данной статье описываются разработанные методы и инструменты, позволяющие получить профильную информацию с помощью трассы исполнения приложения. Процесс сбора трассы реализован с помощью динамической бинарной инструментации. В статье описан алгоритм восстановления профильной информации с применением модели предсказателя перехода. Также описываются реализованные изменения BOLT для расширенной поддержки ARM архитектуры. В результате работы удалось достичь целевых показателей прироста производительности на синтетических тестах и тестовых наборах.

Ключевые слова: бинарная оптимизация, симуляция, моделирование, динамическая бинарная инструментация, профилирование, RISC, BOLT, ARM

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Лисицын, С. А. Сбор профильной информации с помощью трасс исполнения приложения для статической оптимизирующей бинарной трансляции / С. А. Лисицын. – DOI 10.25559/SITITO.17.202102.369-378 // Современные информационные технологии и ИТ-образование. – 2021. – Т. 17, № 2. – С. 369-378.

Original article

Collecting Profile Information Using Application Execution Traces for Static Optimizing Binary Translation

S. A. Lisitsyn[a,b]
[a] Huawei Russian Research Institute, Moscow, Russian Federation
17 Krylatskaya St., building 2, Moscow 121614, Russian Federation
[b] Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation
9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation
lisitsyn.sergey@huawei.com

Abstract
Static binary optimization is one of the ways to speed up an executable file without source code. This technology is used in the BOLT (Binary Optimization and Layout Tool), which requires a profile with information about taken and information about not predicted transitions for its work, which can be obtained from the LBR (Last Branch Record) hardware queue on the x86 architecture. This information is necessary for the BOLT in order to relocate the code, that reduces the number of misses on the instruction cache and the buffer of associative translation of instructions. As a result of the optimization, it was possible to speed up the work of server applications by 8.0% running on x86 architectures. There are limitations on the use of this optimization on the ARM architecture due to the frequent lack of the ability to obtain profile information using hardware. This article describes the developed methods and tools that allow us to obtain profile information using the application execution route. The process of collecting the trace is implemented using dynamic binary instrumentation. The article describes an algorithm for restoring profile information using a branch predictor model. The implemented BOLT changes for extended ARM architecture support are also described. As a result of the work, it was possible to achieve performance growth targets on synthetic tests and benchmarks.

Keywords: Binary optimization, simulation, simulation, dynamic binary instrumentation, profiling, RISC, BOLT, ARM

The author declares no conflict of interest.

For citation: Lisitsyn S.A. Collecting Profile Information Using Application Execution Traces for Static Optimizing Binary Translation. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2021; 17(2):369-378. DOI: https://doi.org/10.25559/SITITO.17.202102.369-378

Введение

В настоящий момент задача улучшения производительности вычислительных машин приобретает всё большее значение. В 2021 году две трети населения планеты ежедневно используют мобильные вычислительные устройства, а их количество перевалило за 4 миллиарда[1]. Уменьшение времени работы на конкретных задачах, или увеличение соотношения производительность/энергопотребление приносит большой выигрыш в масштабе количества устройств [1].

Повышения производительности приложений можно добиться двумя путями: улучшением аппаратного и программного обеспечения. Под последним понимается не только качество написанного кода и выбранных алгоритмов, но и процесс трансляции высокоуровневого языка в бинарное представление вместе с процессом запуска бинарного файла. Для повышения производительности в трансляцию добавляются оптимизирующие стадии, такие как оптимизации компилятора, линкера и бинарные оптимизации, о которых и пойдёт речь в данной статье.

Для повышения качества оптимизаций на вход трансляторам может поступать дополнительная информация: профиль исполнения, вероятные входные данные, дополнительная данные о микроархитектуре целевого процессора и др.

В первой части статьи описываются компоненты и оптимизации бинарного транслятора BOLT, разработанного сотрудниками Facebook [4], [8]. Далее приводится анализ данного инструмента и изучены проблемы его работы на ARM архитектуре. Была решена задача сбора профиля для процессоров этой архитектуры с помощью динамической бинарной инструментации. В работе приводится анализ показателей микроархитектуры в зависимости от размера часто исполняемого кода оптимизируемого приложения. В результате проделанных исследований удалось получить 10% прирост показателей на наборе тестов производительности.

Бинарная оптимизация

Практически всегда приложение проходит процесс оптимизации во время компиляции, так как данные проходы включают в системах сборки по умолчанию. Во время трансляции программы с языка высокого уровня в бинарный код реальной или виртуальной машины получаемое промежуточное представление подвергается упорядоченному набору оптимизаций. Довольно часто для компилируемых языков применяются дополнительные оптимизации времени компоновки кода (LTO – Link Time Optimizations), так как на этом этапе становится доступно больше информации о всех полученных объектных файлах. После компоновки кода приложение готово к использованию [6].

Во время упомянутых оптимизаций использовалась информация о приложении, полученная из исходного кода. Для повышения производительности можно собрать профиль исполнения приложения, что позволит лучше подобрать эвристики для используемых оптимизационных проходов, а также применить дополнительные проходы в компиляторе и линкере.

Если доступа к исходному коду нет, но есть бинарный файл приложения и его профиль исполнения, появляется возможность провести бинарную оптимизацию. Добавив информацию о спецификации процессора, на котором будет исполняться приложение, можно провести ряд микроархитектурных оптимизаций. Необходимо решить задачу записи профиля, так как необходимо осуществить сбор с бинарного файла, собранного для исполнения, а не профилирования.

Профилирование

Один из способов собрать профиль – провести инструментацию бинарного файла. Во время запуска будет произведена запись по счетчикам, расставленных во время компиляции. Но для данного подхода необходим исходный код. Основная проблема заключается в необходимости адаптации инфраструктуры сборки проекта для профилирования, кроме того, используемые сторонние библиотеки не предоставляются в инструментированном виде, что осложняет получения полной картины исполнения [2].

Вместо инструментации кода во время компиляции разработаны среды, позволяющие произвести инструментацию во время исполнения приложения. Таким образом может быть получена вся необходимая информация об исполнении, при разрешении сложностей извлечения собранных данных. Однако, стоимость разработки динамической инструментации выше разработки аналогичного процесса в момент компиляции. Также появляются накладные расходы во время записи трасс исполнения [3].

При наличии архитектурной поддержки есть возможность собрать информацию исполнения с меньшими накладными расходами. В x86 архитектуре такая поддержка есть, что позволило сотрудникам Facebook использовать такой подход в своём бинарном оптимизаторе – BOLT (Binary Optimization and Layout Tool) [4].

Binary Optimization and Layout Tool

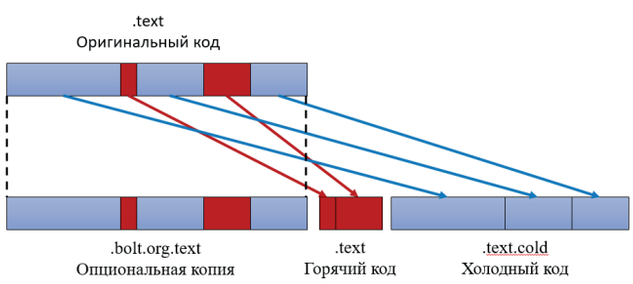
Основными целевыми приложениями для данного оптимизатора являются современные программы для центров обработки данных. Из-за размера их кода, повышение его локализации стало важным инструментом повышения производительности [5]. Основной оптимизацией BOLT является перекомпоновка кода на основе полученного профиля, использование которого на бинарном уровне точнее, чем на этапе компиляции. Создатели данного бинарного оптимизатора достигли прирост производительности в 8% на x86 приложениях, которые были скомпилированы с оптимизациями времени связывания и с использованием профиля [4].

Данный бинарный оптимизатор написан с использованием фреймворка написания компиляторов – LLVM, что делает BOLT потенциально портируемым на множество других архитектур. Базовая поддержка ARM архитектуры (Aarch64) на данный момент уже добавлена.

BOLT эффективно применять на приложениях, размер которых значительно больше размера L1I (инструкционного кэша

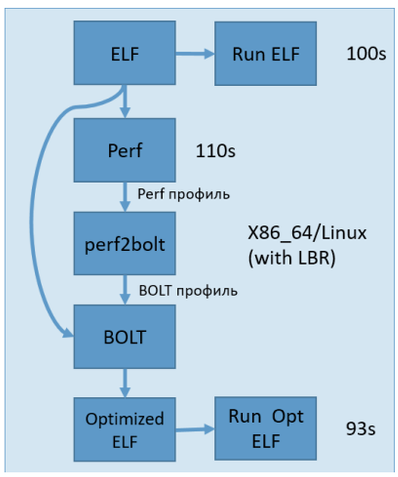
[1] *O'Dea S.* Global smartphone sales to end users 2007-2021 [Электронный ресурс] // Statista.com, 2021. URL: https://www.statista.com/statistics/755388/global-smartphone-unit-sales-by-region (дата обращения: 23.04.2021).

1 уровня) и размера отображаемого кода iTLB (буфер ассоциативной трансляции инструкций) – десятки мегабайт и более. За счет новой компоновки кода, основанной на профиле, исполняемый часто код локализуется в новой секции. Частоту исполнения кода называют его «температурой», и в данном примере созданная секция является «горячей». Остальной же код с меньшей частатой исполнения будет называться «холодным». Теперь при обращении в горячий участок кода, в кэш-линию, помимо исполняемого в данный момент кода, будут попадать более горячие соседние участки из новой секции, и температура кода инструкционного кэша будет выше. Это приводит к меньшим промахам при обращении в L1I и iTLB, а значит повышению производительности.



Р и с. 1. Алгоритм работы BOLT
F i g. 1. BOLT algorithm

Для нахождения горячих и холодных участков кода используется профиль, собранный на основе выборок. BOLT использует данные, полученные с помощью приложения perf, собирающее статистические данные исполнения такие как: значения счетчика команд и последние взятые переходы с информацией от предсказателя переходов. Последний параметр предоставляется perf благодаря аппаратной поддержке в архитектуре x86. Поддержка осуществляется с помощью LBR (Last Branch Record), записывающей информацию о последних 8-32 (в зависимости от модели процессора) переходах.

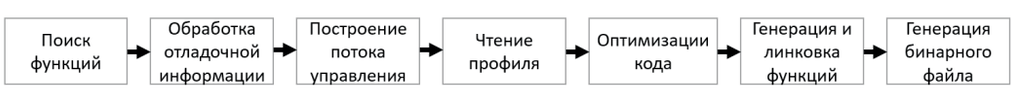


Р и с. 2. Схема оптимизации бинарного файла на x86 архитектуре
F i g. 2. Binary file optimization scheme on x86 architecture

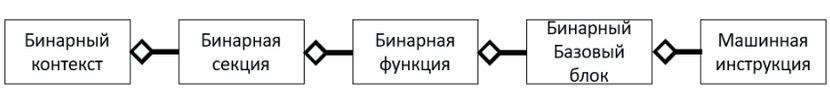
После происходит конвертация профиля из формата perf в формат BOLT при помощи приложения perf2bolt. Будет сгенерирован список записей, состоящих из адресов начала и конца перехода, количества не предсказанных и количества взятых переходов.

На основании полученного профиля начинается работа бинарного оптимизатора. На рисунке 3 предоставлена схема работы BOLT.

Уровни абстракций, на которых работает BOLT, выстраиваются в следующей последовательности:



Р и с. 3. Последовательность работы BOLT
F i g. 3. BOLT Sequence



Р и с. 4. Уровни абстракции BOLT
F i g. 4. BOLT Abstraction Layers

Для первых 4 абстракций создатели BOLT написали в программном коде свои собственные классы. Машинные инструкции были использованы из стандартных классов LLVM (MCInst). Под бинарным базовым блоком понимается линейный участок кода, исполнение которого всегда идёт с его первой интсрукции и заканчивается последней.

После работы бинарного оптимизатора получается новый бинарный файл, в котором вместо одной секции .text с исходным кодом появляется две секции .text (с горячим кодом) и .text.cold (с холодным кодом).

BOLT поддерживает возможность сохранения оригинальной секции. В таком случае он переименует её в .bolt.org.text. Это необходимо для тех случаев, когда перенос некоторых бинарных функций невозможен, и необходимо переиспользовать их оригинальный код.

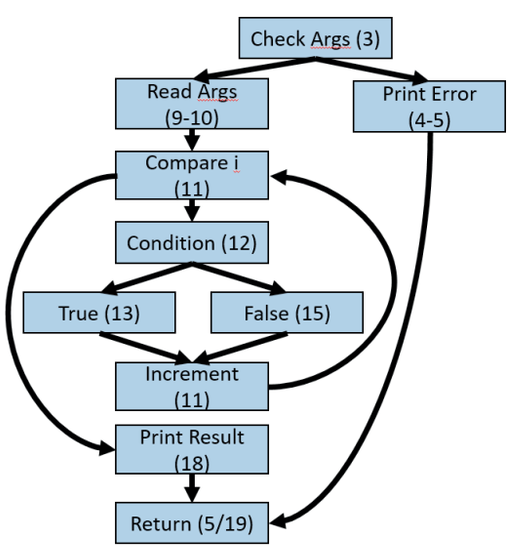
2.3 Пример оптимизации

Рассмотрим пример оптимизации следующей функции main:

```
1.  int main(int argc, char** argv)
2.  {
3.    if (argc != 3) {
4.      printf("Need two arguments\n");
5.      return 1;
6.    }
7.    double res = 0.0;
8.    uint32_t TRUE = 0, FALSE = 0;
9.    uint32_t arg = atoi(argv[1]);
10.   uint32_t itNumber = atoi(argv[2]);
11.   for (uint32_t i = 1; i < itNumber; i++) {
12.     if (condition(i, arg)) {
13.       res += trueFunc(i, &TRUE);
14.     } else {
15.       res += falseFunc(i, &FALSE);
16.     }
17.   }
18.   printf("[T:%d | F:%d] %lf\n", TRUE, FALSE, res);
19.   return 0;
20. }
```

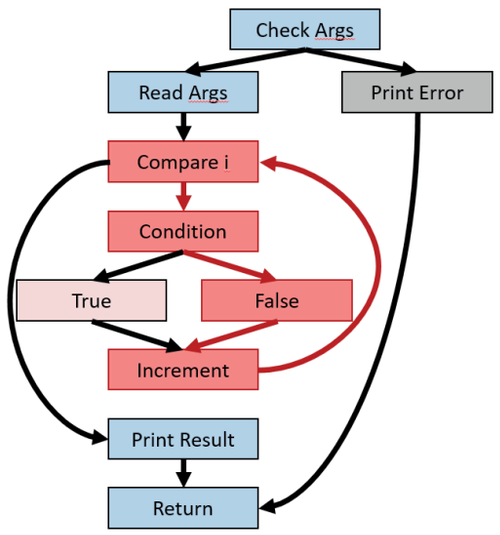
Р и с. 5. Пример оптимизируемого кода
F i g. 5. Optimized code example

BOLT найдёт символ main в таблице символов, перейдёт по указанному адресу в бинарном файле, дизассемблирует тело функции и выстроит поток управления данной функции (в скобках обозначены соответствующие строки кода):



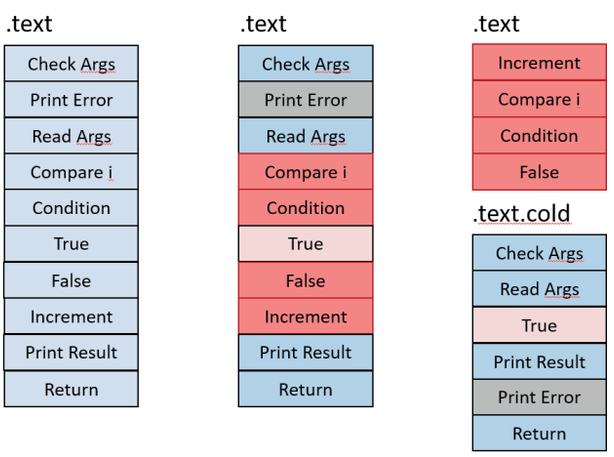
Р и с. 6. Граф потока управления
F i g. 6. Control Flow Graph

Каждому узлу в графе потока управления соответсвует один бинарный базовый блок, которым будет оперировать BOLT. При компиляции без профильной информации компилятор размещает код последовательно в одной .text секции, как указано на рисунке 8 слева. После запуска программы с профилировщиком на конкретных аргументах получим следующий профиль:



Р и с. 7. Граф потока управления с профильной информацией
F i g. 7. Control flow graph with profile information

С учетом полученной информации от профилировщика известна частота исполнения базовых блоков, размещенных в бинарном файле, как показано на рисунке 8 в центре. Получается, что самый часто исполняемый, то есть горячий код разбит холодным бинарным базовым блоком True, который будет попадать в кэш-линию при копировании кода в L1I и понимжать его среднюю температуру.



Р и с. 8. Расположение базовых блоков в секции
(без профильной информации – с профильной информацией – после перекомпоновки кода)
F i g. 8. Location of base blocks in a section
(without profile information - with profile information - after rebuilding the code)

Оптимизация BOLT перекомпанует бинарные базовые блоки, как показано на Рис. 9 справа. Самые горячие блоки попадут в горячую секцию .text, остальные будут скопированы в секцию .text.cold. В итоге, за счёт повышения средней температуры L1I и/или iTLB полученный бинарный файл будет работать быстрее.

Как было сказано раньше, основной архитектурой для оптимизации была заявлена x86. Также заявлена поддержка 64-битной ARM архитектуры (Aarch64), но на деле возникает множество проблем при попытке использования оптимизатора на ней.

Сбор профиля на ARM

Основная проблема работы на ARM архитектуре связана со сложностью получения профиля по переходам. ARM предоставляет функциональность для сбора профильной информации, но она опциональная и не присутствуют во всех процессорах. Отсутствие доступного аналога LBR на большинстве процессорах приводит к невозможности сбора данной информации во время исполнения [9-18].

Альтернативный способ получения профиля – сбор значений программного счетчика. С их помощью получается информация о частоте исполнения кода в процессе исполнения. Однако, такой подход не позволяет использовать весь список оптимизаций, включенных в BOLT. Этот способ применяется аналогично с использованием инструмента perf.

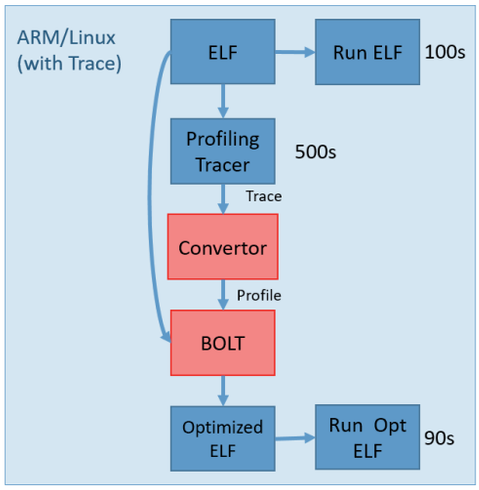
Столкнувшись с проблемой неполноценной оптимизации на ARM архитектуре, было принято решение разработать метод получения полноценного профиля с использованием динамической бинарной инструментации.

Динамическая бинарная инструментация

Для анализа исполняемого файла без исходного кода применяется динамическая бинарная инструментация (DBI, Dynamic Binary Instrumentation). Во время работы приложения производится модификация программы с целью её анализа. Происходит вставка дополнительного кода с помощью инструментирующих процедур [3]. Это позволяет производить анализ во время возникновения в программе необходимых событий, таких как исключения, взятие переходов, создание процессов и т.д. [2].

На основе данного подхода были реализованы DBI фреймворки, такие как Valgring, PIN, DynamoRIO и DynInst. На их основе реализуются динамические бинарные анализаторы (DBA, Dynamic Binary Analysis), один из которых и был использован для сбора профильной информации для ARM архитектуры.

С помощью DBI фреймворка производится запись трассы исполнения приложения. Записывается последовательность инструкций, поступающих на процессор. После исполнения последовательность анализируется конвертором для генерации профиля исполнения, который записывается в формате BOLT. Такое решение приводит к замедлению работы оптимизируемого приложения в несколько раз, но создаёт трассу, которую можно анализировать без привязки к конкретной микроархитектуре.

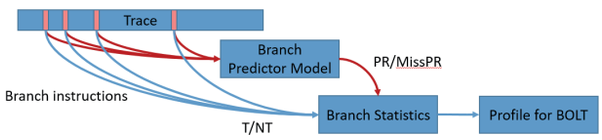


Р и с. 9. Схема оптимизации бинарного файла на ARM архитектуре

F i g. 9. Scheme for optimizing a binary file on ARM architecture

Генерация профиля по трассе

Для получения информации о переходах для профиля конвертер проходит по всей трассе и находит инструкции меняющие поток управления: B, BR, BL, BLR, B.cond, TBZ, TBNZ, CBZ, CBNZ, RET. После чего информация поступает в модель предсказателя переходов (Branch Predictor Model). Данная модель выбирается в зависимости от конкретной микроархитектуры процессора, для которого оптимизируется приложение. В итоге, собирается информация для получения количества не предсказанных и количества взятых переходов.



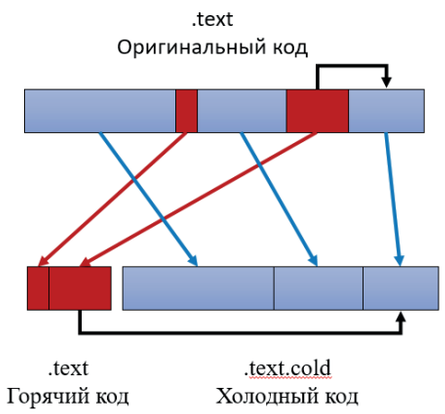
Р и с. 10. Схема генерации профиля из трассы исполнения приложения

F i g. 10. Scheme of profile generation from application execution trace

Адаптация оптимизатора под ARM архитектуру

Помимо отсутствия LBR, ARM архитектура накладывает дополнительные ограничения на перемещение горячих базовых блоков. Перекомпоновка кода проще реализуется на CISC архитектуре за счёт возможности добавления длинных прыжков одной инструкцией. На RISC архитектурах количество битов смещения в инструкциях переходов меньше, что приводит к ограничению диапазона возможного перемещения. Также инструкция загрузки в регистр адреса, зависящего от счетчика инструкций, на ARM архитектуре ограничена 20 битами, выделенными на смещение, и накладывает возможный диапазон перемещения ± 1 мегабайт. Все инструкции, зависящие от адреса инструкции (переходы, загрузка адреса, зависящего от счетчика инструкций, загрузка по регистру/смещению) после

оптимизации необходимо проверить на возможность записи нового смещения в отведенные для этого биты [7].
Таким образом, при перемещении горячего кода, ссылающегося на холодный участок, происходит переполнение значения смещения и оптимизация становится не валидной (Рис. 11).



Р и с. 11. Пример перемещения горячего кода
F i g. 11. Hot Code Move Example

Для решения данной проблемы необходимо сгенерировать трамплин – дополнительный код позволяющий сгенерировать произвольный адрес и произвести переход, либо загрузку по данному значению. Подобное решение будет увеличивать размер кода, что приведёт к понижению производительности, но диапазон перемещаемого кода будет.
Помимо проблем с недостаточным количеством битов в интсрукциях на ARM архитектуре была найдена проблема с таблицей переходов (jump table). При перемещении кода конкретного случая конструкции switch-case, необходимо модифицировать запись в таблице, используемую в косвенном переходе. Оптимизатор узнает адрес записи из релокационной информации, которая для некоторых случаев кода ARM архитектуры не будет корректной. Это происходит по причине не соответствия адреса, от которого вычисляется целевой адрес для прыжка при исполнении и при генерации релокационной информации. Данная проблема обходится переиспользованием оригинального участка кода с данным косвенным переходом. В этом случае необходимо сохранить копию оригинальной секции .bolt.org.text.
Для оптимизации с помощью BOLT необходима релокационная информация, так как его алгоритм производит перекомпоновку всей секции .text, в которой может содержаться как код, так и данные. Это одна из особенностей фон-Неймановских архитектур, когда код и данные располагаются в одном адресном пространстве [19-22].
На рисунке 12 представлен пример, когда одни и те же 4-байтовые данные могут обозначать либо ARM инструкцию условного перехода, либо 4 символа. При перекомпоновки необходимо изменить значение смещения инструкции b.ls, во втором случае 4-байтовые данные должны остаться не изменными. Поэтому без дополнительной информации о расположении данных оптимизатор не сможет решить, как обработать секцию корректно.

Type	CODE	DATA
HEX	0x29542854	0x29542854
Decode	b.ls 0x50A84	“(T)T”
New decode	b.ls 0x4004	“(T)T”
New HEX	0x29000254	0x29542854

Р и с. 12. Пример модификации 4-байтовых данных при перемещении кода
F i g. 12. Example of modifying 4-byte data when moving code

Помимо релокационной информации BOLT требует наличие символов в бинарном файле, так как он оперирует с бинарными функциями. Также формат профиля в BOLT использует имена символов для обозначения местоположения инструкций переходов, и куда они переводят исполение [23-31].

Результаты тестирования

Реализованный метод генерации профиля и модифицированная версия BOLT для ARM архитектуры были протестированы на синтетических тестах. Искусственно были созданы примеры, когда исполняемый код перемешан с неисполняемым холодным кодом, что приводит к неэффективному использованию инструкционного кэша. Для генерации тестов применялись рекурсивный вызов шаблонных функций в языке программирования C++. Исполняемый код равномерно распределяется по секции, перемежаясь с холодными функциями. Полученные статистические данные показывают понижение практически до нуля количество промахов по iTLB, увеличение IPC (instruction per clock) на 41% и уменьшение времени работы на 29%. Для тестирования была выбрана микроархитектура ARM Cortex A76.
Для проверки оптимизации на реальном приложении был выбран набор тестов производительности GeekBench. Среди всего набора был выбран тест с наибольшим числом iTLB и L1I промахов – Clang. При компиляции набора тестов были использован флаг “-O2”, что соответвует оптимизациям при стандартной сборке приложений, публикуемых в открытый доступ.
По результатам тестов был получен прирост в показателях теста на 10%, уменьшение iTLB и L1I промахов на 32% и 38% соответственно, что является показателем увеличения средней температуры кода. При этом остальные тесты из набора GeekBench либо не улучшили производительность, либо ухудшили её.
Регрессии возникают из-за стандартной проблемы оптимизации на основе профильной информации. Когда исполнение приложения отклоняется от использованного для оптимизации, начинается использование не оптимизированного кода. В случае с BOLT происходит уход исполнения в холодную секцию кода, и средняя температура L1I понижается, приводя к понижению производительности.
Для избежания данной проблемы был реализован режим обработки нескольких трасс одновременно, что убрало регрессии на остальных тестах, а в некоторых случаях и увеличило производительность. Данный режим является аналогом приложения для объединения нескольких профилей, предоставляемого вместе с BOLT.

```
erf stat -e L1-icache-load-misses,L1-icache-loads,iTLB-load-misses,iTLB-loads,cycles,instructions
real 4.544466
user 4.532000
sys 0.008000

 Performance counter stats for 'time ./main_ARM64_1k_10_clang.bolt.hack 0 1 1000000':

         38828731      L1-icache-load-misses                                       (16.74%)
      10344492618      L1-icache-loads                                             (16.77%)
              6        iTLB-load-misses          #    0.00% of all iTLB cache hits   (16.74%)
       8292220399      iTLB-loads                                                  (16.69%)
      11658075504      cycles                                                      (16.71%)
      32435371924      instructions              #    2.78  insns per cycle         (16.70%)

      4.558532811 seconds time elapsed

erf stat -e L1-icache-load-misses,L1-icache-loads,iTLB-load-misses,iTLB-loads,cycles,instructions
real 6.375960
user 6.348000
sys 0.008000

 Performance counter stats for 'time ./main_ARM64_1k_10_clang 0 1 1000000':

        567516166      L1-icache-load-misses                                       (16.73%)
      10984131626      L1-icache-loads                                             (16.72%)
        124961644      iTLB-load-misses          #    1.38% of all iTLB cache hits   (16.66%)
       9070438457      iTLB-loads                                                  (16.65%)
      16353327498      cycles                                                      (16.69%)
      32170724391      instructions              #    1.97  insns per cycle         (16.78%)

      6.389770312 seconds time elapsed
```

Р и с. 13. Сравнение оригинального (снизу) и оптимизированного (сверху) бинарных файлов синтетического теста

F i g. 13. Comparison of the original (bottom) and optimized (top) synthetic test binaries

```
Single-Core
  Clang                          703          5.48 Klines/sec

Benchmark Summary
  Single-Core Score              457
    Integer Score                703


 Performance counter stats for './gkb5 --workload 208 --single-core --no-upload':

      16975240547      instructions              #    1.32  insns per cycle          (20.59%)
      12847912630      cycles                                                      (20.59%)
         58933576      L1-icache-load-misses                                       (20.74%)
       6167593264      L1-icache-loads                                             (21.05%)
         15616258      iTLB-load-misses          #    0.29% of all iTLB cache hits   (20.46%)
       5398816554      iTLB-loads                                                  (20.14%)

      5.387183853 seconds time elapsed
```

Р и с. 14. Результаты исполнения оригинального Clang теста

F i g. 14. Results of the original Clang test execution

```
Single-Core
  Clang                          774          6.03 Klines/sec

Benchmark Summary
  Single-Core Score              503
    Integer Score                774


 Performance counter stats for './gkb5.bolt --workload 208 --single-core --no-upload':

      17091629207      instructions              #    1.34  insns per cycle          (21.09%)
      12789839024      cycles                                                      (20.91%)
         40128240      L1-icache-load-misses                                       (21.04%)
       6251372347      L1-icache-loads                                             (20.59%)
          9692777      iTLB-load-misses          #    0.19% of all iTLB cache hits   (20.40%)
       5210630729      iTLB-loads                                                  (20.50%)

      5.228721353 seconds time elapsed
```

Р и с. 15. Результаты исполнения оптимизированного Clang теста

F i g. 15. Results of executing an optimized Clang test

Заключение

Исследование показало, что существующее решение в области бинарной оптимизации не позволяет полноценно оптимизировать приложения под ARM архитектуру. В результате проделанной работы удалось реализовать альтернативный способ сбора информации с использованием динамической бинарной инструментации. Исправлены возникающие для данной архитектуры проблемы, связанные с особенностью кодировки команд. Подход протестирован на синтетических тестах и наборе тестов производительности GeekBench и показал ожидаемые результаты.

В качестве дальнейшего шага исследования рассматривается реализация многопрофильной оптимизации с нормировкой температуры базовых блоков по профилям и их локализацией. Также не рассмотрены используемые платформозависимые оптимизации, которые возможно добавить для ARM архитектуры.

References

[1] Lebras Y., Charif-Rubial A.S., Jalby W. Combining static and dynamic analysis to guide PGO for HPC applications: a case study on real-world applications. *2019 International Conference on High Performance Computing & Simulation (HPCS).* IEEE Press, Dublin, Ireland; 2019. p. 513-520. (In Eng.) DOI: https://doi.org/10.1109/HPCS48598.2019.9188161

[2] Nethercote N., Seward J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN Notices*. 2007; 42(6):89-100. (In Eng.) DOI: https://doi.org/10.1145/1273442.1250746

[3] Li J., Ma X., Zhu C. Dynamic Binary Translation and Optimization. *Journal of Computer Research & Development*. 2007. 44(1):161. (In Eng.) DOI: https://doi.org/10.1360/crad20070123

[4] Panchenko M., Auler R., Nell B., Ottoni G. BOLT: A Practical Binary Optimizer for Data Centers and beyond. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Press, Washington, DC, USA; 2019. p. 2-14. (In Eng.) DOI: https://doi.org/10.1109/CGO.2019.8661201

[5] Ottoni G., Maher B. Optimizing function placement for large-scale data-center applications. *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Press, Austin, TX, USA; 2017. p. 233-244. (In Eng.) DOI: https://doi.org/10.1109/CGO.2017.7863743

[6] Newell A., Pupyrev S. Improved Basic Block Reordering. *IEEE Transactions on Computers.* 2020; 69(12):1784-1794. (In Eng.) DOI: https://doi.org/10.1109/TC.2020.2982888

[7] Blem E., Menon J., Sankaralingam K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).* IEEE Press, Shenzhen, China; 2013. p. 1-12. (In Eng.) DOI: https://doi.org/10.1109/HPCA.2013.6522302

[8] Panchenko M., Auler R., Sakka L., Ottoni G. Lightning BOLT: powerful, fast, and scalable binary optimization. *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC 2021).* Association for Computing Machinery, New York, NY, USA; 2021. p. 119-130. (In Eng.) DOI: https://doi.org/10.1145/3446804.3446843

[9] Valiante E., Hernandez M., Barzegar A., Katzgraber H.G. Computational overhead of locality reduction in binary optimization problems. *Computer Physics Communications.* 2021; 269:108102. (In Eng.) DOI: https://doi.org/10.1016/j.cpc.2021.108102

[10] Hong D.-Y., Wu J.-J., Liu Y.-P., Fu S.-Y., Hsu W.-C. Processor-Tracing Guided Region Formation in Dynamic Binary Translation. *ACM Transactions on Architecture and Code Optimization.* 2018; 15(4):52. (In Eng.) DOI: https://doi.org/10.1145/3281664

[11] Khan T.A., Sriraman A., Devietti J., Pokam G., Litz H., Kasikci B. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE Press, Athens, Greece; 2020. p. 146-159. (In Eng.) DOI: https://doi.org/10.1109/MICRO50266.2020.00024

[12] Lavaee R., Criswell J., Ding C. Codestitcher: inter-procedural basic block layout optimization. *Proceedings of the 28th International Conference on Compiler Construction (CC 2019).* Association for Computing Machinery, New York, NY, USA; 2019. p. 65-75. (In Eng.) DOI: https://doi.org/10.1145/3302516.3307358

[13] Ottoni G., Liu B. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Press, Seoul, Korea (South); 2021. p. 340-350. (In Eng.) DOI: https://doi.org/10.1109/CGO51591.2021.9370314

[14] Sari A., Butun I. A Highly Scalable Instruction Scheduler Design based on CPU Stall Elimination. *2021 Zooming Innovation in Consumer Technologies Conference (ZINC).* IEEE Press, Novi Sad, Serbia; 2021. p. 105-110. (In Eng.) DOI: https://doi.org/10.1109/ZINC52049.2021.9499298

[15] Neves N., Tomás P., Roma N. Compiler-Assisted Data Streaming for Regular Code Structures. *IEEE Transactions on Computers.* 2021; 70(3):483-494. (In Eng.) DOI: https://doi.org/10.1109/TC.2020.2990302

[16] Ying V.A., Jeffrey M.C., Sanchez D. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* IEEE Press, Valencia, Spain; 2020. p. 159-172. (In Eng.) DOI: https://doi.org/10.1109/ISCA45697.2020.00024

[17] Gadioli D., et al. SOCRATES – A seamless online compiler and system runtime autotuning framework for energy-aware applications. *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE Press, Dresden, Germany; 2018. p. 1143-1146. (In Eng.) DOI: https://doi.org/10.23919/DATE.2018.8342183

[18] Lin Y. Control-Flow Integrity Enforcement with Dynamic Code Optimization. *Novel Techniques in Recovering, Embedding, and Enforcing Policies for Control-Flow Integrity. Information Security and Cryptography.* Springer, Cham; 2021. p. 77-94. (In Eng.) DOI: https://doi.org/10.1007/978-3-030-73141-0_5

[19] Ovasapyan T.D., Knyazev P.V., Moskvin D.A. Application of Taint Analysis to Study the Safety of Software of the Internet of Things Devices Based on the ARM Architecture. *Automatic Control and Computer Sciences.* 2020; 54(8):834-840. (In Eng.) DOI: https://doi.org/10.3103/S0146411620080246

[20] Li G., Liu L., Feng X. Accelerating GPU Computing at Runtime with Binary Optimization. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Press, Washington, DC, USA; 2019. p. 276-277. (In Eng.) DOI: https://doi.org/10.1109/CGO.2019.8661168

[21] Zhou R., Jones T.M. Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelisation. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Press, Washington, DC, USA; 2019. p. 15-25. (In Eng.) DOI: https://doi.org/10.1109/CGO.2019.8661196

[22] Fu S. -Y., Lin C. -M., Hong D. -Y., Liu Y. -P., Wu J. -J., Hsu W. -C. Work-in-Progress: Exploiting SIMD Capability in an ARMv7-to-ARMv8 Dynamic Binary Translator. *2018 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES).* IEEE Press, Turin, Italy; 2018. p. 1-3. (In Eng.) DOI: https://doi.org/10.1109/CASES.2018.8516794

[23] Arif M., Zhou R., Ho H. -M., Jones T. M. Cinnamon: A Domain-Specific Language for Binary Profiling and Monitoring. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, Seoul, Korea (South); 2021. p. 103-114. (In Eng.) DOI: https://doi.org/10.1109/CGO51591.2021.9370313

[24] Ottoni G. 2018. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. *ACM SIGPLAN Notices*. 2018; 53(4):151-165. (In Eng.) DOI: https://doi.org/10.1145/3296979.3192374

[25] Ajorpaz S.M., Garza E., Jindal S., Jiménez D.A. Exploring predictive replacement policies for instruction cache and branch target buffer. *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18).* IEEE Press; 2018. p. 519-532. (In Eng.) DOI: https://doi.org/10.1109/ISCA.2018.00050

[26] Khan T.A., Brown N., Sriraman A., Soundararajan N.K. Kumar R., Devietti J., Subramoney S., Pokam G.A., Litz H., Kasikci B. Twig: Profile-Guided BTB Prefetching for Data Center Applications. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21).* Association for Computing Machinery, New York, NY, USA; 2021. p. 816-829. (In Eng.) DOI: https://doi.org/10.1145/3466752.3480124

[27] Khan T.A., et al. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE Press, Valencia, Spain; 2021. p. 734-747. (In Eng.) DOI: https://doi.org/10.1109/ISCA52012.2021.00063

[28] Zhou K., Meng X., Sai R., Mellor-Crummey J. GPA: A GPU Performance Advisor Based on Instruction Sampling. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, Seoul, Korea (South); 2021. p. 115-125. (In Eng.) DOI: https://doi.org/10.1109/CGO51591.2021.9370339

[29] Ashouri A.H., Killian W., Cavazos J., Palermo G., Silvano C. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys*. 2019; 51(5):96. (In Eng.) DOI: https://doi.org/10.1145/3197978

[30] Savage J., Jones T.M. HALO: post-link heap-layout optimisation. *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020).* Association for Computing Machinery, New York, NY, USA; 2020. p. 94-106. (In Eng.) DOI: https://doi.org/10.1145/3368826.3377914

[31] Licker N., Jones N.M. Duplo: a framework for OCaml post-link optimisation. *Proceedings of the ACM on Programming Languages.* 2020; 4(ICFP):98. (In Eng.) DOI: https://doi.org/10.1145/3408980

Поступила 23.04.2021; одобрена после рецензирования 26.05.2021; принята к публикации 07.06.2021.
Submitted 23.04.2021; approved after reviewing 26.05.2021; accepted for publication 07.06.2021.

Об авторе:

Лисицын Сергей Алексеевич, ведущий инженер, Российский исследовательский институт Huawei (121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, корп. 2); аспирант, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: http://orcid.org/0000-0002-8904-0601,** lisitsyn.sergey@huawei.com

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Sergey A. Lisitsyn, Lead Engineer, Huawei Russian Research Institute (17 Krylatskaya St., building 2, Moscow 121614, Russian Federation); Postgraduate Student, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: http://orcid.org/0000-0002-8904-0601,** lisitsyn.sergey@huawei.com

The author has read and approved the final manuscript.