Санкт-Петербургский государственный университет

Программная инженерия

Лобанов Артем Алексеевич

Применение профильной информации в оптимизациях циклов LLVM

Бакалаврская работа

Научный руководитель:
ст. преп. Кириленко Я. А.

Рецензент:
главный инженер-программист, ООО “Синопсис СПб” Якушкин С. И.

Санкт-Петербург
2017

SAINT PETERSBURG STATE UNIVERSITY

Software engineering

Artem Lobanov

Profile-guided loop optimisations in LLVM

Graduation Thesis

Scientific supervisor:
senior lecturer Iakov Kirilenko

Reviewer:
staff engineer, Synopsys, Inc. Sergey Yakoushkin

Saint Petersburg
2017

Оглавление

Введение

Одной из важных составляющих, определяющих качество программного продукта, является эффективность компилятора, применяемого при разработке. В частности, компилятор должен обладать возможностью максимально оптимизировать код в соответствии с выбранными эвристиками. Оптимизации практически всегда производятся над каким-то внутренним представлением компилятора.

Методов оптимизации существует огромное количество: переупорядочивание инструкций, специальные оптимизации времени связывания и многие другие. В частности, значительный интерес вызывают оптимизации циклов, так как циклы часто являются одними из самых нагруженных участков программ. В каком-то виде оптимизации циклов реализованы во всех распространённых компиляторах, однако далеко не все компиляторы используют по отношению к циклам т.н. *профильные оптимизации* (profile-guided optimizations, PGO) — оптимизации, использующие информацию о нагрузке различных частей программы, полученную посредством анализа работы программы в прошлом. Обладая такой информацией, возможно получить представление о типичных запусках программы и сгенерировать код, оптимальный специально для подобного характера работы программы.

Стоит помнить о том, существуют два основных направления оптимизации: оптимизация производительности программ и оптимизация размера кода программ. Последняя особенно актуальна для встраиваемых систем, систем «интернета вещей» (Internet of Things, IoT) и других, где ввиду жёсткого ограничения размера устройства размер памяти также строго ограничен. В реальном мире эти направления оптимизации балансируются, а также, как правило, существует возможность явно указывать, какой тип оптимизации необходим в конкретной ситуации.

Одним из проектов, предоставляющих интерфейс как к оптимизации циклов, так и к генерации профильной информации, является проект LLVM [9], в рамках которого выполняется данная работа. Особый

интерес данный проект представляет потому, что в LLVM используется лаконичное и выразительное внутреннее представление, позволяющее реализовывать сложные, но в то же время легко поддерживаемые системы оптимизаций. Проект активно развивается, однако на момент проведения исследования профильные оптимизации в нём недостаточно развиты, хотя и существует общий механизм генерации и использования профильной информации.

В данной работе представлены методы использования профильной информации в оптимизаторе LLVM, позволяющие более точно и аккуратно оптимизировать циклы, учитывая как требования производительности, так и требования размера кода. Приводится анализ работы оптимизаций, сравнения характеристик производительности и размера кода, а также результаты проведенных экспериментов.

Постановка задачи

Целью данной работы является разработка и реализация профильных оптимизаций циклов в рамках инфраструктуры LLVM. Для достижения поставленной цели были сформулированы следующие задачи:

- изучить архитектуру проекта LLVM, реализованные в нём подходы к оптимизациям;
- разработать методы применения профильной информации к оптимизациям циклов, в частности к векторизации и раскрутке циклов;
- реализовывать разработанные методы в рамках оптимизирующей системы LLVM;
- провести сравнительный анализ работы методов, оценить разницу в производительности программ до и после оптимизаций.

1. Обзор предметной области

1.1. Общая архитектура LLVM

Важной особенностью современных компиляторных инфраструктур является наличие собственного внутреннего представления кода программ, удобного для выполнения над ним преобразований, и определенного вида модульной структурой. Компоненты таких компиляторов разделяются на 3 типа:

- «фронтенды» — транслируют код исходной программы в промежуточное представление языка;
- оптимизаторы — выполняют различного рода преобразования над промежуточным представлением, призванные повысить производительность генерируемого кода, сохраняя его корректность;
- «бекенды» (генераторы кода) — генерируют на основе оптимизированного промежуточного представления код ассемблера или машинный код для требуемой архитектуры.

Таким образом, для реализации в рамках таких инфраструктур нового компилятора для нового языка требуется лишь написать «фронтенд» для этого языка, сразу получив возможность генерировать код для различных архитектур. Аналогично для поддержки всеми языками новой архитектуры необходимо реализовать только генератор кода для этой архитектуры. Оптимизации же возможно производить в рамках одного только промежуточного представления, не задумываясь об особенностях синтаксиса языков и системах команд архитектур.

Из-за вышеописанного устройства архитектуру подобных систем принято называть *retargetable* архитектурой (retarget — переориентировать). Инфраструктура LLVM, в частности, также реализует retargetable архитектуру.

Профильная оптимизация реализована в какой-то форме в основных retargetable оптимизирующих компиляторах, однако все основные реализации имеют определенные значительные недостатки:

- GNU Compiler Collection (GCC) в значительном объеме использует профильную информацию, однако промежуточные представления;

- Intel C/C++ Compiler (ICC) является коммерческим продуктом и поддерживает лишь очень небольшое количество языков программирования;

- Low Level Virtual Machine (LLVM) имеет выразительное внутреннее представление и поддерживает множество языков, содержит мощную систему профилирования, но на данный момент лишь очень небольшое число оптимизаций используют предоставляемую профильную информацию.

Таким образом, при реализации в LLVM профильной оптимизации данная система была бы лишена всех вышеописанных недостатков. В рамках данной работы основным объектом оптимизации были выбраны циклы, так как они, с одной стороны, часто являются ключевыми с точки зрения производительности элементами программ, а с другой — имеют широкий потенциал для оптимизаций.

Промежуточное представление LLVM IR [8] является строго типизированным, практически полностью платформо-независимым; код LLVM IR всегда находится в *static single assignment* (SSA) форме, что позволяет для каждого вхождения переменной тривиально и однозначно находить соответствующее её определение. Одним из важных средств поддержания SSA-представления являются т.н. ϕ-функции, специализированные функции, возвращающие различные значения в зависимости от того, какой блок кода являлся предшественником текущего.

1.2. Типы профильной информации в LLVM

Система LLVM предоставляет 2 основных типа профилирования — *сэмплинг* (sampling-based profiling) и *инструментирование* (instrumentation-based profiling). Инструментирование является более

точным способом анализа производительности программы, однако, ввиду значительных изменений в коде, программы, включающие в себя сбор профильной информации при инструментировании, работают значительно медленнее оригинальных программ. По причине того, что для анализа циклов нужно иметь подробную информацию о том, насколько «нагружен» тот или иной участок программы, в данной работе в качестве основного типа профилирования использовалось инструментирование.

При профилировании через инструментирование в LLVM генерируются специальные файлы с расширением «.profraw», которые с помощью утилиты llvm-profdata сливаются в один результирующий файл «.profdata», содержащий информацию о произвольном количестве запусков программы. Данный файл используется непосредственно оптимизатором (утилита opt) и компилятором внутреннего представления (драйвер бекендов, утилита llc) для реализации профильных оптимизаций.

Существует 2 основных аспекта представления профильной информации в LLVM: частота базовых блоков (*block frequency* [6]) и вероятности дуг в графе управления программы (*branch probability*).

Частота базовых блоков показывает насколько часто в программе используется данный базовый блок по отношению к остальным. Является относительной величиной, сравнимой только в рамках конкретной функции. Значение величины получается разделением условного её максимума — значения начального блока функции, в соответствии с потоком управления. При вычислении частоты блоков для циклов учитывается дополнительный параметр, *loop scale* данного цикла — метрики, отражающей количество итераций, выполняемых программой для каждого входа в участок кода, содержащий цикл.

Вероятность дуг отражает то, насколько ожидаемо принятие программой данной метки в данной инструкции ветвления. Например, если за все время запусков программы в данной инструкции ветвления всегда принималась только одна метка, то эта метка будет иметь вероятность 1, а все остальные метки - вероятность 0.

2. Методы применения профильной информации

В рамках работы были выбраны несколько оптимизаций, в которых возможно выгодным образом применить профильную информацию:

- векторизация циклов [2] — распараллеливание итераций циклов на архитектурах, поддерживающих SIMD инструкции (single instruction multiple data);
- раскрутка циклов — разворачивание (возможно частичное) тела цикла для снижения вычислений, управляющих итерациями.

Данные оптимизации включают в себя несколько факторов, на которые возможно повлиять при наличии профильной информации.

2.1. Коэффициент векторизации

При векторизации в первую очередь рассматривается *коэффициент векторизации* (vectorization factor) — количество элементов в векторах, которыми оперируют SIMD инструкции [10]. Коэффициент векторизации, как правило, выбирается на основне информации о наибольшей длине SIMD регистров в данной системе. Однако на коэффициент также влияют и другие факторы, например, среднее ожидаемое количество итераций цикла — величина, которой возможно управлять с помощью профильной информации.

2.2. Скаляризованная версия цикла

При векторизации циклов число итераций не всегда кратно наиболее выгодному коэффициенту векторизации, ввиду чего возникает необходимость последние несколько итераций цикла исполнять скаляризованно, без использования векторных регистров.

Среди недостатков реализации викторизации в LLVM стоит отметить, что при оценке коэффициента векторизации не учитывается, насколько сильно скаляризованные итерации могут повлиять на время

работы цикла. При наличии профильной информации появляется возможность определять, какое количество скаляризованных итераций удовлетворительно для данного коэффициента векторизации, а какое количество итераций может не обеспечить значительный прирост производительности, даже несмотря на векторизацию.

2.3. Чередованный доступ к данным

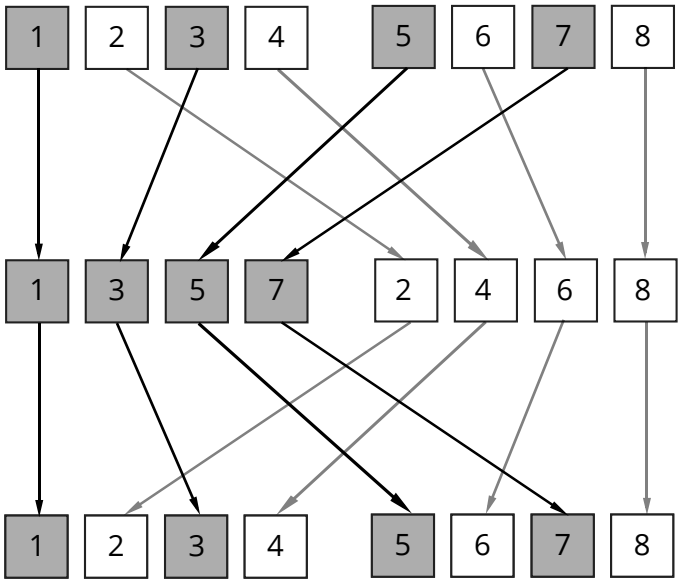

Рис. 1: Пример чередованного доступа к массиву: аналогичные операции выполняются над всеми чётными/всеми нечётными элементами.

Кроме того при векторизации часто возникает потребность обрабатывать чередующиеся данные, когда доступ к элементам массива производится по шаблону

$$a[i] = b + \delta u i \ , \tag{1}$$

например, заполнять по различным шаблонам чётные и нечётные элементы массивов и т.д. В формуле (1) b — начальный адрес массива, u — размер единицы адресации, а δ — *коэффициент интерливинга*

(interleaving factor, stride), величина, отражающая максимальную длину участка массива без чередующихся элементов [11]. Коэффициент интерливинга также является важным параметром векторизации.

Существует несколько основных случаев, когда интерливинг может обеспечить значительный прирост производительности:

- в цикле присутствуют редукции — подсчёт некоторых значений, аккумулируемых по ходу цикла;
- в теле цикла поддерживается небольшое количество актуальных значений одновременно, есть возможность увеличить нагрузку на регистры;
- тело цикла занимает незначительную часть времени исполнения по сравнению с накладными расходами на обеспечение итерирования.

В случае наличия редукций основной выигрыш в том, что удается уменьшить зависимости по данным. При интерливинге в случае задержки доступа к определенной части данных (например, при отсутствии данных в кэше процессора) возможно внеочередное исполнение инструкций (*out-of-order execution*), оперирующих над данными, доступ к которым не требует задержки.

Интерливинг имеет много общего с раскруткой циклов, в процессе интерливинга также происходит линейный рост объёма тела цикла. Однако в отличие от раскрутки интерливинг предполагает сохранение порядка между соответствующими инструкциями конкретной итерации, ввиду чего для анализа интерливинга необходимо рассматривать отличные от раскрутки циклов зависимости по данным.

При раскрутке циклов используется коэффициент раскрутки (*unroll factor*), аналогичный коэффициенту интерливинга в случае векторизации.

3. Реализация

В ходе работы были разработаны следующие основные методы применения профильной информации к оптимизациям циклов:

- профильная информация очередного запуска программы может быть предсказана на основе некоторого числа наборов профилей, сформированных при предыдущих запусках программы;
- в случае неизвестного количества итераций цикла коэффициент векторизации возможно определять на основе профильной информации; при этом для коэффициента векторизации существуют оптимальные эвристики;
- для более тщательного баланса между производительностью и размером кода есть возможность векторизовать и раскручивать только определенные части циклов на основе информации о частичном инлайнинге.

Ниже представлены особенности реализации каждого из данных методов.

3.1. Предсказание профильной информации

Характер работы программы может содержать определенные закономерности: например, может происходить линейный сдвиг более нагруженных участков кода в сторону определённых функций; также может наблюдаться сезонная зависимость нагруженности участков кода от времени и т.д. Чтобы использование данное наблюдение, в работе было реализовано предсказание профильной информации на основе нескольких наборов данных о предварительных запусках программы (см. Рис. 2) [3].

Для предсказания профильной информации был использован стандартный метод предсказания временных рядов — т.н. *long short-term memory network* (LSTM) — разновидность рекуррентной нейронной сети, оптимизированная для решения задач, в которых необходимо на по-

следующих этапах работы сети запоминать результаты работы предыдущих. В частности, в качестве библиотеки, предоставляющий интерфейс создания и тренировки LSTM, была использована TensorFlow.

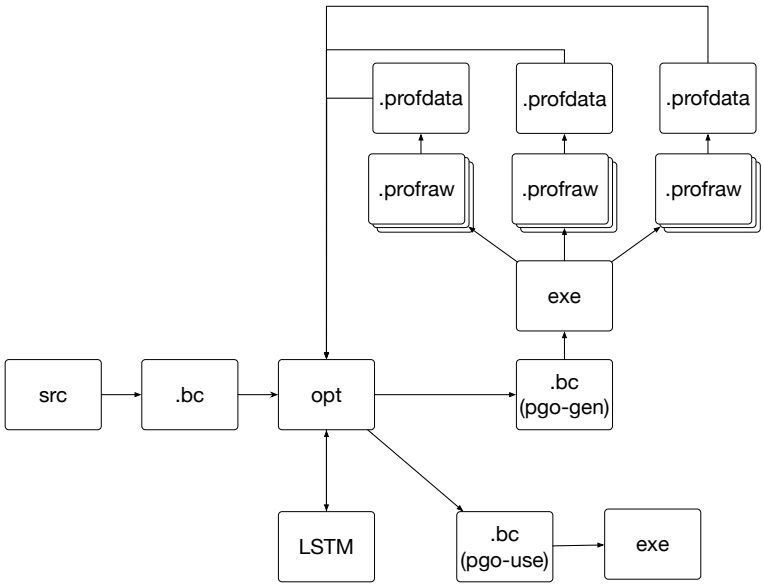

Рис. 2: Схема работы оптимизатора при использовании предсказания профильной информации.

Предсказание реализуется на основе *счётчиков профилирования* (profile counts) — абсолютных значений того, сколько раз в программе исполнялась та или иная инструкция. Такой подход позволяет предсказывать будущие профили ещё на этапе генерации LLVM IR, содержащего PGO-информацию. Таким образом, отсутствует необходимость генерировать в промежуточном представлении сложные наборы метаданных [7], хранящие данные сразу о нескольких измерениях профильной информации.

Для передачи набора профилей в модуль предсказания реализуется обход *минимального остовного дерева* (minimum spanning tree, MST) функции, счётчики профилирования линеаризуются, передаются в линеаризованном виде модулю предсказания, после чего дерево функции заполняется предсказанными значениями счётчиков.

3.2. Оптимизация коэффициента векторизации

Стандартно коэффициент векторизации выбирается, как коэффициент, при котором оцениваемая стоимость (*cost*, отражение в условных единицах ожидаемого времени исполнения программы) для одной итерации цикла была бы максимальной. Конкретно, выражение для определения коэффициента следующее:

$$VF_{best} = x \mid x = 2^t for\ some\ t\ \textbf{\textit{and}}\ x = \arg\min_{VF} cost(VF).$$

Такая оценка коэффициента эффективно работает при большом количестве итераций, если количество скаляризованных итераций незначительно по сравнению с количеством векторизованных итераций. При небольшом количестве итераций скаляризованные версии цикла (таких итераций может быть до 63 при использовании регистров AVX512) могут занимать значительную часть времени работы программы. Учитывая тот факт, что циклы с небольшим числом итераций (<1000) используются в программах гораздо чаще, чем циклы с бóльшим числом итераций, повысить эффективность их исполнения особенно важно.

При числе итераций (*trip count*), которое невозможно определить посредством анализа *scalar evolution*[1], в LLVM векторизация запрещена полностью. Более того, в случае векторизации независимо от доступности числа итераций в силу вышеописанных причин код, генерируемый оптимизатором, часто неэффективен.

Таким образом, во-первых, в ходе работы была реализована поддержка векторизация при наличии профильной информации для данного цикла даже при неизвестном числе итераций. Кроме того, был реализован анализ стоимости работы цикла, учитывающий скаляризованные итерации, возникающие при данном коэффициенте векторизации. Формально усовершенствованная оценка коэффициента векторизации отражена в следующем выражении:

[1]Scalar evolution — оптимизационный проход в LLVM, анализирующий неявно заданные, но известные программе значения переменных.

$$VF_{best} = x \mid x = 2^t for\ some\ t\ \textbf{and}$$
$$x = \arg\min_{VF} \left(cost(VF) * \left\lfloor \frac{TC}{VF} \right\rfloor + (TC \bmod VF) \times cost(1) \right).$$

В результате изменения метода выбора коэффициента векторизации количество инструкций векторизуемого цикла в среднем уменьшилось в несколько раз (см. Рис. 3). Особенно заметно ускорение для некоторых промежутков с небольшим числом инструкций.

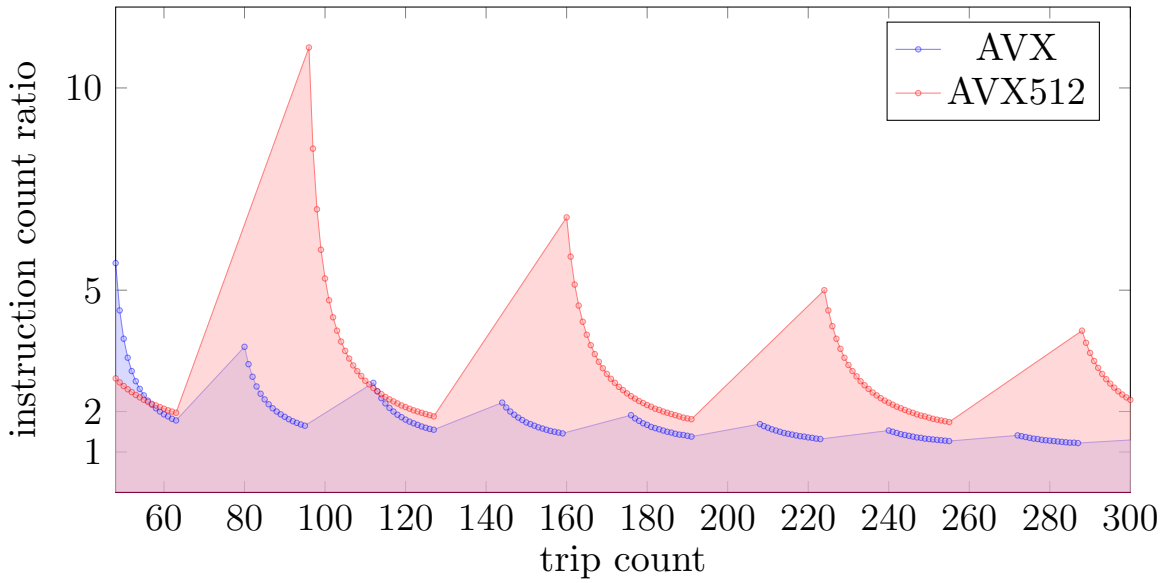

Рис. 3: Уменьшение числа инструкций с усовершенствованным VF в зависимости от числа отераций цикла.

3.3. Расширение частичного инлайнинга

В ходе работы было решено использовать вспомогательные оптимизации, в которых анализ профильной информации может дать результаты, полезные для векторизации и раскрутки. В частности, в качестве основной вспомогательной оптимизации был выбран частичный инлайнинг [4, 5].

Данная оптимизация представляет собой вынесение частей функции за её пределы. Как правило, частичный инлайнинг применяется в случае, если функция представляет из себя последовательность некоторой

инициализации, условного выражения и финализации. В частности, в LLVM на момент исследования частичный инлайнинг был реализован только для случая условия с единственным условным блоком.

```
caller() {
    ...
    callee()
    ...
}

callee() {
    <callee init>
    if (cond) {
        <cond = true>
    } else {
        <cond = false>
    }
    <callee finalize>
}
```

```
caller() {
    ...
    <callee init>
    if (cond) {
        callee_true(...)
    } else {
        calee_false(...)
    }
    <callee finalize>
    ...
}

callee_true(...) {
    <cond = true>
}

callee_false(...) {
    <cond = false>
}
```

Листинг 1. Простейший пример частичного инлайнинга: слева вызов функции до оптимизации, справа — после частичного инлайнинга.

Частичный инлайнинг данной функции func реализуется следующей последовательностью операций:

- копирование исходной функции в функцию func_copy, сохранение оригинальной копии для сохранения корректности указателей на функцию и др. конструкций;
- обработка ϕ-функций: разделение ϕ-инструкций на инструкции, получающие значения из условных блоков, и на инструкции, получающие значения из блока инициализации;
- извлечение условных блоков в отдельные функции;

- инлайнинг функции func_copy.

В рамках работы было принято решение расширить поддержку частичного инлайнинга на произвольное число условных блоков. Таким образом, в высокоуровневых языках появилась возможность использовать if/else-цепи и switch-инструкции для частичного инлайнинга.

Кроме того, было реализовано 2 варианта частичного инлайнинга относительно финализирующего блока:

- частичный инлайнинг со слиянием условных блоков в единый финализирующий блок, выполняющийся независимо от выполнения условных блоков;

- частичный инлайнинг без финализирующего блока, с потоком управления функции, завершающимся в конце каждого условного блока.

В случае без финализирующего блока необходимо проверять, чтобы все условные блоки были независимы по инструкциям, т.к. это необходимое условие вынесения каждого условного блока в отдельную функцию.

Данная оптимизация прежде всего позволяет избежать накладных расходов на накладных расходов на вызов функции, которые могут включать в себя генерацию информации для обработки исключений и других видов т.н. информации фрейма (*call frame information, CFI*). Частичный инлайнинг особенно актуален ввиду того, что накладные расходы являются существенными в коде практически любой структуры.

3.4. Балансировка производительности и размера кода

Стандартно в компиляторах доступны опции оптимизации размера кода, возможно доступны доволнительные опции, жертвующие для минимизации размера производительностью. Однако при разработке для конкретной архитектуры может возникнуть необходимость не сжимать код настолько, насколько это возможно, а лишь обеспечить максимальную производительность при условии, что размер будет достаточен для данной архитектуры.

Для решения данной проблемы было предложено реализовать в LLVM более тщательную балансировку производительности и размера кода. В частности, были использованы следующие критерии:

- оценивать необходимость частичного инлайнинга на основе профильной информации;
- не применять векторизацию и раскрутку к циклам в функциях, определенных эвристиками частичного инлайнинга неподходящими для оптимизации;
- варьировать в соответствии с параметрами балансировки коэффициенты интерливинга и раскрутки.

При использовании в частичном инлайнинге профильной информации были выбраны эвристики:

- высокое значение block frequency условных блоков;
- равномерное распределение branch probability для условных блоков — для уменьшения возможных потерь при небольшом изменении поведения;
- количество инструкций в исходной функции, а также в функции, полученной после извлечения условных блоков;
- количество точек вызова исходной функции — в случае повторных проходов частичного инлайнинга.

При варьировании параметров интерливинга и раскрутки учитывается, что данные параметры изначально максимизируются компилятором для создания оптимальной нагрузки на регистры (*register pressure*). Таким образом, балансировать данные параметры без неоправданных потерь в производительности имеет смысл лишь в диапазоне до значения, изначально определённого компилятором.

Коэффициент, варьируемый при раскрутке, аналогичен коэффициенту интерливинга при векторизации: в частности, если при попытке векторизации выбирается коэффициент векторизации 1 (то есть фактически векторизация оценивается как не приносящая выигрыша по производительности), то оптимизатор LLVM в принципе не пытается применять интерливинг цикла, вместо этого используется стандартная раскрутка во избежание накладных расходов на стандартные механизмы векторизации.

4. Эксперименты

Для оценки работы предложенных методов оптимизации были проведены 2 группы экспериментов:

- оценка производительности векторизации — для модифицированного коэффициента векторизации и предсказания профильной информации в случае сезонной работы программы;
- оценка балансировки параметров оптимизации — анализ комбинаций производительности и размера кода программ.

Эксперименты проводились на компьютере со следующими системными характеристиками:

- CPU — Intel Core i7-6700k;
- RAM — 16 Гб DDR4.

Время работы отдельных участков программы замерялось с помощью профилировщика perf.

4.1. Оценка производительности векторизации

При тестировании параметров векторизации была использована библиотека gcc-loops [1], классический набор программ для оценки качества оптимизаций циклов. Были выбраны несколько тестов, наиболее явно отражающих преимущества реализованных оптимизаций. В каждом тесте было вычислено среднее значение по параметру, числу итераций в основном цикле теста, в диапазоне значений, повышающих производительность при изменённой оценке коэффициента векторизации. Для анализа результатов предсказания профильной информации эксперименты запускались с цикличными значениями параметров, с длиной цикла, равной 4. Результаты приведены на Рис. 4.

Эксперименты показывают, что уточнённая метрика коэффициента векторизации даже при непостоянном числе итераций в среднем может

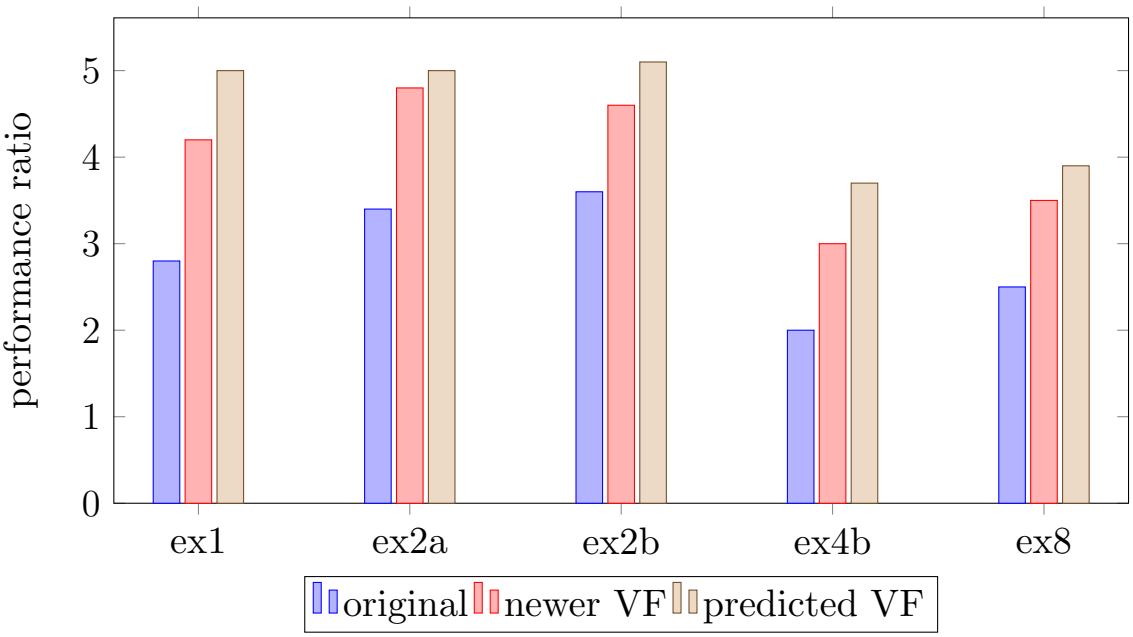

Рис. 4: Сравнение различных метрик для коэффициента векторизации: исходная метрика, усовершенствованная, предсказанная.

значительно повысить производительность цикла; предсказание профильной информации с учётом сезонности работы программы также может дать прирост производительности, зависящий от резкости изменения параметров.

4.2. Оценка балансировки параметров оптимизации

При тестировании степени производительности, зависящей от размера кода, было проведено 2 группы экспериментов, отвечающих соответственнот за частичный инлайнинг и за коэффициенты интерливинга/раскрутки. Эксперименты, посвященные частичному инлайнингу, проводились с использованием библиотеки raymath — части более крупной библиотеки raylib [12]. Балансировка параметров в случае варьирования параметров интерливинга/раскрутки проводилась на основе библиотеки gcc-loops. Результаты приведены на Рис. 5.

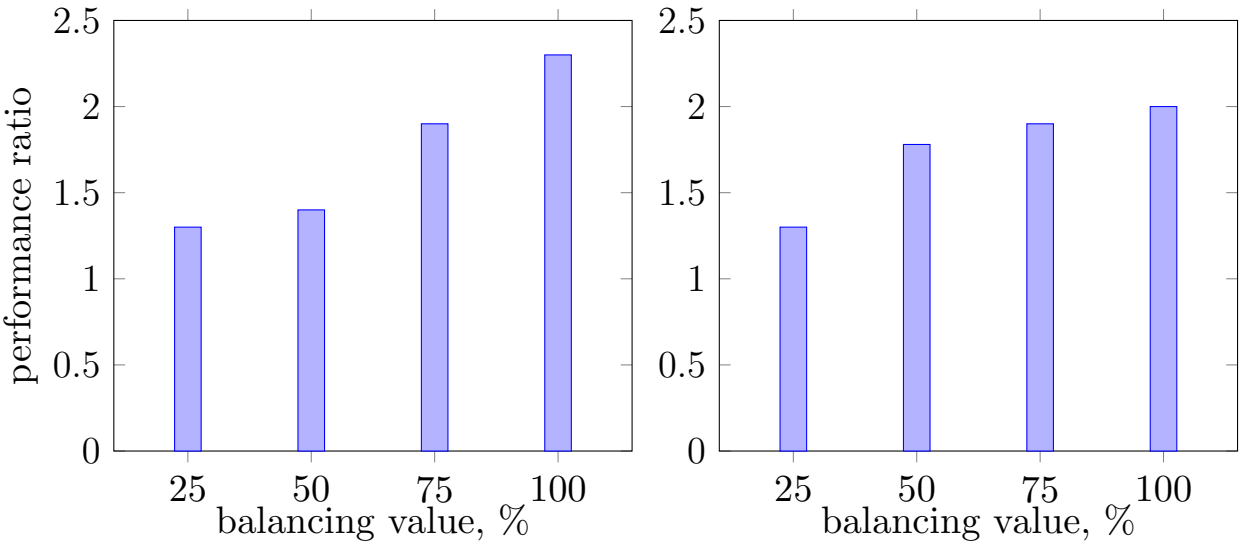

Рис. 5: Оценка балансировки производительности и размера кода.

Как можно заметить, полученный рост отношения размера кода и производительности практически линейный в случае частичного инлайнинга; в случае же варьирования параметров интерливинга/раскрутки изменение производительности уменьшается с увеличением значения параметра.

Заключение

В рамках данной работы были разработаны и реализованы в инфраструктуре LLVM методы использования профильной информации, повышающие качество оптимизаций циклов. Показано, что полученные методы обеспечивают более тщательный анализ как производительности целевого кода, так и его размера, что позволяет использовать их для различных типов систем в зависимости от того, какие именно параметры наиболее важны для данной системы. В частности, были получены следующие результаты:

- оптимизация частичного инлайнинга расширена на случай произвольного количества условных блоков;
- разработана усовершенствованная эвристика определения коэффициента векторизации;
- разработана и реализована инфраструктура, позволяющая предсказывать профильную информацию на основе предыдущих данных о профилях;
- разработана и реализована система балансировки производительности и размера кода относительно оптимизаций циклов;
- проведен сравнительный анализ предложенных оптимизаций с уже существующими, продемонстрировано повышение качества генерируемого кода как по производительности, так и по размеру кода.

Список литературы

[1] Auto-Vectorization in GCC. — URL: http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] Auto-Vectorization in LLVM. — URL: http://llvm.org/docs/Vectorizers.html.

[3] Cavazos William Killiana, Renato Miceli, Eun Jung Parka, Marco Alvarez Vegaa, John. Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics. — Partnership for Advanced Computing in Europe (PRACE) Performance Prediction, 2014.

[4] Grove Bowen Alpern, Anthony Cocchi, David. Some new approaches to partial inlining. — VMIL'12 Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages, 2012. — P. 39–48. — URL: http://doi.org/10.1145/2414740.2414749.

[5] Kim Jun-Pyo Lee, Soo-Mook Moon, Suhyun. Aggressive Function Splitting for Partial Inlining. — INTERACT-15 Proceedings 15th Workshop on Interaction between Compilers and Computer Architectures, 2011. — URL: http://doi.org/10.1109/INTERACT.2011.14.

[6] LLVM Block Frequency Terminoligy. — URL: http://llvm.org/docs/BlockFrequencyTerminology.html.

[7] LLVM Branch Weight Metadata. — URL: http://llvm.org/docs/BlockFrequencyMetadata.html.

[8] LLVM Language Reference Manual. — URL: http://llvm.org/docs/LangRef.html.

[9] Lattner Chris, Adve Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. — CGO'04 Proceedings of the 2004 International Symposium on Code Generation and Optimization, 2004.

[10] Padua Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong. David A. An Evaluation of Vectorizing Compilers. In PACT '11 Proceedings of the 2011 International. — PACT'11 Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011. — P. 372–382. — URL: http://doi.org/10.1109/PACT.2011.68.

[11] Zaks Dorit Nuzman, Israel Ira Rosen, Israel Ayal. Auto-vectorization of interleaved data for SIMD. — PLDI'06 Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006. — P. 132–143. — URL: http://doi.org/10.1145/1133255.1133997.

[12] raylib. — URL: http://www.raylib.com.