АО «МЦСТ»

На правах рукописи

Четверина Ольга Александровна

ПОВЫШЕНИЕ КАЧЕСТВА КОМПИЛЯЦИИ КОДА В РЕЖИМЕ ПО УМОЛЧАНИЮ

05.13.11 - математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
к.ф.-м.н.
Нейман-заде Мурад Искендер оглы

Москва – 2019

Содержание

Введение.

Актуальность темы исследования.

Производительность современных вычислительных машин в значительной мере зависит от качества планирования исполняемого кода оптимизирующими компиляторами в соответствии с особенностями архитектуры.

С целью обеспечения оптимального планирования, в процессе компиляции к коду применяется последовательность преобразований, которые формируют более эффективный семантически эквивалентный код. Для части преобразований качественный результат их применения можно определить однозначно по анализируемому коду. Однако в большинстве случаев оптимальный способ применения преобразований либо зависит от недоступных на этапе компиляции характеристик исполнения кода, таких как профиль исполнения или особенности обрабатываемых данных, либо не может быть точно вычислен за приемлемое время. Для более точного применения преобразований используются различные техники, включающие в себя использование полученного на тренировочных данных профиля исполнения кода (PGO)[12], обеспечение возможности одновременного анализа кода всей программы (IPA)[13] или явная настройка оптимизаций регулирующими опциями. Максимальная производительность кода, которую удается при этом получить, называют пиковой производительностью. Разница в производительности кода, собранного в режиме по умолчанию и при пиковой сборке, особенно значительна для архитектур со статическим планированием кода, поскольку в динамическом случае работу по уточнению профиля и анализу данных берет на себя архитектура. Еще большее отличие возникает при компиляции под архитектуры с высокой параллельностью на уровне команд (EPIC) из-за высокой спекулятивности и неточности оценки реальной параллельности кода.

Однако указанные техники достижения пиковой производительности не

всегда доступны для использования. И построение профиля исполнения, и явная настройка оптимизаций, как ручная, так и автоматическая [14][15][16] [17], требуют тренировочного исполнения приложения. При этом качественно подобрать входные данные и получить достоверный полный профиль исполнения удается только для сравнительно небольших приложений, а использование неточного профиля может приводить к ощутимому отрицательному эффекту. Так, автором было замерено, что отключение оптимизации неиспользуемых в тренировочном запуске процедур профессионально сформированного для тестирования производительности пакета spec2000 CFP [18], приводит к среднему замедлению на 20% исполнения приложений на машине Эльбрус с EPIC архитектурой [19].

Вторая проблема достижения высокой производительности заключается в значительных затратах времени на осуществление компиляции. Даже в процессе проведения компиляции в режиме по умолчанию компилятором Эльбрус осуществляется более 300 этапов компиляции, и затрачивается в среднем в 20 раз больше времени, чем в случае неоптимизирующей сборки. Для ряда задач такой расход времени становится неприемлемо большим при предварительной компиляции и недопустимым для Just-in-time компиляции, когда задержка по компиляции приводит к более продолжительной работе плохо оптимизированного кода. Техники же многократной компиляции с использованием различных опций оптимизации и сравнением времен исполнения с выбором наименьшего, или итерационные решатели, реализованные для процессоров ARM, серверов Sun и для других вычислительных машин [20][21][22], практически не применяются промышленно. То есть, помимо проблемы недоступности ряда техник, повышающих качество оптимизации кода, пользователи сталкиваются и с проблемой ограниченного времени, доступного для анализа и компиляции кода.

Необходимость повышения скорости исполнения кода в режиме по умолчанию с одновременным учетом времени компиляции определяет актуальность диссертационной работы.

Цель исследования.

Целью представленного исследования является поиск и обоснование методов автоматического повышения качества компиляции кода в режиме по умолчанию, доступному к применению для большинства приложений. В соответствии с этой целью были поставлены следующие задачи:

- доработать имеющиеся и разработать альтернативные оптимизационные преобразования, позволяющие приблизить базовую оптимизацию к пиковой;
- разработать наборы оптимизационных последовательностей с настройками;
- разработать метод качественного учета времени исполнения, времени компиляции и других характеристик;
- разработать механизм поиска лучшего набора оптимизаций по промежуточному представлению.

Научная новизна.

Научной новизной обладают следующие результаты работы.

- Построена единая функция для попроцедурной многокритериальной оценки качества компиляции.
- Теоремы о соответствии точек на границе Парето минимумам функционала и о представлении функций, сохраняющих порядок при растяжениях.
- Утверждение о представлении функции качества компиляции для

одновременного учета времени исполнения и времени компиляции.

- Формулировка задачи минимизирующей классификации с заданной функцией потери. Алгоритм ее решения и метод перехода к аддитивной функции потери для отдельных минимумов. Теоремы о сходимости.
- Методы оптимизации циклов посредством раскрутки части их путей и теоремы, позволяющие оценить их эффективность. Метод выбора адресов данных для подкачки нерегулярных чтений структур. Алгоритм конвейеризации с забросом чтений и ограничением планируемого времени исполнения цикла.

Теоретическая и практическая значимость.

Теоретическую ценность представляет собой обоснование постановки задачи машинного обучения в виде требования минимизации нелинейного функционала качества, зависящего от назначения классов объектам всей выборки, предложенные алгоритмы решения такой задачи и доказанные утверждения об их сходимости. Интерес представляют и доказанные утверждения о строении многокритериального функционала качества, удовлетворяющего свойству сохранения монотонности при растяжениях.

Практическую значимость представляют предлагаемые оптимизационные преобразования, доступные для применения в базовом режиме компиляции; разработанный оптимизационный набор оптимизирующих последовательностей; механизм построения классификационного решения на основе статистической информации о компиляции тренировочного пакета задач. Эффективность описанных механизмов и методов оценивалась путем замера времени исполнения и компиляции на вычислительном комплексе с микропроцессором «Эльбрус» с VLIW архитектурой. Замеры производились на задачах пакетов, разработанных для замера производительности процессоров: SPEC CPU95, SPEC CPU2000, SPEC CPU2006.

Методология и методы исследования.

Методы исследования заимствованы из областей системного программирования, технологии компиляции и различных областей математики:

- методы анализа потоков данных и управления программы для выявления способов повышения параллельности на уровне операций;
- методы анализа работы памяти вычислительных машин при исполнении приложений;
- методы математической статистики;
- методы математического анализа;
- методы машинного обучения;
- методы теории графов;
- методы математической логики;
- методы теории алгоритмов.

Положения, выносимые на защиту.

В процессе проведения диссертационного исследования были получены следующие результаты, выносимые на защиту.

- Разработаны методы частичной раскрутки коротких путей и рекуррентностей циклов, повышающие качество слияния кода сложных циклов при статическом планировании. Доказаны утверждения об оценке их эффективности. Предложены неагрессивные методы оптимизации работы с памятью.
- Предложена числовая многокритериальная оценка качества компиляции, учитывающая попроцедурный характер оптимизации. Доказаны утверждения о ее представлении и показано ее преимущество по сравнению с вероятностной оценкой качества.
- Сформулирована задача минимизирующей классификации, соответствующая обучению с построенной оценкой качества

компиляции. Разработаны алгоритмы ее решения и доказаны утверждения об их сходимости. Построенная в соответствии с разработанными методами классификация попроцедурного назначения оптимизационной последовательности внедрена в промышленный компилятор для архитектуры Эльбрус.

Апробация.

Результаты, полученные в работе, были доложены на научных конференциях и семинарах:

- 56-ой научной конференции МФТИ, Москва, МФТИ, 2013;
- Национальном Суперкомпьютерном Форуме НСКФ-2013;
- I-ой Всероссийской научно-технической конференции «Расплетинские чтения», Москва, 2014;
- Spring/Summer Young Researchers' Colloquium on Software Engineering – SYRCoSE, Самара, 28-30 May, 2015;
- II-ой Международной конференции «Инжиниринг & Телекоммуникации — En&T», Москва/Долгопрудный, 2015.
- Open Conference on Compiler Technologies, Москва, РАН, 2015;
- Семинаре кафедры МаТИС механико-математического факультета МГУ им. М.В. Ломоносова, 2018.

Публикации.

По теме диссертации опубликовано 10 печатных работ [1]-[10] и получено свидетельство о государственной регистрации программы для ЭВМ [11]. Работы [4], [5], [6], [8], [9], [10] опубликованы в изданиях из списка ВАК на русском и иностранном языке. Статья [10] опубликована в журнале, который также входит в списки Scopus и Web of Science. В работах [3], [4] и публикации [11] личный вклад автора заключается в выборе числовой оценки качества компиляции, создании наборов оптимизационных последовательностей и в разработке и реализации методов машинного

обучения для выбора последовательности из набора. В совместной работе [5] вклад автора заключается в анализе программ и выбору для них пиковых наборов опций компиляции, а также в разработке новых методов оптимизации кода и предложениях по коррекции ранее реализованных. В совместной работе [8] автором предложен метод последовательного выбора пар для инлайн-подстановок на основе изменения значения функции, зависящей от предсказанных характеристик компиляции.

Личный вклад автора.

Все представленные в диссертации результаты получены лично автором.

Структура и объем работы.

Работа состоит из введения, четырех глав и заключения. Основной текст диссертации (без приложений и списка литературы) занимает 115 страниц, общий объем – 125 страницы с 18 рисунками и 6 таблицами. Список литературы содержит 73 наименования.

Глава 1. Повышение качества работы оптимизирующих компиляторов

1.1 Оптимизация кода

Высокая скорость исполнения программ компьютерами определенной архитектуры достигается путем преобразований *(оптимизаций)* кода, выполняемых разработанным применительно к ее специфике компилятором. Они обеспечивают прирост производительности за счет более удачного планирования кода и повышения эффективности работы с памятью. Для архитектур со статическим планированием задача оптимизация кода является особенно важной и сложной, поскольку при этом, кроме проведения универсальных преобразований, сокращающих время исполнения, требуется заранее определить точную последовательность выполнения операций. То есть, уже на этапе компиляции необходимо произвести всю работу по выявлению и обеспечению параллелизма исполнения на уровне операций с учетом аппаратных возможностей, а также по предсказанию блокировок конвейера и уменьшению соответствующих потерь производительности [23]. Еще большую значимость оптимизация кода имеет для VLIW архитектур, поскольку в этом случае высокая скорость работы кода достигается за счет одновременного исполнения сразу нескольких команд в одном такте, и на компилятор ложится задача планирования эффективного и сбалансированного использования большого числа вычислительных устройств процессора.

Такие требования к оптимизации в сочетании с широкими предоставленными аппаратурой возможностями приводят к постоянному усложнению соответствующих компиляторов. Для иллюстрации можно привести в пример архитектуру микропроцессора «Эльбрус», которая поддерживает одновременное выполнение до 23 операций за такт. Операции выполняются в конвейере, поэтому с учетом всех особенностей аппаратуры на различных стадиях исполнения одновременно могут находиться несколько сот операций [23]. Такой зна-

чительный параллелизм требует существенной поддержки со стороны оптимизирующего компилятора [24][25][26][27]. Так, компилятор, разрабатываемый для процессора Эльбрус, в процессе режима компиляции по умолчанию выполняет более 300 этапов оптимизации кода. Это позволяет обеспечить высокую производительность, хотя требует значительных временных затрат.

Оптимизацию кода, или же оптимизационную последовательность, разделяют на межпроцедурную и попроцедурную часть. Обе части представляют из себя *последовательность аналитических, технических и оптимизационных фаз*.

В процессе межпроцедурной оптимизации компилятором lcc проводится межпроцедурный анализ указателей [28], применяется подстановка процедур [29], проводится глобальная прорагация констант, может производится клонирование некоторых процедур или разрезание процедур на участки меньшего размера и ряд менее важных вспомогательных преобразований [30]. Такие методы оптимизации позволяют осуществить удаление лишнего кода, уточнить характеристики исполняемого кода и повысить параллелизм исполнения операций. По завершении межпроцедурного этапа окончательно формируются процедуры с точки зрения логики исполнения, фиксируются все входные и выходные параметры отдельных участков кода.

Далее, вся работа по оптимизации и планированию кода производится независимо для каждой процедуры. Поскольку архитектура Эльбрус требует статического планирования кода, то на попроцедурном этапе, кроме универсальной оптимизации, осуществляется работа по эффективной подготовке и конвейеризации циклов, а для ациклических участков выделяются регионы, операции которых объединяются в гиперблоки. При этом для распараллеливания программ с разветвленным управлением используются спекулятивный и предикатный режимы исполнения операций, позволяющие выполнять некоторые вычисления заблаговременно или одновременно для нескольких условий управления. Такой способ сокращает время реального исполнения, но приводит к росту спе-

кулятивного кода. Как правило, для всех процедур приложения при этом используется общая, стандартная последовательность фаз компиляции.

Работа других компиляторов может отличаться по составу и последовательности применения преобразований. В первую очередь, разница между ними обусловлена особенностями целевой архитектуры. Однако, последовательность применяемых преобразований каждым компилятором, как правило, одинакова для всех оптимизируемых процедур и зависит только от режима и настройки дополнительных опций компиляции.

Оптимизирующий компилятор может создавать достаточно эффективный параллельный код при стандартном процессе компиляции. Однако, такая компиляция требует больших временных затрат, часть которых расходуется на незначительное ускорение последующего времени исполнения. При этом итоговый код зачастую значительно увеличивается в размерах из-за многочисленных проверок и дублирований, что иногда приводит к замедлению из-за блокировок по чтению кода. Кроме того, остается проблема более точной настройки и использования оптимизаций, которые либо применяются только при возведении дополнительных опций, либо на ряде задач приводят к замедлению работы кода. Поэтому для достижения максимальной, или *пиковой*, скорости исполнения приложений используются специальРис. 18. Использование линеек для выделенных алгоритмом КФК для частичной ошибки 5-ти областей на проверочном пакете приложений spec95. ные режимы и другие дополнительные методы настройки компиляции.

1.2 Методы настройки оптимизации

С целью получения *пиковой скорости исполнения*, ограничения размера кода или уменьшения времени компиляции используются различные механизмы настройки компиляторов. К ним относятся использование специальных режимов и взведение опций, которые могут влиять как на всю

оптимизационную последовательность, включая межпроцедурную и попроцедурную оптимизацию, так и на работу отдельных преобразований.

К режимам компиляции можно отнести:

1) Выбор общего уровня оптимизации - O1, O2, O3, O4.

   Уровень -On задает основной набор применяемых оптимизаций, в случае gcc это наборы опций, в компиляторе семейства Эльбрус - разные последовательности оптимизаций. Более высокий уровень оптимизации подразумевает использование большего набора агрессивных преобразований, которые могут значительно, то есть в разы, увеличивать размер кода и время компиляции. При этом на каких-то контекстах такие преобразования могут обеспечить значительное ускорение, на других же приводят к некоторому замедлению исполнения. В случае необходимости получения высокой производительности для компиляторов под Эльбрус обычно используется режим O3 и выше, для gcc основным режимом получения производительного кода является режим O2.

2) Компиляция программы *в режиме вся програма* (whole program optimization, WPO) [13].

   В этом режиме компилятору становится доступна информация одновременно обо всех модулях, то есть возможно провести межпроцедурный анализ (interprocedural analysis, IPA) и применять межмодульные преобразования (interprocedural optimizations, IPO).

3) Режим двухфазной компиляция со сбором и применением профиля программы.

   Эффективность применения ряда значимых оптимизаций может быть увеличена при использовании *профиля программы*, как правило представляющего из себя значений счетчиков, содержащих число исполнения линейных участков и переходов между ними. Для этого на первой фазе компиляции строится инструментированный код с дополнительными операциями, позволяющий собрать профиль

программы, после чего выполняется его тренировочный запуск. Использование полученного профиля на второй фазе компиляции дает дополнительное ускорение за счет более эффективного применения оптимизирующих преобразований [31] [32]. Такой режим компиляции называют PGO (profile guided optimization) [12].

4) Установка правил для кода программы и ее исполнения.

   Обычно это правила, которые при использовании по умолчанию могут приводить к различным ошибкам исполнения программы. Так, они могут передавать информацию о независимости указателей для объектов разных типов (-fstrict-aliasing); или разрешать применение преобразований, приводящих к уменьшению точности вычислений (-ffast, -ffast-math). Для некоторых языков или их расширений ряд таких правил может входить в стандарт [33][34], в этом случае могут использоваться опции, запрещающие их использование.

В целом режимы компиляции определяют набор доступной для анализа компилятором информации, ограничений на итоговый код и общую скорость компиляции. К сожалению, во многих случаях даже при необходимости получения эффективного кода использовать часть указанной информации весьма затруднительно.

Так, режим WPO может быть использован только при одновременноом наличии в процессе компиляции исходных кодов или специального промежуточного представления для всех модулей программы, которые в дальнейшем нельзя будет подменять. При этом иногда полное представление всей программы может оказаться слишком большим для проведения его анализа, поскольку для этого требуется строить дополнительные конструкции с информацией о взаимосвзязях между всеми операциями и операндами.

Часто возникают сложности со сбором и применением к компиляции профиля программы. Для ее результативности предшествующий

тренировочный запуск необходимо проводить на *представительных данных,* то есть, должны использоваться те же пути в управляющем графе, которые будут исполняться в последующих реальных запусках, при этом должны получиться аналогичные соотношения значений счетчиков между узлами (вершинами) управляющего графа. В случае, если тренировочные данные не соответствуют реальному исполнению программы, применение профилезависимых оптимизаций может приводить к замедлению результирующего кода. Проблема формирования представительных тренировочных данных может быть продемонстрирована даже на профессионально разработанном для тестирования производительности пакете spec2000. Так, тестовое применение к не используемым в тренировочном запуске процедур самой простой оптимизирующей последовательности привело к 6-ти процентному снижению производительности CFP-задач в среднем. При этом наибольшее замедление произошло на задаче 301.apsi (-47%) из-за того, что одна из ее основных процедур просто ни разу не выполнялась при тренировочных запусках. В случае же компиляции библиотек для проведения тренировочного запуска необходимо не только сформировать входные данные, но и разработать достаточно типичный вызывающий код. А для такой задачи как операционная система, тренировочный запуск представляет еще большую сложность, поскольку исполнение инструментированного кода приведет к изменению работы самой операционной системы.

Использование же дополнительных правил, которые позволяют получить значительное повышение скорости исполнения большинства приложений, запрещено по умолчанию, поскольку может приводить к ошибкам.

Поэтому самыми эффективным режимом, доступным для компиляции в большинстве случаев, можно назвать помодульный режим -O3 без использования профиля. Далее будем называть такой режим *базовым* режимом компиляции.

1.3 Условно полезные оптимизации

Применение некоторых преобразований является безусловно полезным, к примеру, удаление мертвого, то есть не использующегося в дальнейшем кода. Однако, значительную часть оптимизационной последовательности составляют *условно полезные* оптимизации или же оптимизации, требующие настройки параметров применения. Это свойство справедливо даже для таких важнейших оптимизации, как программная конвейеризация циклов или выделение регионов с последующим планированием исполнения операций под предикатами. Так, конвейеризация цикла (softpipe) с увеличением логической итерации окажется вредной с точки зрения производительности, если итерация цикла будет исполнена только один раз. Мало того, результат любого преобразования кода в конечном счете зависит от его взаимодействия с последующими преобразованиями. Например, удаление дублирующих операций может привести к нехватке регистров при конечном планировании. По умолчанию преобразования кода производятся компилятором с использованием оценок, сформированных статистическим образом на некотором тестовом наборе задач. Такая настройка является в среднем оптимальной, но может быть неэффективной для конкретной задачи. Основные причины двойственности производимого настраиваемыми оптимизациями эффекта при статической компиляции следующие:

1) Агрессивное дублирование участков кода, расщепления схождений. К соответствующим преобразованиям относятся подстановка процедур в места вызовов (inline), выделение регионов и расщепление боковых вхождений для последующего слияния кода (regions), различные преобразования расщепления циклов (loop split by index, loop split by condition, loop unswitching), преобразования построения гнезда из цикла (nesting), раскрутка циклов (unroll).

   “+” : Расщепление схождений может позволить точно вычислить результат части операций, избавиться от участков кода для части

путей исполнения, подготовить горячий участок для формирования гиперблока без редко исполняемой длинной части, а также обеспечить возможность конвейеризации цикла. Возникающие при дублировании операции иногда могут быть конвейеризованы.

“-” : При агрессивном дублировании значительно вырастает объем кода, что может замедлять исполнение из-за возникновения блокировок по коду; может увеличиваться нагрузка на регистры и появляться необходимости построения spill и fill операций для временного хранения данных регистров в памяти; иногда происходит дублирование исполнения части операций.

2) Агрессивное слияние кода и сопутствующее повышение степени его спекулятивности, которое происходит при программной конвейеризации циклов (softpipe, overlap), при объединении нескольких узлов в гиперблоки (if_conversion).

“+” : Слияние кода и спекулятивное исполнение кода позволяет скоращать критический путь исполнения, при конвейеризации уменьшается физическая итерация цикла, то есть среднее планируемое время исполнения его одной итерации.

“-” : При избыточном слиянии кода ограничивающим планирование фактором может стать нагрузка на ресурсы, что может привести к увеличению спланированного времени исполнения. При ошибках предсказания наиболее вреоятных путей исполнения может увеличиваться длина критического пути горячего участка. Возможно исполнение лишних операций чтения, что приведет к дополнительной нагрузке на память и последующим блокировкам конвейера из-за ожидания результата других чтений.

3) Подкачка кода или данных (-flist-prefetch, -fcache-opt).

“+” : Предварительная подкачка кода или данных, а также заброс операций чтений значительно выше операции использования их

результата, может позволить уменьшить количество блокировок конвейера в случае, если соответствующий код или данные не находятся в кэш-памяти 1-ого уровня.

“-” : Требующиеся дополнительные операции могут увеличивать спланированное время исполнения кода; подкаченные заранее данные могут не потребоваться, а нагрузка на кэш-память при этом увеличится, что может привести к блокировкам в других точках кода.

4) Настройка распределения ресуров.

Указанная задача возникает при проведении преобразования выноса инвариантов из цикла (invup), удаления дублирующих операций, при конвейеризации циклов.

“+”“-” : Необходимо соблюдение баланса между уменьшением планируемого времени исполнения кода и увеличением количества требуемых регистров. На промежуточных этапах оптимизации зачастую сложно или невозможно оценить возникающую нагрузку на регистры. В результате при конечном планировании могут потребоваться spill и fill операции, которые переносят данные с регистров в память и обратно. Особенно сильно указанная проблема проявляется в том случае, когда архитектура процессора предоставляет сравнительно небольшое число регистров.

Для регулирования наиболее значимых настроек условно полезных преобразований разработаны опции, которые могут быть поданы компилятору. Набор опций для повышения эффективности оптимизации конкретной задачи может быть получен с помощью ручной настройки. Однако, процесс такой настройки затратен по времени, поскольку требуется провести подробный анализ всех горячих, то есть часто исполняющихся, участков кода. При этом обычно пользователь не обладает в полной мере информацией о работе всех

оптимизаций, что ограничивает его возможности по анализу с точки зрения оптимизации. Дополнительная сложность заключается в наличии эффекта *интерференции*, то есть совокупного результата применения опций, пример которого приведен на Рисунке 1. На рисунке отображены результаты отдельного применения двух опций, и их комбинации (both), показывающие на ряде задач противоположный эффект. Проблема интерференции затрудняет последовательный подбор опций или применение методов внешней настройки, таких как Ortogonal Arrays [35], хотя и может быть легко преодолена при настройке оптимизации внутри компилятора или при предварительном создании наборов опций.

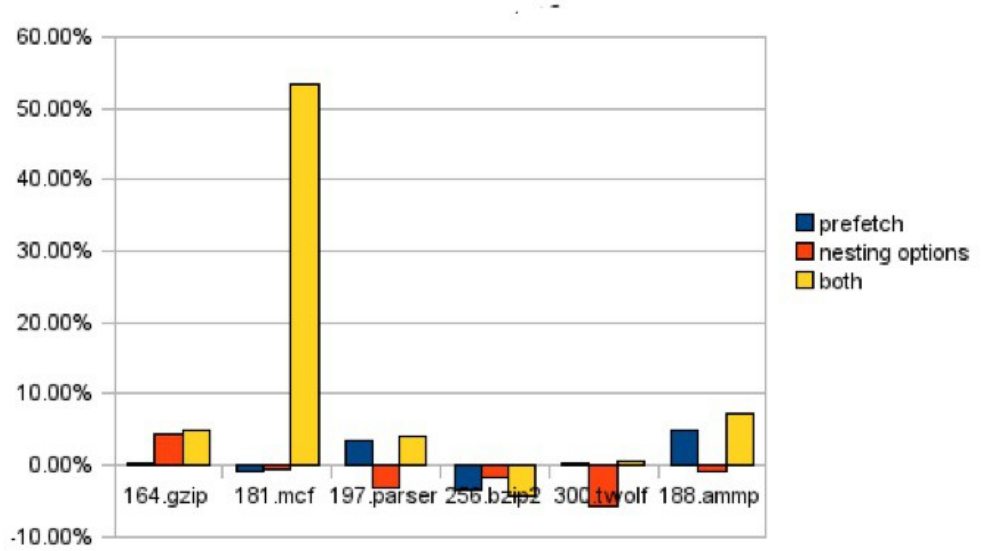

Рисунок 1: Интерференция оптимизаций

1.4 Однопроходный механизм автоматической попроцедурной настройки компиляции

В последние годы активно ведутся исследования в области автоматизации поиска эффективной последовательности компиляции. Кроме отсутствия обозначенных ранее недостатков ручного подбора опций, существенным

преимуществом такого подхода признается возможность выбирать не общие настройки оптимизации для всей задачи, а наиболее подходящие для каждой процедуры.

Самыми известными методами автоматической настройки набора оптимизационных опций являются итеративные решатели, которые можно разделить на две основные категории по способу выбора варианта компиляции:

1) Итеративная компиляция с последующим исполнением [14].

В этом случае производится перебор оптимизационных последовательностей или опций из некоторого заранее сформированного набора [15] [16] [17]. После каждого варианта компиляции приложение исполняется на тренировочных данных с замером времени. Для ускорения такого поиска используется последовательный подбор наилучшего набора настроек компиляции по заранее сформированному на тренировочном пакете приложений дереву опций, подбор опций только для самых горячих процедур. Подобные системы достаточно эффективны, но их существенный недостаток заключается в требовании многократного исполнения на представительных данных. Из-за этого ограничения метод подходит только для небольших приложений.

2) Итеративная компиляция с оценкой времени исполнения по спланированному коду.

Метод аналогичен предыдущему, но в этом случае не производится многократное исполнение кода. Вместо этого время исполнения оценивается по спланированному ассемблеру с учетом счетчиков исполнения тренировочного запуска и особенностей архитектуры. Такой подход к настройке был использован в [20] для пакета простых тестов. В [21] описан способ оценки эффекта применения небольшой последовательности цикловых оптимизаций по неполному результату компиляции. Похожее решение с заранее настроенным деревом опций и более точной оценкой времени исполнения, получил название OSE (Optimization-Space Exploration) и был применен к более значительному набору оптимизаций в компиляторе для архитектуры Итаниум

(Itanium) уже для полноценного пакета приложений spec2000 [22]. Помимо оценки времени планирования, в нем также осуществляется попытка оценить нагрузку на кэш-память. Чтобы результаты оценок времени исполнения были в достаточной мере достоверными для целой последовательности преобразований и для уменьшения числа настраиваемых процедур до приемлемого значения с точки зрения времени компиляции, OSE требует осуществить однократное исполнение с получением профиля приложения. В компиляторе под архитектуру Эльбрус метод итеративной компиляции применяется при вычислении фактора раскрутки преобразования unroll для обеспечения последующей векторизации [36][37]. Как и в [21], для подсчета времени исполнения осуществляется не полная последовательность компиляции с планированием, а оценивается работа только нескольких наиболее значимых фаз, но даже такая настройка составляется значительную часть времени компиляции для многих приложений. Поскольку результат применения указанной оптимизации меньше других зависит от профиля исполнения, то такой метод дает неплохие результаты и в базовом режиме.

То есть подход с итеративной компиляцией без проверочного исполнения доступен для использования на большем числе приложений и для отдельных оптимизаций применим в промышленных компиляторах в режиме по умолчанию. Тем не менее, для настройки нескольких разноплановых оптимизаций одновременно он все равно требует представительного по профилю исполнения кода и значительно увеличивает время компиляции. Вдобавок, отсутствует возможность качественно оценить работу памяти и возникновение блокировок конвейера из-за промахов в кэш. Стоит отметить, что при оптимизации кода под VLIW-архитектуру задача ограничения времени компиляции особенно важна. Из-за сложности используемых алгоритмов и необходимости полного статического планирования расход времени при максимальном уровне оптимизации (-O3) для ряда задач становится неприемлемо большим при статической компиляции и недопустимым – в

динамическом варианте (JIT-компиляция), когда время компиляции фактически добавляется к времени исполнения. При этом некоторая часть применяемых преобразований дублирует ту или иную часть кода, чтобы в дальнейшем можно было перемешать более ранние и более поздние операции. Это часто позволяет плотней спланировать код, но сопутствующее увеличение кода значительно увеличивает время работы всех последующих этапов компиляции и иногда само по себе приводит к замедлению исполнения из-за возникающих блокировок по чтению кода. Такой эффект возникает, к примеру, на задаче 176.gcc пакета CINT spec2000. То есть требуется найти набор опций, не только обеспечивающих быстрый итоговый код, но и ограничить преобразования некоторым оптимальным набором, который позволит получить близкую к максимальной скорость исполнения, но не будет отрабатывать впустую и не увеличит чрезмерно код. Для компилятора gcc было проведено исследование, показывающее теоретический потенциал получения ускорения компиляции в два раза без деградаций по скорости исполнения итогового кода [17]. Однако, в этом случае использовалось фактически неограниченное количество оптимизационных последовательностей, поскольку для каждой процедуры строилась своя, самая короткая эффективная последовательность.

Методы итерационной компиляции позволяют существенно повысить производительность кода, при этом они позволяют проводить настройку компиляции отдельных процедур, входящих в состав приложений. Однако, как было указано, все они требуют значительного времени компиляции и тренировочного исполнения приложений на представительных данных. В результате они практически не применимы для больших приложений.

В качестве альтернативного метода настройки компиляции интересной представляется задача машинного обучения с целью выбора оптимальных настроек компиляции по вычислимым характеристикам компилируемого кода.

Подобный подход был реализован в Milepost GCC для настройки компиляции приложений [38] . В этом решении оценивается вероятность того, что приложение с некоторым значениям характеристик соответствует некоторому оптимальному набору настроек.

Подход настройки приложений целиком интересен для небольших вычислительных задач с однотипным составом с точки зрения процедур. Однако для больших приложений интересна также возможность попроцедурной настройки, которая может позволить по-разному работать с небольшими вычислительными процедурами и большими процедурами, занимающимися управлением.

В представленной работе исследован вопрос машинного обучения с целью попроцедурного выбора оптимизационной последовательности.

Схема предлагаемого метода состоит в следующем:

1. Разрабатывается набор эффективных оптимизационных последовательностей и в дальнейшем используется как составная часть механизма. Последовательности могут отличаться наличием, последовательностью и настройками оптимизирующих преобразований, технических и аналитических фаз компиляции.

Набор последовательностей разрабатывается на некотором тренировочном статистически значимом пакете задач, то есть на взвешенном пакете, содержащем задачи с разными типами вычислений, с разной работой с памятью и другими свойствами приложений. Эффективность сформированного набора последовательностей для тренировочного пакета задач может определяться минимальным достижимым суммарным временем исполнения всего пакета или же другой совокупностью числовых характеристик результата компиляции, таких как время исполнения, время компиляции, размер кода и т.п., достижимой на пакете при условии оптимального выбора последовательности для каждой процедуры. С целью обеспечения числовой оценки эффективности предлагается использовать *метрику*, определенную на множестве

всевозможных назначений последовательностей.

2. В процессе компиляции после проведения межпроцедурных преобразований вычисляются характеристики каждой процедуры (Рисунок 2).

Под характеристиками процедуры подразумеваются числовые свойства кода, доступные в процессе компиляции, такие как среднее число итераций циклов, число операций в процедуре, максимальная глубина вложенности циклов, процент арифметических операций с плавающей точкой, процент операций чтения адреса, число вызовов других процедур, среднее предполагаемое время исполнения процедуры (вес процедуры) и другие. Кроме того, в качестве характеристик могут быть использованы относительные характеристики рассматриваемой процедуры по сравнению с другими доступными процедурами. В случае режима «вся программа» это все процедуры приложения, иначе — процедуры компилируемого модуля (файла).

3. По вычисленным характеристикам выбирается оптимизационная последовательность для каждой процедуры из предварительно разработанного набора (Рисунок 2).

После чего методами машинного обучения реализуется классификация обучающего пакета процедур по характеристикам, позволяющая добиться хорошего выбора при назначении каждому классу некоторой общей последовательности из набора. Метод числовой оценки качества выбора будет описан далее.

4. В процессе компиляции задачи после межпроцедурной фазы оптимизации по характеристикам процедуры определяется соответствующий ей класс. Дальнейшая, то есть попроцедурная компиляция, осуществляется в соответствии с назначенной классу процедуры оптимизационной последовательностью.

Предлагаемый механизм далее будет называться *система направленной оптимизации процедур,* или сокращенно СНОП.

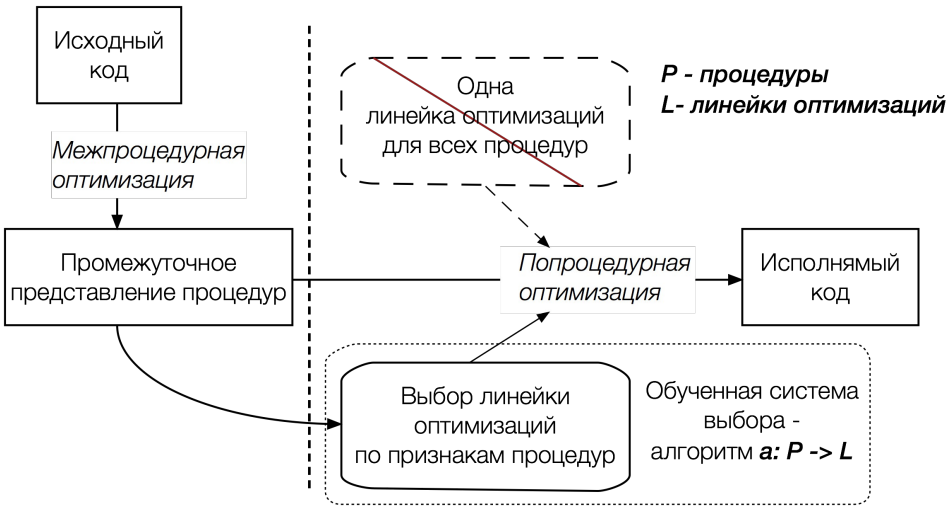

Рисунок 2: Схема применения механизма СНОП

1.5 Разработка набора последовательностей оптимизаций и настроек

Для всех вариантов автоматической настройки компиляции, включая предлагаемый, требуется определить ограниченный набор рассматриваемых или допустимых комбинаций опций и последовательностей оптимизаций.

С целью создания пригодных для реального, промышленного использования итеративных систем, были исследованы различные подходы к ограничению допустимых конфигураций, то есть всего набора опций и настроек. Значительная часть работ по итеративной компиляции ограничивается совсем небольшим числом настраиваемых оптимизаций, которые затем строит их в того или иного рода связки [17]. В большинстве случаев выбор опций осуществляется в ручную на основе имеющейся информации о преобразованиях. Подобный подход к выбору опций был применен и при построение итеративного решателя для процессора Intel Itanium [22].

На этом фоне интересной представляется работа по предварительному созданию эффективного ограниченного покрывающего набора оптимальных настроек для некоторого множества задач [39]. В указанной работе

рассматривается большое количество настраиваемых оптимизаций на пакете небольших тестов. Часть наборов опций заранее отбраковывается по тем или иным соображениям, для оставшихся 300 производится прогон на тестовом пакете, после чего выбираются наиболее покрывающие 10. Такой подход представляется весьма интересным с точки зрения аналитического создания наборов оптимизаций, однако, даже в таком варианте для обучающей выборки из больших задач он не применим. Основной его недостаток — это невозможность учитывать работу с памятью. К сожалению, многие ситуации, приводящие к блокировкам конвейера, не воспроизводятся на коротких тестах. В то же время, блокировки конвейера занимают значительную часть времени исполнения реальных приложениях. Так, проведенные на машине (Эльбрус-2С+ -кубик Эльбрус-4С e2s 14-ый год) замеры событий при исполнении пакета приложений CINT spec2000 выявили, что блокировки конвейера при чтениях (в пиковом режиме?) из-за отсутствия данных в кэш-памяти занимают в среднем 17%, достигая 74% на приложении 300.twolf. То есть эффективность работы с памятью является крайне важным фактором, который требуется учитывать при сравнении результатов компиляции кода, поэтому более предпочтительным является обучение на больших задачах, исполняющихся на полных данных. Кроме реалистичной работы с памятью, использование полных задач представляется более взвешенным с точки зрения различных типов процедур, что необходимо при обучении априорному выбору последовательности. Так, на вычислительной задаче процедура инициализации будет исполняться меньше по времени, чем процедура, осуществляющая собственно вычисления, что позволит сделать вывод о соответствующей значимости оптимизации их планирования. Если же рассматривать эти процедуры независимо, то эта информация будет утеряна. С другой стороны, использование в качестве тестовой базы больших задач значительно ограничивает количество допустимых для замера набора опций.

Расчет на возможность статического выбора эффективной

последовательности оптимизаций связан с предположением о существовании неявной типизации процедур с точки зрения требований к оптимизации, различимой по исходному коду. Естественно предположить, что похожая типизация характерна и для горячих процедур внутри одного приложения, поэтому эффективные настройки для всего приложения будут достаточно эффективны и для настройки значительной части его горячих процедур. Для разработки оптимизационных наборов были изучены и проанализированы ранее подобранные пиковые настройки пакетов приложений CINT spec2000 и CFP spec2000. Анализ показал неэффективность ряда значимых оптимизаций в режиме компиляции без профиля. Для них были разработаны альтернативные преобразования и настройки. Проведенный анализ и предложенные методы описаны в **Главе 2**. Эффект от разработанных и ранее использованных опций и режимов был замерен на всех приложениях пакетов. На основе полученной таблицы были выделены наборы, позволяющие в совокупности получить значительную часть эффекта по производительности.

В результате в качестве первого набора опций была выбрана использующаяся компилятором последовательность для базового уровня оптимизации (режима -O3). Второй выбранный набор был ранее разработан для режима оптимизации O2. Основной целью его построения было создание оптимизационной последовательности, позволяющей быстро (на момент создания она позволяла почти в 3 раза быстрей осуществить компиляцию, чем последовательность для режима O3) получить достаточно эффективный код на задачах с целыми вычислениями при условиях ограничения роста кода и спекулятивности. Полученная последовательность, несмотря на значительные ограничения по многим оптимизациям, на некоторых задачах была успешно использована для получения пиковой производительности. Еще два набора опций были сформированы на основе набора оптимизаций режима -O3. По своей сути один из них позволяет качественно спланировать исполнение многоитерационных циклов с большой нагрузкой на память, а другой – циклов с

большим количеством ветвлений.

В большинстве случаев в итеративных решателях используются наборы подаваемых опций, регулирующие применение отдельных оптимизаций. Для удобной настройки не только набора, но и последовательности применяемых преобразований, а также аналитических этапов, в компиляторе для архитектуры Эльбрус был реализован механизм внешней передачи компилятору полной последовательности производимых компилятором действий, то есть оптимизирующих *линеек*. Этот механизм является внутренним, то есть не предполагается его использование пользователем, тем не менее, он позволяет достаточно легко создавать и обновлять оптимизирующие последовательности, что значимо для использования при разработке и настройке компилятора. Набранные наборы опций были сформированы в оптимизационные линейки, после чего линейки были дополнительно скорректированы за счет удаления более неэффективных этапов компиляции в каждом случае.

Список использованных для настройки оптимизаций и аналитических этапов приведен в Таблице 3. При этом оптимизации в итоговых линейках могут быть по необходимости выключены, включены или на них изменены настройки применения. Кроме того, для разных линеек может отличаться последовательность осуществления преобразований.

Реализованный механизм выбора набора опций является менее аналитичным, чем предложенный в [39], поскольку рассматривается меньшее количество опций и программ. Однако, за счет использования результатов проведенной ранее работы по подбору пиковых опций для разнородных приложений, он позволил за умеренное время получить эффективные оптимизационные наборы, учитывающие проблему работы с памятью, проблему блокировок по чтению кода и другие особенности, связанные с исполнением больших приложений.

Оптимизирующие преобразования	cache optimization * loop unswitching group loop unswitching split by index split be condition split by inequality loop undercondition loop undercase vector canonization vector idiom vector invariant removing memory access widening additional scalar replacement loop lazy code motion redndant loop load elimination redundant loop store elimination softpipe interchange reroll reverse global copy propagation move optimization altexpel nesting vectorization maw peeling unroll unroll_fuse altexpel regions+if conversion (superblocking) srtmd dam guess prefetch list prefetch prefetch * short path unroll *
Аналитичекие этапы	loop dependence analysis forks dependence analysis value numbering nodes sizes unsigned to signed

Таблица 3. Преобразования, технические и аналитические этапы, использованные при построении набора оптимизационных линеек. Оптимизации, помеченные (*), были разработаны и настроены для использования в базовом режиме в рамках данной работы. Их описание приведено в Главе 2.

1.6 Проблема вероятностного подхода при обучении компилятора

При попытке использовать вероятностный подход для задачи выбора оптимизационной линейки для процедуры, предложенный в Milepost для приложений [38], автором была выявлена проблема, связанная с несоответствием вероятностной ошибки реальному ожиданию от обучения компилятора.

Так, обучение компиляции сетями Байеса с целью повышения производительности и менее приоритетному ускорению компиляции (то есть выбору быстрой компиляции при одинаковом времени исполнения) позволило с вероятностью 95% предсказать лучшую линейку, но при этом привело к среднему замедлению приложений на 21%. Анализ указанного замедления показал следующее:

1) ошибка выбора линейки для процедуры зависит от того, какая именно не оптимальная линейка была выбрана;

2) ошибка выбора линейки для одной и тоже же процедуры зависит содержащего ее приложения.

Тезис: Указанные особенности связаны с тем, что в действительности при обучении компилятора **требуется повысить не вероятность выбора оптимальных настроек, а среднюю производительность пакета приложений**. При этом производительность чаще всего вычисляется как некоторая функция от времен исполнения приложений тестового пакета $TestPack = \{App_i\}$, к примеру, как их среднее геометрическое:

$$F(TestPack) = \sqrt[K]{\prod_K \sum_i exe(App_i)}$$

В **Главе 3** будет подробно рассмотрен вопрос построения функции, позволяющей оценить качество обучения компилятора. Для требуемого учета

нескольких критериев компиляции будет решена задача построения многокритериальной оценки качества компиляции.

В **Главе 4** будет сформулирована и решена задача машинного обучения, соответствующая минимизации построенного функционала качества компиляции.

Глава 2. Оптимизация в режиме без профиля

Как уже упоминалось в Главе 1, самым эффективным режимом, доступным для компиляции в большинстве случаев, являются помодульный режим -O3 без использования профиля. Однако, поиск опций для применения к приложениям в этом режиме выявил, что часть значимых преобразований, реализованных в компиляторе для архитектуры Эльбрус, настроена к работе или применима только для случая использования тренировочного профиля. В результате без их адаптации к однофазному варианту не удается добиться значительного увеличения производительности кода. Поиск альтернатив в научной и технической литературе не привел к успеху. Поэтому была поставлена задача выявления главных причин отставания результирующего кода при использовании статического профиля и разработки альтернативных методов повышения производительности для наиболее страдающих контекстов.

Анализ различий в показателях производительности полученного при двухфазной и однофазной компиляции кода приведен в разделе 2.1 Разделы 2.2, 2.3, 2.4 посвящены разработанным альтернативным методам для применения в случае отсутствия тренировочного профиля, - 2.2 описан метод раскрутки коротких путей, позволяющий лучше спланировать некоторые участки цикла, в 2.3 предлагается метод предварительной подкачки данных для уменьшения числа блокировок по нерегулярным адресам, в 2.4 описан способ настройки оптимизации работы с кэш-памятью для базового использования. В 2.5 приведены экспериментальные результаты применения предложенных методов в компиляторе под Эльбрус.

В разделе 2.6 описан способ сокращения средней длины рекуррентности конвейеризуемых циклов, сходный с методом 2.2 и применимый как для однофазной, так и для двухфазной компиляции.

2.1 Проблемы производительности кода при однофазной компиляции

Сравнительная производительность кодов, полученных в результате базовой однофазной и двухфазной компиляций, была исследована на примере тестового пакета spec2000.

Исполнение задач с большим числом вычислений с плавающей запятой, то есть пакета CFP, в режиме по умолчанию -O3 практически не дало разницы по времени. В этом случае за счет анализа структуры кода и типа исполняющихся операций удается достаточно качественно с точки зрения использования основными оптимизациями предсказать число итераций цикла. При этом показания счетчиков, собирающих число исполнения узлов внутри ациклических участков, для этого типа задач достаточно равномерны, то есть отсутствует код с большим количеством маловероятных ветвлений. Единственное существенное отличие возникло при попытке применения дополнительной оптимизации работы с кэш-памятью, о которой будет написано в 3.4.

Основные же проблемы однофазной компиляции выявились на задачах с большим числом ветвлений, значительная часть которых редко исполняется. Такая структура кода характерна для вычислений с целыми, представленных пакетом CINT. Это компиляторы, базы данных, интерпретаторы скриптовых языков, задачи сжатия и дискретные задачи, такие как искусственный интеллект для игры в шахматы или расчет алгебраических моделей. Проблема предсказания маловероятных ветвлений обусловлена тем, что ветвления, характеризующиеся одинаковой локальной структурой графа потока управления и сходным набором операций, на разных задачах могут иметь

существенно отличающиеся вероятности. Это приводит к тому, что при построении предсказанного профиля на ветвлениях они получаются усредненными, и, соответственно, маловероятные ветвления редко удается угадать.

Наиболее значительные отличия в работе оптимизаций, полученные на проанализированных задачах при однофазной и двухфазной компиляции из-за неточного предсказания вероятного и маловероятного кода, свелись к следующему:

1) Различным образом происходит оптимизация подстановки процедур.

Оптимизация подстановки процедур (inline) производит дублирование кода вызываемой процедуры и подстановку ее в точку вызова, что позволяет перемешивать исполнение операций вызывающей и вызываемой процедуры и удалять часть избыточных операций. Отличия ее работы при двухфазной и однофазной компиляции связаны с возможностью в первом случае использовать более точное значение счетчика вызова для оценки изменения времени исполнения [29], и, при меньшем увеличении размера кода, обеспечить больший прирост производительности. В однофазном режиме компилятор не может ориентироваться на наличие основного пути исполнения задачи, поэтому решения о подстановке кода принимаются в результате попыток улучшить планирование рассматриваемых процедур. При этом подстановка некоторых часто исполняемых участков не выполняется.

2) Не удается устранить из циклов неисполняющийся код с последующей *программной конвейеризацией циклов* (software pipelining) [40][41][42] оставшихся операций.

Программная конвейеризация циклов позволяет спланировать одновременное исполнение операций с разных итераций, существенно повышая эффективность использования исполняющих устройств при статическом

планировании. Фактически она производит многократный перенос наборов операций цикла по обратной дуге для их более раннего исполнения, в результате чего сокращается критический путь, то есть минимальное число тактов исполнения итерации с среднем, или длина физической итерации. Однако, есть несколько факторов, ограничивающих возможности такого преобразования. Так, известные механизмы конвейеризации не применимы при наличии внутренних циклов или нескольких обратных дуг. Кроме того, наиболее эффективные методы конвейеризации, позволяющие избежать исполнения одновременно нескольких вариантов одной и той же исходной операции, требуют отсутствия вызовов и ветвлений в цикле. В случае наличия вызовов такие методы не применяются, а при наличии ветвлений и нескольких выходов производится слияние кода в гиперузел, в котором планируется исполнение всех операций цикла, и сведение выходов в один. Особенностью описанных преобразований является удлинение критического пути исполнения до максимального из всех исходных путей выполнения цикла. Дополнительным ограничением с точки зрения времени спланированного исполнения одной итерации цикла при конвейеризации является длина его максимальной рекуррентности. Для отдельных контекстов исследуются другие возможности перемешивания операций. Так, в ряде работ исследуется возможность одновременной конвейеризации внешнего и внутреннего цикла [43][44], а также возможность более эффективной конвейеризации нескольких малоитерационных циклов за счет перемешивания пролога и эпилога соседних циклов [45]. Описанные методы повышают эффективность исполнения циклов с равномерными счетчиками числа исполнения узлов, но аналогичная программная конвейеризация циклов с большим количеством редко исполняемого кода приведет к повышению параллельности по планированию операций, при этом незначительно увеличит реальную параллельность кода.

В случае, если участки, содержащие операции самой длинной рекуррентности, циклы, вызовы или длинные по критическому времени

выполнения пути являются маловероятными, то может применяться преобразование гнездования (nesting) [46], которое формирует внутренний цикл из горячих участков, а маловероятный выносит во внешний цикл с построением переходов. Такое преобразование дает наибольший эффект в том случае, когда горячий участок имеет не только значительно больший счетчик, но и при его меньшем весе с точки зрения критического пути или количества операций. То есть при *разновесных ветвлениях в цикле,* или несбалансированности ветвлений цикла с точки зрения их времени исполнения и счетчиков числа исполнения. Гнездование также может обеспечить возможность применения конвейеризации в случае циклов или вызовов в холодных участках. Однако, для положительного эффекта от его применения требуется, чтобы переходы на участки цикла, содержащие перечисленные ограничивающие факторы для конвейеризации, были маловероятными, поскольку выделение операций в разные циклы не позволит спланировать их одновременное спекулятивное исполнение. А для выявления указанной малой вероятности нужно иметь достоверную информацию о профиле исполнения задачи.

3) Возникают сложности с планированием одновременного исполнения операций соседних веток.

Для сокращения длин критических путей ациклических участков производится объединение в один гиперблок операций с соседних веток, то есть узлов, в которые ведут дуги с общим началом. Исполнение части операций при этом планируется безусловно, а часть ставится под предикат перехода на соответствующую дугу. Такое объединение операций эффективно до тех пор, пока требуемое количество ресурсов для планирования операций в одном такте не превышает аппаратные возможности. Оптимизация построения гиперблоков (if_conversion) выбирает для объединения наиболее вероятные узлы, а в последующем, на фазе финального планирования, операции упорядочиваются в соответствии со значениями счетчиков числа их исполнения [47]. При наличии

большого количества маловероятных участков и отсутствии профильной информации в один гиперблок попадают операции с горячих и холодных веток, и значительная часть спланированных для одновременного исполнения операций не используется. То есть, даже при плотном планировании операций понижается реальная параллельность кода.

4) Возникает большее число блокировок конвейера при чтениях.

Причина заключается в перемешивании используемых и неиспользуемых в дальнейшем чтений и в отсутствии возможности в ряде случаев дополнительно увеличить время от часто выполняемых чтений до использования без значительного суммарного увеличения планируемого времени исполнения. В случае же двухфазной компиляции часто и редко используемые чтения, как правило, разделяются, что уже само по себе уменьшает нагрузку на кэш-память, поскольку данные, используемые в маловероятных участках, не вытесняют другие.

Возможным частичным решением этой проблемы может быть предварительная программная подкачка данных (software prefetch), которая представляет собой построение дополнительных операций чтения для переноса данных по вычисленным или вероятным адресам в кэш-память. В работе [48] описаны существующие методы программной подкачки данных. К сожалению, в довольно частом случае нерегулярных чтений единственным доступным методом предварительной подкачки является использование встроенных функций __builtin_prefetch, то есть явное добавление в код дополнительных операций пользователем с вычислением и указанием адреса подкачки.

В соответствии с результатами классификации основых отличий оптимизации при двухфазной и однофазной компиляцией были проведены исследования по поиску методов уменьшения отставания производительности во втором случае.

Чтобы оценить возможность улучшения производительности кода за счет более точной оптимизации подстановки сначала была собрана профильная информация на тренировочном запуске, которая затем использовалась компилятором до фазы подстановки включительно. Далее, имеющаяся профильная информация процедур была заменена построенной статически, и дальнейшая оптимизация проводилась с использованием статического профиля. Сравнительный запуск полученной сборки и сборки при статическом профиле в среднем выявил только незначительное отличие, обусловленное тем, что отсутствие информации о маловероятных ветвлениях, особенно оптимизаций выделения регионов и одновременного планирования нескольких узлов ациклических участков (проблема 3), не позволяют достаточно эффективно использовать преимущества более качественного проведения подстановок.

Вторая и четвертая проблемы довольно независимы, особенно в случае, когда не достигнута максимально возможная плотность спланированных команд. В разделе 3.2 предлагаются механизмы компенсации проблемы 2 при однофазной компиляции: для увеличения параллельности вычислений в циклах с разновесными ветвлениями и с неравномерными по счетчикам исполнения узлами автором разработан механизм частичной раскрутки путей исполнения. Что касается блокировок по чтению, то их количество удалось сократить за счет применения разработанной оптимизации подкачки данных для части нерегулярных чтений, имеющих пространственную локализацию. Этому посвящен раздел 3.3.

Проблема планирования ациклических участков с большим количеством маловероятных ветвлений в рамках оптимизаций частично уменьшается за счет уменьшения размера создаваемых суперблоков по сравнению с двухфазной компиляцией с тем, чтобы ошибочное предположение о распределении вероятностей переходов меньше увеличивало длину вероятных путей. Дальнейшее ускорение производительности для этого случая видится возможным только за счет дополнительного анализа при построении

предсказанного профиля.

2.2 Метод раскрутки коротких путей цикла

Основная идея метода раскрутки коротких путей (short path unroll) заключается в том, чтобы выделить поднабор коротких путей возможного исполнения итерации цикла и спланировать его дополнительные итерации для исполнения одновременно с одной тяжелой по времени исполнения итерацией исходного цикла. При этом в случае достаточно высокой вероятности прохода по выделенным путям существует возможность ускоренного исполнения цикла. Описанный далее механизм применим к циклам с любым количеством обратных дуг, а также вложенных циклов и вызовов. В соответствии с далее описываемым алгоритмом для циклов, у каждого из которых имеется один вход и один узел вхождения обратных дуг (голова цикла), выделяется часть путей от головы цикла до обратных дуг. В случае несколько входов и/или несколько голов выделяются пути от каждой головы.

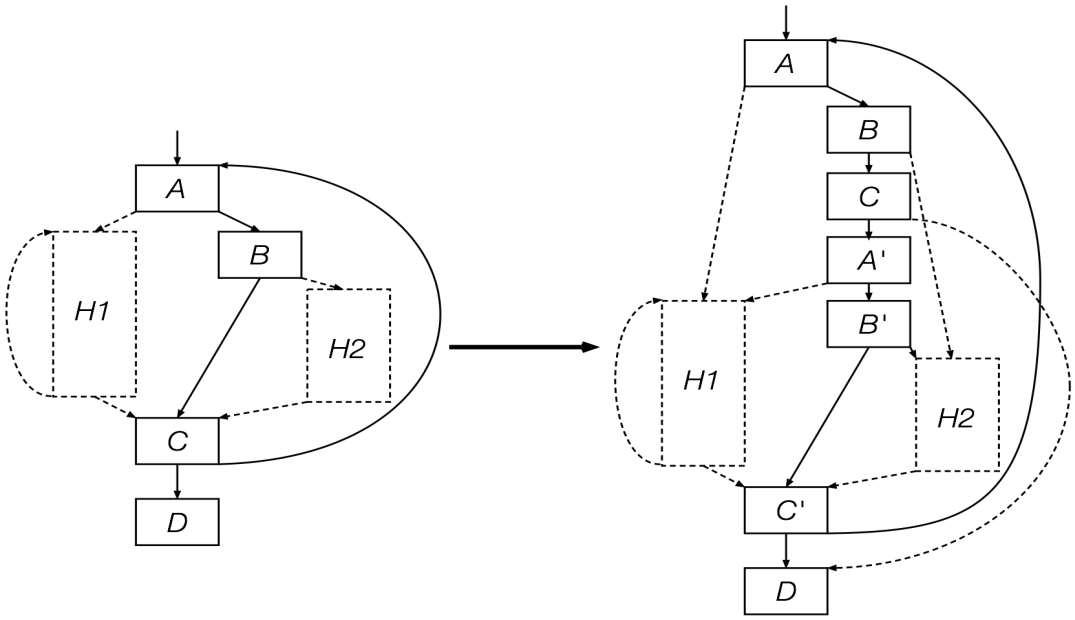

Рисунок 3: Схема применения преобразования раскрутки коротких путей цикла

Обозначим раскручиваемую часть цикла U, а нераскручиваемую часть,

состоящую из оставшихся узлов цикла, то есть остаток, - R. Раскручиваемая часть копируется в теле цикла один или более раз согласно эвристической оценке, кроме того, для каждой копии строятся дополнительные дуги перехода в нераскручиваемую часть. Число созданных копий будем называть фактором частичной раскрутки *k*. Это преобразование и рассматривается как частичная раскрутка цикла: на Рисунке 3 выбраны части исходного цикла

$U = \{ \text{узлы } A, B, C, \text{дуги } A \to B, B \to C, C \to A \}$,

$R = \{ \text{участки } H1, H2 \}$

и построена дополнительная итерация раскручиваемой части с переходами в остаток

$U' = \{ \text{узлы } A', B', C', \text{дуги } A' \to B', B' \to C', C' \to A', A' \to H_1, B' \to H_2 \}$, а так же переход из частичной итерации U на итерацию U', соответствующий обратной дуге $C \to A$ исходного цикла.

Предлагаемое преобразование является более безопасным с точки зрения планирования по сравнению с комбинацией nesting и программной конвейеризации, поскольку позволяет планировать одновременное исполнение раскручиваемого участка с остатком и не требует дополнительного времени на разгон внутреннего цикла. При этом, в отличие от методов программной конвейеризации, это преобразовании применимо к циклам любой сложности.

Для выделенной раскручиваемой части можно аналитически вычислить пользу с точки зрения планирования от раскрутки в том случае, если точно известен профиль исполнения кода, а также известно, какие операции будут удалены в дальнейшем.

В случае однофазной компиляции нет возможности полагаться на значения счетчика, поэтому предлагается более грубая оценка, позволяющая выделить незначительные по времени исполнения пути, замедление при дублировании которых будет незначительным даже в случае их небольшой

реальной вероятности.

Алгоритм выбора раскручиваемых путей и применения преобразования для их дублирования при однофазной компиляции:

1) Рассматриваются циклы с оценочным по статическому профилю числом итераций не менее 2, которые не подходят для программной конвейеризации.
2) Из состава рассматриваемых узлов цикла удаляются все узлы вложенных циклов, вызовы, узлы, которые нельзя добавлять в суперблоки для планирования совместного исполнения с другими узлами, а также все участки, у которых после такого удаления не остается путей до обратных дуг. Оставшийся стартовый участок алгоритма обозначим U.
3) Для каждого узла в U подсчитывается количество узлов xmin минимального содержащего его пути в графе управления, который проходит через голову цикла и какую-нибудь из обратных дуг, и максимальная длина пути цикла smax. Находится минимальное (smin) значение из всего набора xmin.
4) Если в исходном цикле были вызовы или циклы (большое число исполняемых операций по сравнению с U), то для решения о применении раскрутки используется U, переход к 5. Иначе, если разница между smin и smax существенная, то есть, smin <= smax/k, где k>=2 - настраиваемый параметр, то из U выкидываются все узлы, для которых xmin > smax/k.

Таким образом, в цикле выделяются те пути, длина которых при раскрутке на k дополнительных итераций будет сравнима с оставшимися нераскручиваемыми участками, что сделает цикл более сбалансированным.

5) Если размер выделенного региона по числу узлов меньше некоторого значения k, то полученный участок U принимается в качестве раскручиваемого набора путей. Ограничение на размер раскручиваемого участка использовано с целью избежать излишнего разрастания кода и уменьшить возможное увеличение критического пути из-за схождений на

переходе между раскрученными итерациями, и в то же время получить достаточный эффект от возможностей параллельного планирования соседних итераций. Наиболее эффективное ограничение будет зависеть от архитектурных особенностей вычислительной машины и используемого пакета для настройки из-за разного доступного количества вычислительных устройств в такте и среднего количества операций в узлах. Для архитектуры «Эльбрус» наиболее эффективным оказалось ограничение длины пути значением 5.

6) В случае результативного выделения раскручиваемого участка U создается его копия - дублируются узлы и внутренние дуги U в соответствии с количеством голов цикла, и строятся необходимые дополнительные дуги. При этом обратные дуги самого региона направляются на головы копии, а обратные дуги копии – на те же узлы U, куда ведут соответствующие дуги основного региона. Пример перенаправления дуг и построения дополнительных дуг и узлов приведен на рисунке 1. При значительной несбалансированности исходного ациклического участка, то есть smin<smax/k, где k>=3, для части путей может быть оправдано построение большего числа – до k копий.

Замечание:

При применении частичной раскрутки возможно увеличение времени планирования одной итерации нового цикла, если результат операций раскручиваемого региона используется операциями остатка следующей итерации. То есть, если в обозначениях Рисунка 3 операции из $R = H_1 \wedge H_2$ используют результат операций A, B, C. Однако, при небольшой вероятности перехода в остаток, суммарное время планирования цикла все равно уменьшится.

Попробуем оценить сравнить время исполнения полученного после преобразования цикла с временем исполнения исходного цикла.

Теорема 1:

Пусть среднее число итераций исходного цикла равно I_L, полная вероятность пройти от стартового узла до узла с его единственной обратной дугой, не посетив узлы остатка, равна $p(U)$. При этом вероятности всех ветвлений цикла независимы. Тогда среднее число итераций цикла, полученного путем частичной раскрутки с фактором *k* равно

$$I_{L(U,k)}=\frac{I_L}{1+\sum_{i=1...k} d_p(L,U)^i},\ d_p(L,U)=p(U)\cdot(1-\frac{1}{I_L})$$

Доказательство:

Поскольку вероятности всех ветвлений независимы, то вероятность выхода из исходного цикла не зависит от номера итерации и равна $p_b=\frac{1}{I_L}$ [49], и такую же вероятность имеют все ветвления, которые являются парами к переходам на текущую голову цикла и все копии головы исходного цикла. Вероятность дойти от i-ой копии до i+1 копии равна $p(U)*(1-p_b)$, так как для этого надо сначала не выйти из i-ой копии U, а затем не выйти из цикла. То есть вероятность дойти до головы i-ой копии раскручиваемой части равна:

$$p_{Ui}=(p(U)\cdot(1-p_b))^i$$

Далее, пусть мы уже дошли до i-ой копии, тогда вероятность выйти из цикла и не дойти до i+1-ой копии складывается из вероятности попасть в остаток и выйти и вероятности не попасть в остаток и выйти:

$$b_i=(1-p(U))\cdot p_b+p(U)\cdot p_b=p_b$$

Получаем, что полная вероятность выхода из созданного цикла равна

$p_b'=p_b+\sum_{i=1...k} p_b\cdot p_{Ui}=p_b\cdot(1+\sum_{i=1...k} p(U)\cdot(1-p_b)^i)$, а

$$I_{L(U,k)}=\frac{1}{p_b}'=\frac{1}{p_b\cdot(1+\sum_{i=1...k}(p(U)\cdot(1-p_b))^i)}=\frac{I_L}{1+\sum_{i=1...k}(p(U)\cdot(1-\frac{1}{I_L}))^i}$$

Следствие:

В условиях доказанной теоремы верны следующие оценки для $I_{L(U,k)}$:

1) $\lim\limits_{k\to\infty} I_{L(U,k)} = \lim\limits_{k\to\infty} \dfrac{I_L}{1+\sum\limits_{i=1\ldots k} \left(p(U)\cdot\left(1-\dfrac{1}{I_L}\right)\right)^i} = I_L - p(U)\cdot(I_L-1)$

2) Если $I_L \geq 2$, то $I_{L(U,1)} \leq \dfrac{I_L}{1+\dfrac{1}{2}p(U)}$

3) При большом значении $I_L \gg 1$:

$$I_{L(U,k)} = \frac{I_L}{1+\sum\limits_{i=1\ldots k} \left(p(U)\cdot\left(1-\dfrac{1}{I_L}\right)\right)^i} \approx \frac{I_L}{1+\sum\limits_{i=1\ldots k} p(U)^i},$$

в частности, $I_{L(U,1)} \approx \dfrac{I_L}{1+p(U)}$

Теорема 2:

Пусть критический путь выделенного участка $cr(U)$, а критический путь всего цикла cr_L, тогда после частичной раскрутки U с фактором k критический путь нового цикла $cr_L' = cr_L(U,k) \leq cr_L + k * cr(U)$.

Доказательство:

Для доказательства достаточно заметить, что проведенное преобразование увеличивает длину любого пути цикла не более, чем на $k * cr(U)$ тактов.

Следствие:

Пусть планирование циклов ограничивается длиной критического пути, то есть ресурсное ограничение не достигается. Тогда в условиях доказанных утверждений получаем оценку на отношение времени исполнения циклов:

$$\frac{t_L}{t_L(U,k)} = \frac{cr_L \cdot I_L}{cr_L(U,k)\cdot I_{L(U,k)}} = \frac{\left(1+\sum\limits_{i=1\ldots k} \left(p(U)\cdot\left(1-\dfrac{1}{I_L}\right)\right)^i\right)\cdot cr_L}{cr_L(U,k)} \quad (1)$$

$$\frac{t_L}{t_L(U,k)} \geq \frac{1+\sum_{i=1\ldots k}\left(p(U)\cdot\left(1-\frac{1}{I_L}\right)\right)^i}{1+\frac{k\cdot cr(U)}{cr_L}} \quad (2)$$

а при достаточно больших $I_L \geq 2$ можно еще больше упростить вычисления

$$\frac{t_L}{t_L(U,k)} \approx \frac{1+\sum_{i=1\ldots k} p(U)^i}{1+\frac{k\cdot cr(U)}{cr_L}} \quad (3)$$

Следствие: При наличии информации о профиле исполнения, использованные в алгоритме раскрутки коротких путей оценки 3), 4), 5), 6) могут быть уточнены с помощью вычисления отношений (1), (2) или (3). При этом (1) даст более точную оценку, но потребует значительно больше вычислений для расчета.

2.3 Уменьшение количества блокировок по чтениям в базовом режиме

Как уже упоминалось в Главе 1, блокировки конвейера при ожидании результатов чтения занимают значительную часть времени исполнения приложений, а для некоторых приложений - основную часть времени исполнения, и особенно остро эта проблема проявляется в случае однофазной компиляции. Поэтому работа по их предотвращению не менее важна, чем уменьшение спланированного времени исполнения кода.

При планировании операции использования чтения обычно компилятор использует базовое значение времени исполнения чтения процессором. Однако, соответствующее время является минимально возможным и будет реализовано только в случае наличия читаемых данных в кэш-памяти первого уровня. Если требуемых данных там не окажется, то запрос отправится дальше, в кэш-память второго уровня или же в оперативную память. При этом для процессоров Эльбрус время исполнения самого короткого и самого долгого запроса составит 3 и 84 такта соответственно. Поэтому, если потребитель результата чтения спланирован ранее 84-ех тактов, то есть вероятность возникновения

блокировки, при которой конвейер будет простаивать. Для борьбы с описанной проблемой используются два основных метода:

1) Техники планирования кода, при которых производится заброс чтений на большое расстояние от использования.

2) Построение дополнительных операций, которые заранее подкачают какие-то участки данных в кэш-память, или prefetch.

По умолчанию заброс операций чтения ограничивается дистанцией, при которой он не приводит к удлинению критического пути или рекуррентности. Однако, при программной конвейеризации возможен метод агрессивного заброса чтений, при котором дополнительно увеличивается критический путь исполнения отдельно взятой итерации. В этом случае спланированное увеличение времени исполнения при этом связано с удлинением пролога цикла, а время планирования конвейеризованной итерации остается прежним за большего числа стадий. Не всегда удается осуществить такой заброс из-за увеличивающейся нагрузки на регистры, при этом он не применим, если чтение стоит на ограничивающей время планирования рекуррентности. Кроме того, при таком методе возникает дополнительная нагрузка на кэш-память из-за лишних чтений на последних итерациях цикла, отсутствующих в первоначальном случае, что часто приводит к блокировкам по чтению на тех программах, на которых они отсутствовали при конвейеризации без такой техники. Поэтому реализованный механизм агрессивного заброса чтения в компиляторах обычно включается по опции и даже при этом работает только на циклах с заведомо большим числом итераций, то есть обычно при наличии информации о профиле исполнения программы. В разделе 2.3.2 будет предложен и обоснован способ оценки расстояния для заброса, который позволяет применять его по умолчанию.

Построение дополнительных операций для предварительной подкачки данных может осуществляться вручную с помощью специальных функций или

компилятором. Чтобы указать область памяти для подкачки необходимо вычислить адрес, иногда при этом используются различные проверки для адресов. В разных реализация это могут быть точно вычисленные адреса будущих чтений или некие участки памяти, обращение к которым представляется достаточно вероятным. В любом случае, при подкачке может происходить перенос в кэш-память данных, которые в последствии не будет использованы с вытеснением полезных данных, поэтому она может сама по себе провоцировать блокировки. Вдобавок, работа по вычислению адресов подкачки, в ряде случаев достаточна объемная [50], может приводить к значительному увеличению планируемого времени исполнения кода. Поэтому, как и в случае агрессивного заброса чтений, техники предварительной подкачки данных обычно применяются по дополнительным опциям. В разделе 2.3.1 будет предложен метод подкачки данных, который за счет достаточно серьезных ограничений на подкачиваемые области памяти и компилируемый код дает в среднем прирост производительности кода в режиме при применении по умолчанию, а потому может быть использован в базовом режиме.

2.3.1 Подкачка данных для нерегулярных чтений в неконвейеризуемых циклах

В компиляторе для Эльбруса поддержан архитектурный механизм APB (Array Prefetch Buffer) и реализован программный механизм подкачки по вычислимым заранее адресам, в совокупности они позволяют избавиться от значительной части блокировок при регулярных чтениях. Поэтому блокировки в основном возникают при нерегулярных чтениях. При двухфазной компиляции есть возможность оценить частоту промахов по памяти, выделить проблемные чтения и для самых горячих участков с многократным исполнением сформировать дополнительный анализирующий код для настройки подкачки в процессе исполнения [50]. Поскольку в случае однофазной компиляции гарантированно выделить небольшой горячий участок

невозможно, а применение подобной подкачки к большому количеству чтений приводит к значительному замедлению исполнения в среднем, то необходим простой метод подкачки небольшого количества часто блокируемых типов чтений с минимальными вычислительными затратами.

Основные блокировки при нерегулярных чтениях в циклах проанализированных задач были связаны с чтением полей агрегатных объектов, то есть struct в C, или полей объектов в C++. При анализе адресов таких чтений и структуры циклов было замечено следующее:

1) Ввиду того, что структура в памяти и поля объектов представляют из себя выделенный участок, в котором последовательно хранятся данные, то при работе с ними часто происходит обращение к нескольким близко расположенным в памяти данным. В таких случаях появляется возможность избежать блокировки для нескольких чтений за счет только одной подкачки, и как следствие контролировать переполнение кэш-памяти и уменьшить число дополнительных операций.
2) Смещение по памяти между адресами чтений соседних итераций часто производится на одну и ту же величину, даже если при этом не является постоянным для цикла. Этот факт дает возможность в ряде случаев осуществить эффективную подкачку из памяти при условии вычисления текущего смещения.

Кроме того, следует учесть, что:

3) Применять подкачку имеет смысл только к тем циклам, для которых не работает техника с забросом чтений при конвейеризации.
4) Желательно не увеличивать время исполнения циклов операциями подкачки данных. В этом случае при ошибочном применении подкачки вред будет минимален.

В результате был сформирован следующий список компиляторных

ограничений для чтений, к которым применяется оптимизация предварительной подкачки по вероятному адресу:

- операнд чтения получен в результате другого чтения.
- операция чтения находится в цикле (внутренним или внешнем), который не будет конвейеризирован.
- чтение находится на критическом пути, то есть, нет возможности без увеличения времени планирования исполнить его дальше от использования, чтобы избежать блокировки в случае промаха.
- по адресу чтения производится k>=2 чтений в цикле с точностью до некоторого константного смещения.
- после добавления операций подкачки спланированное время исполнения узла не увеличивается; при компиляции для архитектуры «Эльбрус» упрощенный вариант этого ограничения означает, что число операций в узле не должно стать больше, чем 6*crit, где crit - критический путь узла.

Для таких чтений вычисляется вероятный адрес следующего чтения с помощью подсчитанного смещения относительно предыдущего адреса и строится операция подкачки по нему (см Таблицу 4).

Стоит отметить, что введенные серьезные ограничения на применение предлагаемой подкачки данных хоть и сужают контекст, но позволили производить ее в режиме по умолчанию, поскольку приводят к повышению производительности в среднем.

Исходный код	Результирующий код
load ... -> Vs1 load Vs1 Const1 -> load Vs1 Const2 ->	load ... -> Vs1 sub Vs1 Vs2 -> Vs3 mov Vs1 -> Vs2 load Vs1 Vs3 -> empty load Vs1 Const 1 -> load Vs1 Const 2 ->

Таблица 4. Построение кода подкачки.
Обозначения: load – операция чтения из памяти; Vs1, Vs2, Vs3 – виртуальные регистры; Const1, Const2 – константные аргументы операций; empty – результат чтения не записывается в регистр, то есть фактически подкачивается в кэш

2.3.2 Вычисление оптимальной дистанции заброса чтений при конвейеризации

Исходно реализованный в компиляторе для Эльбруса вариант увеличения дистанции между операциями чтения и использующими их результат операциями, работающий по опции *-fcache-opt,* предусматривал заброс на максимально возможную, не превышающую максимальное время чтения из памяти, дистанцию, при которой все еще хватает регистров для планирования. При этом, для уменьшения контекста и ограничения на рост пролога, дополнительно были установлены серьезные ограничения снизу на число итераций. Снятие этих ограничений привело к значительному замедлению большинства задач, что вызвало необходимость коррекции метода.

Для того, чтобы позволить применять оптимизацию работы с кэш-памятью без опции в однофазном режиме компиляции, предлагается ввести ограничение на расстояние откладывания использования, которое соответствует увеличению количества тактов по сравнению с планированием по умолчанию не больше, чем в *par* раз, где *par* – регулируемый параметр.

Программная конвейеризация цикла характеризуется следующими значениями (Рисунок 3):

I - число итераций

II - длина физической итерации, то есть количеством тактов, в которые спланированы все операции цикла по одному разу с разных итераций, оно же среднее число тактов на итерацию без учета пролога

S - число стадий – количество итераций, операции с которых спланированы для одновременного исполнения

По этим значения планирумое число тактов исполнения цикла вычисляется по формуле:

$$Ticks = II \cdot (I + S - 1)$$

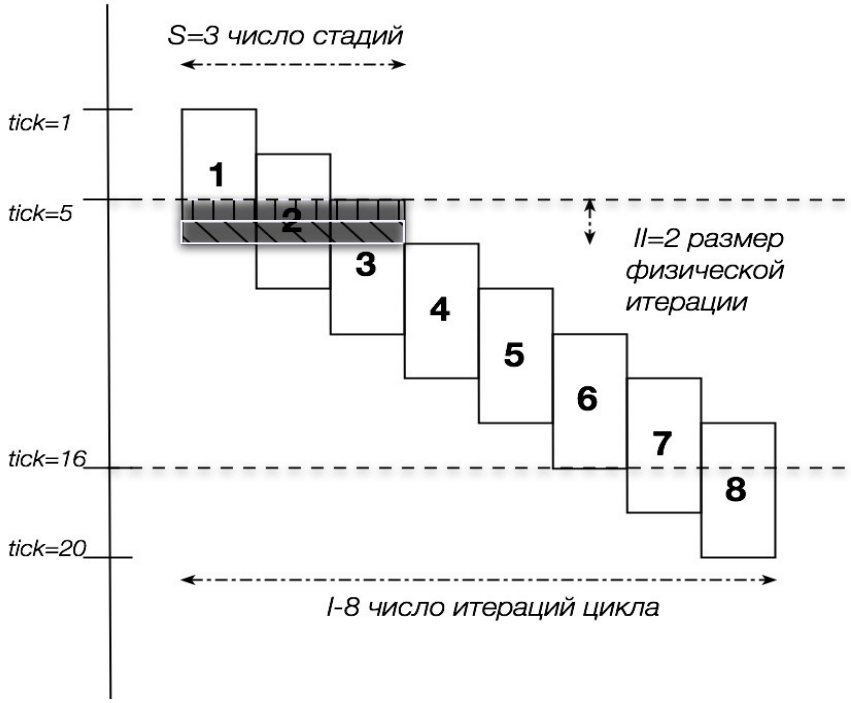

Рисунок 4. Конвейеризация цикла. Число итераций I=8, число стадий S=3, размер физической итерации II=2. Планируемое время исполнения $Ticks = II \cdot (I + S - 1) = 20$

Вычислим максимально допустимое число стадий, при котором число тактов ticks увеличивается в некоторое par раз, то есть:

$$Ticks_{new} = II \cdot (I + S_{new} - 1) \le II \cdot (I + S - 1) \cdot par$$

Это требование равносильно:

$$S_{new} \le (I - 1) \cdot (par - 1) + S \cdot par$$

То есть дистаниция для заброса должна быть не больше, чем максимальное время чтения *LongestLoad*, и из ограничения на S_{new} она должна быть не больше, чем $S_{new} \cdot II - (time_{back} - time_{load})$ - допустимое время логической итерации минус время от чтения до последнего такта цикла, то есть до обратной дуги. Это минимальное ограничение на дистанцию заброса, но оно не достаточное. Сложность состоит в том, что до полного планирования с распределением регистров невозможно определить, с каким числом стадий удастся спланировать цикл. Поэтому предлагается сначала вычислить исходное значение S и, начиная с вычисленной дистанции заброса чтения, уменьшать ее до момента, пока число стадий при планировании цикла не станет равным или меньшим, чем вычисленное S_{new} .

Алгоритм 1: Регулируемый заброс чтений.

1: Произвести конвейеризацию цикла без увеличения дистанции для чтений, пусть при этом получается число стадий *S*.

2: Вычислить $S_{new} = (I - 1) \cdot (par - 1) + S \cdot par$

3: **Если** *Snew* получился больше хотя бы на 1, чем *S*, **то** :

4: Вычислить максимальное (стартовое) расстояние заброса load-а по следующей формуле :

$$length < min\left(LongestLoad\ ,\ S_{new} \cdot II - (time_{back} - time_{load})\right)$$

где $LongestLoad$ — максимальная задержка в случае промаха

$time_{back}$ - исходное время планирования обратной дуги

$time_{load}$ - исходное время планирования load

5: Осуществить планирование конвейеризованного цикла.

6: **Если** число стадий S получилось больше S_{new} или не хватило регистров, **то** *length* уменьшается на 1, повторить 5.

Параметр par позволяет регулировать агрессивность заброса чтений с точки зрения увеличения планируемого времени исполнения цикла. Эвристически подобранное наиболее эффективное в среднем на пакете spec2000 значение параметра $par = 1.6$.

2.4 Экспериментальные результаты

1) Применение short path unroll.

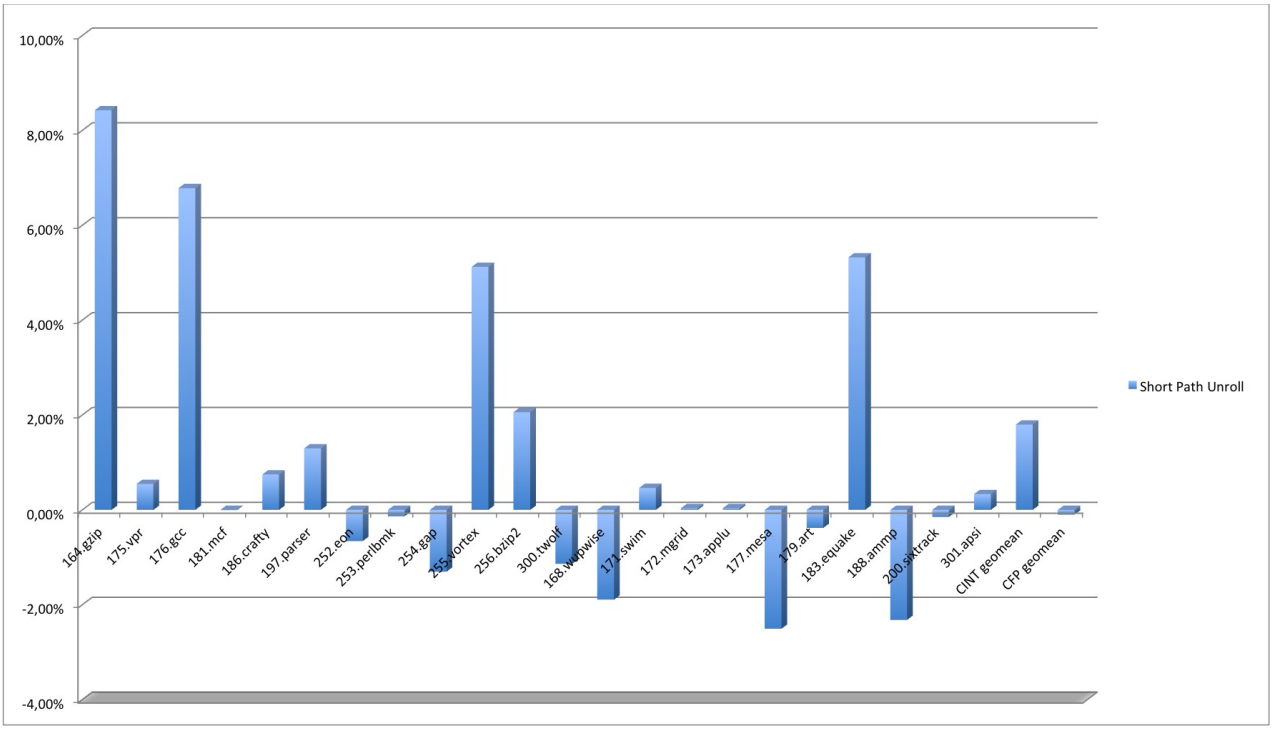

Рисунок 5. Раскрутка коротких путей, spec2000 benchmark
CINT - среднее для задач с целыми вычислениями
CFP – среднее для задач с плавающими вычислениями

На Рисунке 5 иллюстрируется эффект применения метода раскрутки коротких путей на пакете benchmark spec2000 в режиме –O3. В среднем было

получено ускорение на 1,85% задач с целыми вычислениями и на 0,2% задач с плавающими вычислениями. На задачах 164.gzip (+8,4%) и 256.bzip2 (+2,1%) прирост производительности был достигнут за счет оптимизации гнезд циклов с небольшим средним числом итераций как внешнего, так и внутреннего цикла, для задачи 255.vortex (+5,1%) было улучшено планирование горячего внутреннего цикла с тяжелым по числу узлов маловероятным ветвлением и наличием наиболее вероятного короткого пути исполнения, а задача 176.gcc (+6,8%) показала ускорение на многих процедурах с разным устройством циклов. Стоит отметить, что эффект связан не только с улучшенным планированием, но и с избавлением от ряда операций дополнительной итерации, к примеру, от чтений, результат которых может меняться только в длинном участке, то есть, на короткой итерации осуществляется аналог оптимизации выноса инвариантов из цикла.

Замедления на задачах с плавающими вычислениями связаны с тем, что при запуске этих задач исполняются, как правило, самые длинные участки кода и все внутренние циклы, а потому оптимизация коротких путей приводит к небольшому замедлению за счет увеличения критического пути. Таких деградаций можно избежать при проведении некоторого анализа структуры кода для выявления типа задачи.

2) Применение -fprefetch к нерегулярным чтениям объектов.

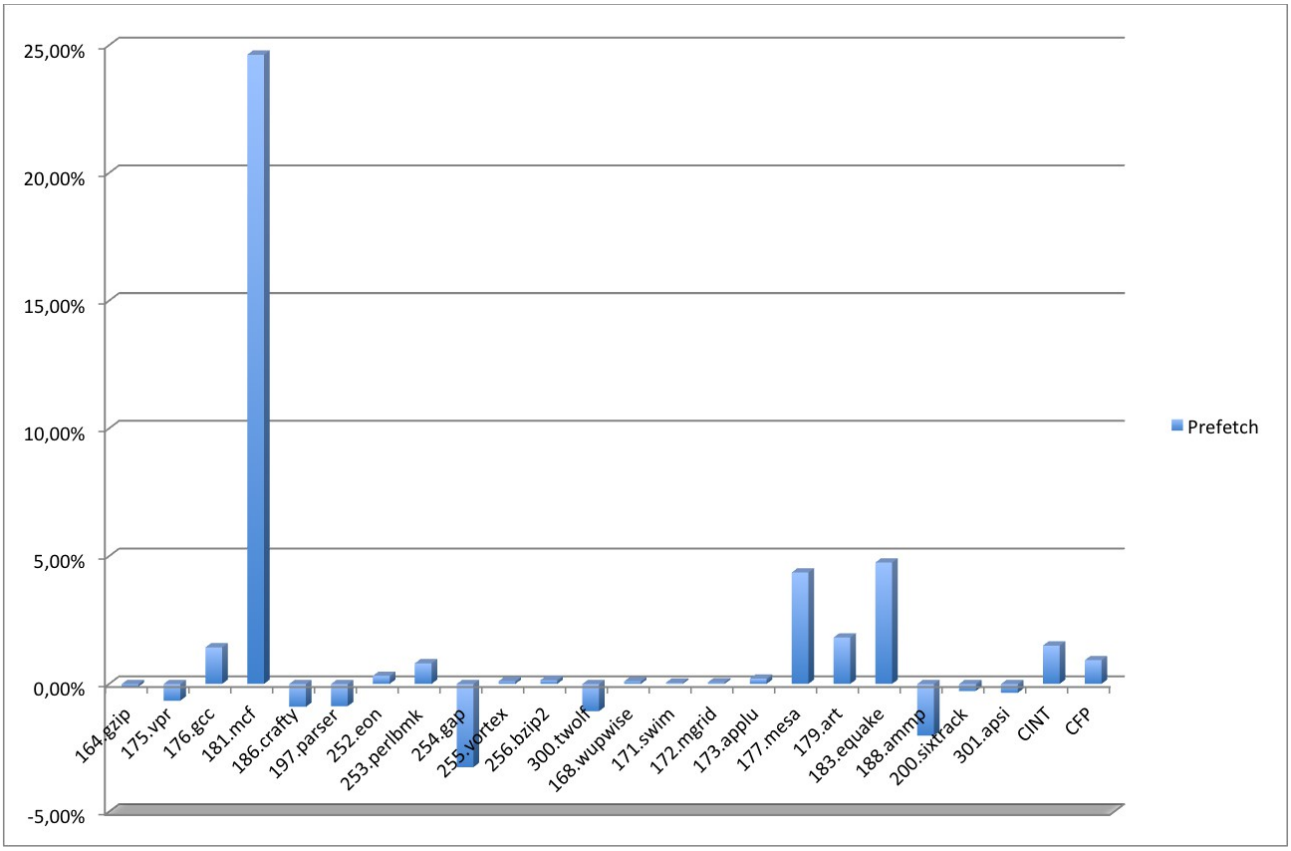

Рисунок 6: Применение подкачки данных для нерегулярных чтений на пакете spec2000.

Эффект от применения предлагаемого метода, замеренный в режиме компиляции –O3 на пакете spec2000, показан на Рисунке 6. За счет осуществленной подкачки значительно сократилось число блокировок по чтениям на задаче 181.mcf, что привело к ускорению ее исполнения на 24,6%. Кроме того, уменьшилось количество блокировок в ряде внешних циклов задач с плавающими вычислениями используемого тестового пакета. Так, на 4,4% и 4,8% ускорились задачи 177.mesa и 183.equake. Единственное значительное замедление на задаче 254.gap (-3,3%) произошло на процедуре, к которой оптимизация не применялась, оно связано с увеличением давления на кэш-память из-за применения подкачки к соседней процедуре, что в рамках попроцедурной оптимизации сложно устранить. В среднем было получено ускорение на 1,5% задач с целыми вычислениями и ускорение на 0,92% задач с плавающими вычислениями.

3) Применение -fcache-opt с ограничениями для базового режима.

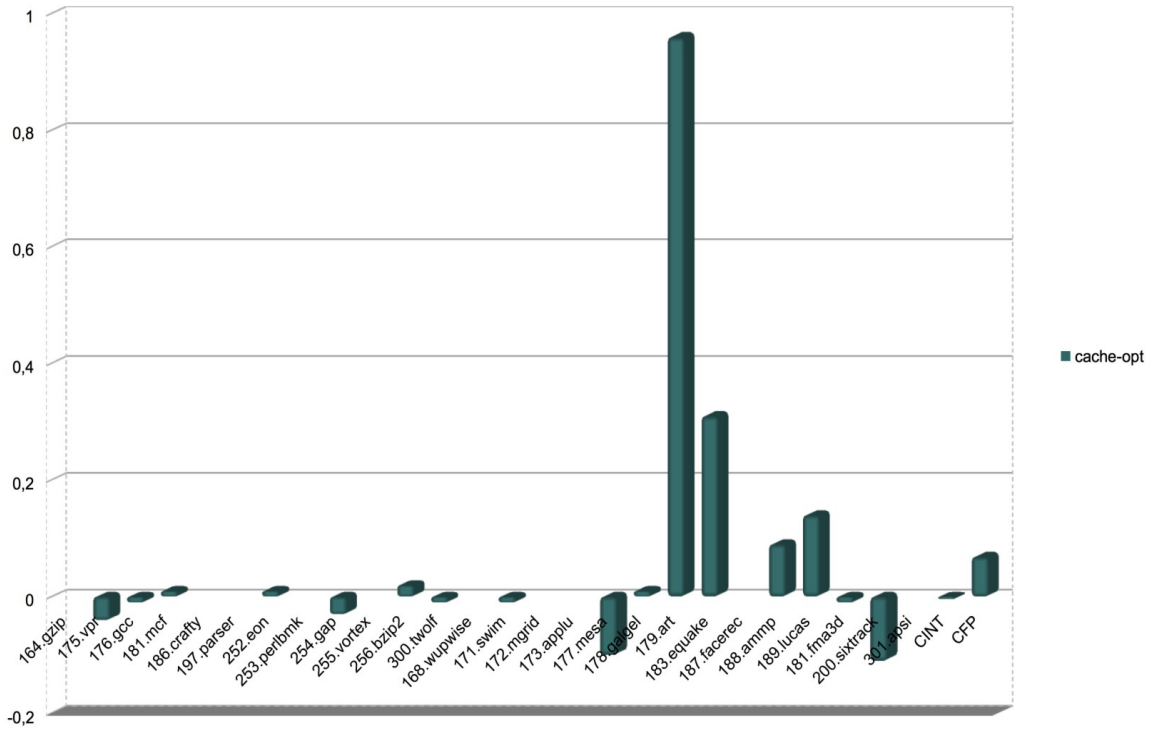

Рисунок 7: Применение опитмизации работы с кэш-памятью в режиме без профиля по умолчанию.

Применение метода увеличения дистанции с предложенными ограничениями в среднем приводит к ускорению ряда задач пакета spec2000. На Рисунке 7 видно, что основные ускорения произошли на задачах пакета CFP 179.art и 183.equake. Произошло небольшое замедление нескольких задач из пакета с целыми вычислениями CINT, а также примерно на 10% увеличилось время исполнения задач 177.mesa и 200.sixtrack из пакета CFP. Основные замедления связаны с небольшим количеством итераций целевых циклов при реальном исполнении. Однако, использование вычисленных ограничений позволило значительно сократить потери производительности в этом случае. В результате среднее замедление задач с целыми вычислениями составило 0,2%, при этом задачи с плавающими вычислениями ускорились в среднем на 6%.

2.5 Метод частичной раскрутки рекуррентностей цикла

Метод раскрутки коротких путей позволяет сбалансировать сложные циклы, удлинив короткие пути, что оказывается эффективным при оптимизации циклов, для которых неприменима программная конвейеризация. Однако, можно предложить похожий метод для применения не только к критическим путям неконвейеризуемых циклов, но и к циклам с конвейеризацией для балансировки рекуррентностей.

Под рекуррентностями цикла подразумеваются цепочки операций, связанные прямыми зависимостями, то есть каждая следующая операция ожидает начала или окончания выполнения предыдущей. Длиной рекуррентности называют минимальное число тактов, которое требуется для выполнения ее операций без учета других операций цикла. Эффект от предлагаемого далее *метода раскрутки коротких рекуррентностей* может быть получен в случае наличия достаточно вероятных длинных рекуррентностей и наличии вероятных коротких рекуррентностей, проход по которым регулируется условием. Как и в предыдущем случае, наиболее точно эффективность применения метода можно оценить при с использованием тренировочной профильной информации, но он может быть эффективен и при ее отсутствии.

Предлагаемый *метод раскрутки коротких рекуррентностей цикла* (short recurrency unroll) заключается в том, чтобы:

1) Выделить стоящие под условием пути цикла, которые содержат только короткие по времени исполнения рекуррентности цикла, с достаточно высокой суммарной вероятностью выполнения.
2) Применить к выбранным путям преобразование частичной раскрутки рекуррентностей цикла, то есть планировать их копии, соответствующие дополнительным (последующим) итерациям, для исполнения одновременно с одной итерацией остальной части цикла.

В отличие от метода раскрутки коротких путей цикла, основная область

применения предлагаемого преобразования – это циклы, подлежащие в дальнейшем программной конвейеризации.

Преобразование частичной раскрутки рекуррентностей цикла состоит в следующем:

Пусть выбран набор рекуррентностей цикла. Выделим минимальный набор путей цикла, который содержит все операции соответствующих рекуррентностей. Обозначим через R выделенный набор путей цикла, то есть раскручиваемую часть цикла, а нераскручиваемую часть, состоящую из оставшихся узлов цикла, - S. Набор путей R копируются в теле цикла один или более раз согласно вычисленной оценке эффективности, кроме того, для каждой копии строятся дополнительные копии переходов в нераскручиваемую часть S, а переходы из S внутрь цикла переносятся в соответствующие узлы последней копии. Это преобразование рассматривается как *частичная раскрутка рекуррентностей*.

Замечание: Фактически при этом производится дублирование не только операций, стоящих на рекуррентностях, но и других операций соответствующих узлов.

Пример: На Рисунке 8 приведен пример применения метода раскрутки коротких рекуррентностей. В цикле выделены операции короткой и длинной рекуррентности

$oper1 \to oper2 \to oper3 \to oper1$ и

$oper1 \to oper3 \to oper4 \to oper5 \to oper6 \to oper7 \to oper8 \to oper9 \to oper3 \to oper1$

Выбран путь, состоящий из узлов и дуг исходного цикла, который содержит только короткую рекуррентность $U = \{ узлы\ A, B, C, дуги\ A \to B, B \to C \}$, остаток $R = \{ D \}$ и построена дополнительная итерация раскручиваемой части

$U' = \{$ *узлы* A', B', C', *дуги* $A' \to B', B' \to C', C' \to A, A' \to D\}$, а дуга $D \to C$ заменена на дугу $D \to C'$.

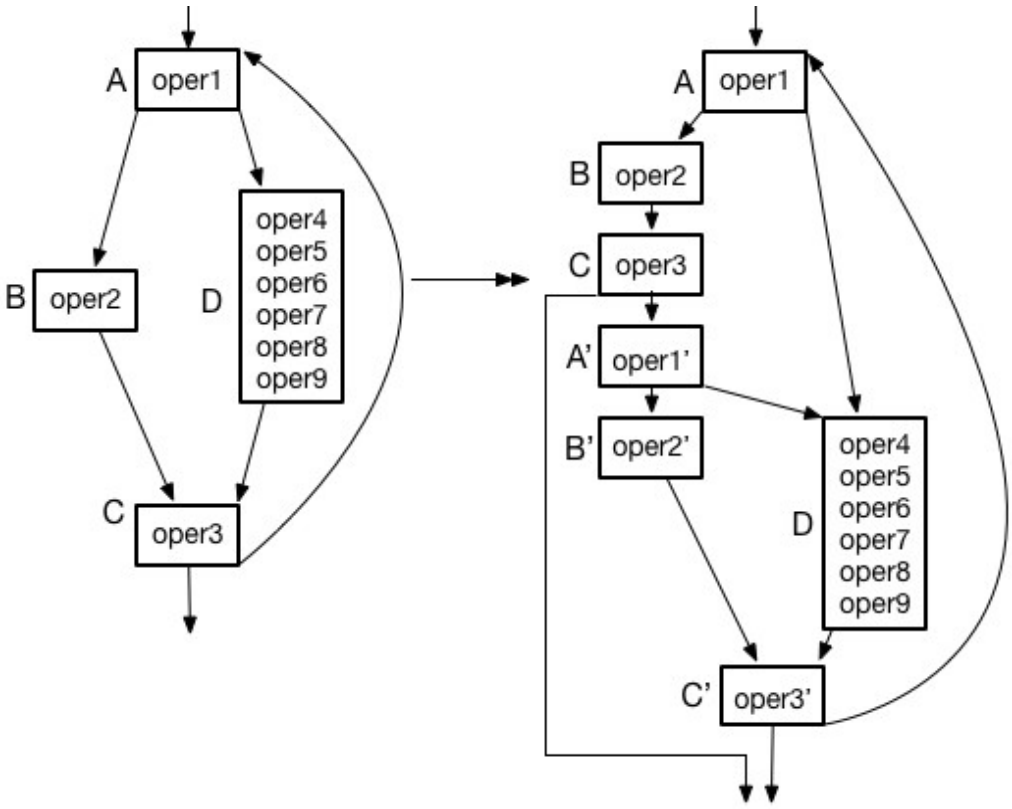

Рисунок 8. Частичная раскрутка рекуррентностей цикла.

Качество выбранного набора путей с точки зрения применения предлагаемого преобразования можно оценить с помощью приближенного вычисления изменения времени исполнения цикла, исходя из максимальной длины рекуррентности результирующего цикла, изменения среднего числа итераций цикла и нагрузки на исполнительные устройства.. Докажем ряд утверждений, позволяющих это сделать.

Чтобы соотнести изменение длины рекуррентности и числа итераций цикла, нужно подсчитать совокупную вероятность пройти по набранным путям ($prob(rec_U)$), длину в тактах самой длинной рекуррентности для набранных путей $len(rec_U)$ и длину самой длинной рекуррентности в цикле $len(rec_L)$.

Замечание: В цикле могут присутствовать рекуррентные цепочки операций, которые замыкаются, то есть приходят к исходной операции, только через несколько итераций. В этом случае для рекурентности rec_S длиной в j тактов, которая замыкается через q итераций длина рекуррентности вычисляется как:

$$rec_S = \frac{j}{q}$$

Теорема:

Пусть длина в тактах самой длинной рекуррентности для набранных путей rec_U, длина самой длинной рекуррентности в цикле rec_L, тогда длина рекуррентности цикла после частичной раскрутки U с фактором k, и формированием цикла $L(U,k)$:

$$rec_L(U,k) \le k * rec_U + rec_L$$

Замечание:

Оценка среднего числа итераций результирующего цикла не отличается от оценки числа итераций цикла в Теореме 1 для случя раскрутки коротких путей цикла и вычисляется через $prob(rec_U)$ и число итераций исходного цикла I_L.

Замечание:

При конвейеризации цикла, кроме самой длинной рекуррентности, важное значение приобретает и нагрузка на вычислительные ресурсы, поэтому при оценке минимального количества тактов, в которое может быть спланирован цикл, используется соответствующее значение tr_L.

Следствие:

Оценка ускорения времени исполнения цикла при частичной раскрутке выбранных рекуррентностей цикла может быть произведена по формулам:

$$\frac{t_L}{t_L(U,k)} = \frac{min\left(tr_L, \left(1 + \sum_{i=1\ldots k} \left(prob(rec_U) \cdot \left(1 - \frac{1}{I_L}\right)\right)^i\right) \cdot rec_L\right)}{min\left(tr_{L(U,k)}, rec_L(U,k)\right)}$$

или, для упрощения вычислений,

$$\frac{t_L}{t_L(U,k)} \geq \frac{min\left(tr_L, \left(1+\sum_{i=1\ldots k}\left(prob(rec_U)\cdot\left(1-\frac{1}{I_L}\right)\right)^i\right)\cdot rec_L\right)}{min\left(tr_{L(U,k)}, rec_L + k\cdot rec_U\right)}$$

2.6 Выводы

В Главе 2 приведен анализ и классификация основных причин неэффективной работы оптимизаций в случае однофазной компиляции по сравнению с двухфазной, при которой собирается профиль исполнения задачи. Предложены применимые в однофазном режиме методы частичной раскрутки цикла, подкачки данных для нерегулярных чтений. Для архитектуры VLIW была показана их эффективность на группе задач, оптимизация которых представляет особую сложность из-за большого количества ветвлений и, следовательно, слабой параллелизуемости. Кроме того, чтобы обеспечить возможность применения по умолчанию метода уменьшения блокировок по чтению данных с помощью увеличения дистанции до чтений, предложен метод ее расчета, позволяющий эффективно ограничить возможное замедление задач.

Представленные методы позволяют получить значительное ускорение ряда задач и ускорить исполнение задач в среднем в режиме однофазной компиляции. Проведенный анализ причин замедления отдельных случаях позволяет предположить возможность классификации задач по типам, коррелирующим с производимым оптимизациями эффектом.

Дополнительно, по аналогии к методу частичной раскрутки цикла, для случаев как однофазной, так и двухфазной компиляции, предложен метод балансировки рекуррентностей цикла, находящихся под условием, позволяющий произвести более эффективную конвейеризацию.

Глава 3. Построение функционала качества для задачи многокритериальной оптимизации

3.1 Задача назначения оптимизационной последовательности

Вне зависимости от способа построения набора оптимизационных последовательностей после их фиксации требуется решить задачу назначения последовательности для компиляции конкретной процедуре. В случае итерационных решателей и необходимости минимизировать исключительно время исполнения без учета дополнительных свойств компиляции этот выбор производится довольно легко на основе апостериорной информации некоторого вида — или времени исполнения кода, полученного в результате компиляции различными линейками, или же расчетного времени исполнения кода по финальному планированию и значениям счетчиком линейных участков, собранным на тренировочном запуске. Для поставленной же проблемы предварительного назначения последовательности с учетом нескольких характеристик результата и процесса компиляции необходимо решить ряд подзадач, а именно:

1) Математически сформулировать критерий оптимального выбора, учитывающий несколько характеристик для случая, когда все результаты компиляции уже известны, и позволяющий при этом качественно оценить степень отклонения некоторого выбора от оптимального. То есть, во-первых, расширить условия апостериорной задачи дополнительными требованиями. Они могут включать в себя учет времени, требуемого для проведения компиляции процедур задачи выбранным набором оптимизационных последовательностей, времени исполнения задачи, размера получаемого кода, количества используемых при исполнении регистров и других характеристик, связанных с процессами компиляции и исполнения приложений. Во-вторых, сформулировать их виде, позволяющем в дальнейшем вычислить отклонение от минимума при априорном выборе.

2) Решить задачу апостериорного назначения оптимизационных последовательностей в соответствии с сформулированным критерием на некотором статистически значимом тренировочном наборе приложений.
3) Определить набор независимых характеристик процедур, доступных на раннем этапе компиляции, на основе которого возможно принятие априорного решения о назначении той или иной оптимизационной последовательности.
4) Формализовать и решить задачу автоматического выбора решения по известным значениям характеристик с учетом критерия оптимального выбора.

Пункты 1) и 2) являются этапами формализованной постановки задачи и ее точного математического решения, им посвящена Глава 3. Пункты 3) и 4) — постановка и решение задачи машинного обучения для имеющихся условий раскрыты в Главе 4.

3.2 Постановка задачи оптимального выбора

Чтобы выбрать алгоритм машинного обучения формализуем задачу, которую этот алгоритм должен решать.

Пусть есть некоторое множество процедур P и множество оптимизационных последовательностей L . Тогда назначение процедурам P некоторых последовательностей — это отображение

$$l^* : P \to L, l^* \in L^*$$

Чтобы сравнивать результаты различного назначения последовательностей, то есть элементы L^*, необходимо формализовать оценку качества назначения. Фиксация метрики для сравнения, или положительно определенного функционала, необходима как для апостериорного выбора лучших последовательностей для процедур по уже имеющимся результатам их компиляции и исполнения, то есть

$$\mu(l^*): L^* \to R\ ,$$

так и для априорного выбора последовательности по признакам процедур, чтобы обеспечить возможность оценки качества работы алгоритма машинного обучения:

$$\mu(a), \text{где } a: P \to L$$

В некоторых задачах машинного обучения, то есть в задачах построения алгоритма $a: X \to Y$, лучшее отображение $y*$ - это заранее известное отнесение объектов к некоторому классу, а μ - это просто количество или вероятность выбора правильного класса для объектов. В рассматриваем же случае для процедур приложений из обучающей выборки наиболее эффективные последовательности заранее не определены, их требуется определить по набору числовых характеристик, таких как время исполнения, время компиляции или размер итогового кода.

Далее будут перечислены и объяснены эмпирически возникающие свойства метрики μ качества компиляции и доказаны теоремы о ее представлении. Раздел 3.3 посвящен свойствам метрики, которые связаны с самой задачей компиляции приложений и выполняются даже при оптимизации только по одной характеристике компиляции, такой как время исполнения задач. В Разделе 3.4 ставится задача многокритериальной оптимизации характеристик компиляции и доказываются теоремы о представлении требуемой метрики в виде функций от нескольких характеристик компиляции процедур.

3.3 Свойства функционала качества компиляции

Общий вид математической записи метрики качества компиляции имеет следующий вид:

Пусть P *- множество процедур*

$L = (l_1, l_k)$ *- набор выбираемых оптимизационных линеек,*

$L^*: P \to L$ - *всевозможные отображения между множеством процедур и множеством линеек*

Тогда функционал качества компиляции — это отображение в область действительных чисел

$F: l^* \to R, l \in L^*$

Поскольку значение функционала всегда вычисляется для конечного числа процедур, то есть

$P = (p_{1.} p_n)$

то его удобно представлять, как

$F: L^n \to R$ - *функционал на множестве всевозможных вариантов назначения объектов L входным элементам из P, где*

$F(l^*(p_1), ...l^*(p_n)) = F(l^*)$

Для определенности будем считать, что более эффективный выбор оптимизационных последовательностей характеризуется меньшим значением функционала F. То есть, чтобы решить задачу оптимального выбора сущности, нужно найти его минимум в конфигурационном пространстве назначения линеек :

$$F\left(l^*(p_1)...l^*(p_n)\right) \to \min_{l^*}, l^*: P \to L \quad (1)$$

Рассмотрим теперь задачу минимизации времени исполнения приложений и докажем возникающие при этом свойства функционала F.

Свойство 1:

Ошибка назначения на некотором элементе зависит от выбранной неоптимальной последовательности.

То есть общем случае

$$F\left(l^*(p_1)...l_i^*...l(p_n)\right) ! = F\left(l^*(p_1)...l_j^*...l^*(p_n)\right)$$

для любых пар *(i,j)* .

Замечание: В большинстве алгоритмов классификации используется одинаковый вес любых ошибок на данном объекте. Это свойство, в частности, не позволяет использовать вероятностные методы.

Обоснование:

Покажем, что в общем случае $F\left(l^*\left(p_1\right)\ldots l_i\ldots l^*\left(p_n\right)\right)!=F\left(l^*\left(p_1\right)\ldots l_j\ldots l^*\left(p_n\right)\right)$ для любых пар *(i,j)*. Для этого достаточно рассмотреть задачу назначения линеек на множестве, которое состоит из одной процедуры, и для которой время исполнения после компиляции каждой линейкой отличается, то есть

$$exe\left(l_i, p\right)!=exe\left(l_j, p\right) \quad (2)$$

где $exe\left(l_i, p\right)$ - время исполнения процедуры p после компиляции линейкой l_i

В этом случае функционал должен отображать отличие во времени исполнения между каждыми двумя линейками, то при $exe\left(l_i, p\right)<exe\left(l_j, p\right)$ должно выполняться $F\left(l_i\right)<F\left(l_j\right)$. Поэтому из (2) следует, что все значения $F\left(l_i\right)$ - различны.

Свойство 2:

Если некоторой процедуре назначена неоптимальная последовательность с точки зрения минимизации метрики, то в общем случае отклонение значения метрики от минимума, или же ошибка, зависит от всего множества назначений. То есть функционал

$$F\left(l^*\left(p_1\right)\ldots l^*\left(p_n\right)\right)$$

не представим в виде суммы

$$F_1\left(l^*\left(p_1\right)\right)+\ldots+F_n\left(l^*\left(p_n\right)\right)$$

Замечание: Проведенный анализ описанных в литературе методов машинного обучения, которые будут далее перечислены в Главе 4, показал их неприменимость для ошибки с таким свойством.

Обоснование:

Рассмотрим задачу сравнения только времени исполнения некоторого пакета приложений, каждое из которых состоит из некоторого набора процедур, и покажем, что пользовательская оценка больше всего соответствует нелинейной формуле сравнения. Самой простой оценкой времени исполнения является:

1 . *Сравнение суммы времени исполнения всех приложений то есть сравнение суммы времени исполнения всех процедур.*

Пусть $\mathrm{exe}\left(p_i, l^*\left(p_i\right)\right)$ - время исполнения процедуры p_i при использовании линейки $l^*\left(p_i\right)$, тогда требуемый функционал имеет следующий вид

$$F\left(l^*\left(p_1\right), l^*\left(p_n\right)\right)=\sum_i \mathrm{exe}\left(p_i, l^*\left(p_i\right)\right) \quad (3)$$

В результате минимизации функционала (1) будет получено минимальное суммарное время исполнения всего пакета. Однако, небольшие по времени приложения могут сильно терять в производительности из-за одновременного ускорения одного долгого. Поэтому для достаточно взвешенного по типу приложений пакета производительности больше подходит:

2. *Сравнение суммы относительного времени исполнения задач.*

Пусть $\mathrm{exe}\left(p_i, l^*\left(p_i\right)\right)$ - время исполнения процедуры p_i при использовании линейки $l^*\left(p_i\right)$, $w_k=\sum_{ki} \mathrm{exe}\left(p_{ki}, l^*_{start}\right)$, где ki - индексы процедур, входящих в задачу k, l_{start} - начальная оптимизационная последовательность, тогда взвешенный по временам исполнения функционал выглядит следующим образом:

$$F\left(l^*\left(p_1\right), \ldots l^*\left(p_n\right)\right)=\sum_k\left(\sum_{ki} \mathrm{exe}\left(p_{ki}, l^*\left(p_{ki}\right)\right) / w_k\right) \quad (4)$$

Замечание: Минимум построенного функционала зависит от

оптимизационной последовательности, для которой были зафиксировано значения времен исполнения, или же *начальной последовательности*. Здесь и далее под ней будет подразумеваться базовая оптимизационная последовательность. При этом в начальной точке $F\left(l^*\left(p_1\right),\ldots l^*\left(p_n\right)\right)=k$.

Недостаток учета взвешенных (2) и нет (1) сумм времен исполнения приложений состоит в том, что замедление приложений рассматривается более значимым, чем их ускорение. Так, при арифметическом среднем ускорение одной задачи в 2 раза одновременно с замедлением другой в 1,5 не считается выгодным. В результате этого поиск оптимума начинает сильно зависеть от стартовой точки, и одно исходно эффективно скомпилированное приложение не позволяет ускорить остальные. Более независимым от стартового качества компиляции функционалом является взятие произведения времен исполнения приложений или же их среднего геометрического:

3. *Минимизация среднего геометрического взвешенных времен исполнения приложений* означает, что

$$F\left(l^*\left(p_1\right),\ldots l^*\left(p_n\right)\right)=\sqrt[K]{\prod_K\left(\sum_{ki}\mathrm{exe}\left(p_{ki},l^*\left(p_{ki}\right)\right)\right)} \quad (5)$$

где K — количество приложений.

Достаточно легко показать, что построенный функционал (5) не приближается суммой по отдельным объектам. Взятием корня можно пренебречь, поскольку он не влияет на результат сравнения значений функционала. Поэтому остается

$$F\left(l^*\left(p_1\right),\ldots l^*\left(p_n\right)\right)=\prod_K\left(\sum_{ki}\mathrm{exe}\left(p_{ki},l^*\left(p_{ki}\right)\right)\right) \quad (6)$$

то есть для двух линеек и двух приложений с одной процедурой в каждой функционал имеет вид

$$F\left(l^*\left(p_1\right),l^*\left(p_2\right)\right)=exe\left(p_1,l^*\left(p_1\right)\right)*exe\left(p_2,l^*\left(p_2\right)\right)$$

Замечание: Стоит отметить, что построенный функционал (6) является

является наиболее часто используемым для оценки изменения производительности компиляторов на тестовом пакете приложений.

Различие функционалов (3), (4), (5) в общем случае дает разный результат сравнения наборов назначения оптимизационный последовательностей, но можно показать, что верно и следующее утверждение:

Замечание: Для апостериорного выбора последовательностей оптимизации минимизация функционалов 3-5 дает один и тот же результат. Для априорного выбора последовательностей оптимизации минимизация каждого из функционалов 3-5 может давать различные результаты.

Свойство 3:

> Для идентичных с точки зрения всех вычислимых характеристик процедур оптимальная последовательность преобразований может отличаться, то есть теоретический минимум метрики является недостижимым для машинного решения.

Обоснование:

Указанное свойства следует из того, что настройки наиболее эффективной оптимизационной последовательности для процедуры могут отличаются в случае разных входных данных. Примером является ситуация, когда входные данные программы задают число итераций цикла. В этом случае может значительно меняться оптимальный фактор раскрутки, варьироваться эффективное число стадий программной конвейеризации, а предварительная подкачка данных для последующих итераций может как уменьшать количество промахов, так и просто создавать бесполезную дополнительную нагрузку на кэш-память.

3.4 Построение многокритериального функционала качества

Формализуем теперь задачу уменьшения не только времени исполнения

процедур, но и времени компиляции или размера кода. Задача одновременной оптимизации нескольких характеристик является задачей многокритериальной оптимизации.

Определение: Задача *многокритериальной минимизации* это задача вида

$$F(x)=(f_1(x),\dots,f_k(x))\to min,\ x\in X,\ F(x)\colon X\to R^K,\ K\geq 2$$

Поскольку в общем случае не существует единого решения для всех целевых функций, то вместо него ищут *Парето-минимумы* [51], то есть такие $x\in X$, что не существует $x'\in X$, для которых $f_i(x)\geq f_i(x')\ \forall i=1..\mathrm{k}$ и $f_i(x)>f_i(x')$ хотя бы для одного *i*.

В этом разделе будет доказано, что задача многокритериальной оптимизации для задачи компиляции решается путем минимизации единого функционала качества, который представим в виде функции от критериев, а также будет доказана теорема о точной структуре соответствующей функции.

Стоит отметить, что в процессе разработки компилятора задача уменьшения времени компиляции или ограничения размера итогового кода возникает достаточно часто. Как правило, она решается эвристически с помощью общего ограничения для отдельной оптимизации, которая на некотором контексте приводит к нелинейному росту кода или к неразумно большому времени компиляции. Так, для преобразования оптимизации критического пути, содержащего потенциально зависимые по данным операции чтения и записи (crp_opt), используется ограничение на количество чтений. По сути, при этом допускается некоторая потеря производительности за счет контроля времени компиляции и размера кода. В статье [17] исследуется вопрос подбора минимальной последовательности оптимизации для каждой процедуры, позволяющей получить максимальную скорость исполнения. На тестовом пакете для компилятора gcc авторам удалось получить ускорение

компиляции в среднем в 2 раза без потери по производительности. Но для такого ускорения компиляции было использовано в некотором смысле неограниченное число линеек, так как для каждой процедуры набиралась и применялась отдельная последовательность оптимизаций до тех пор, пока проведение преобразований давало эффект.

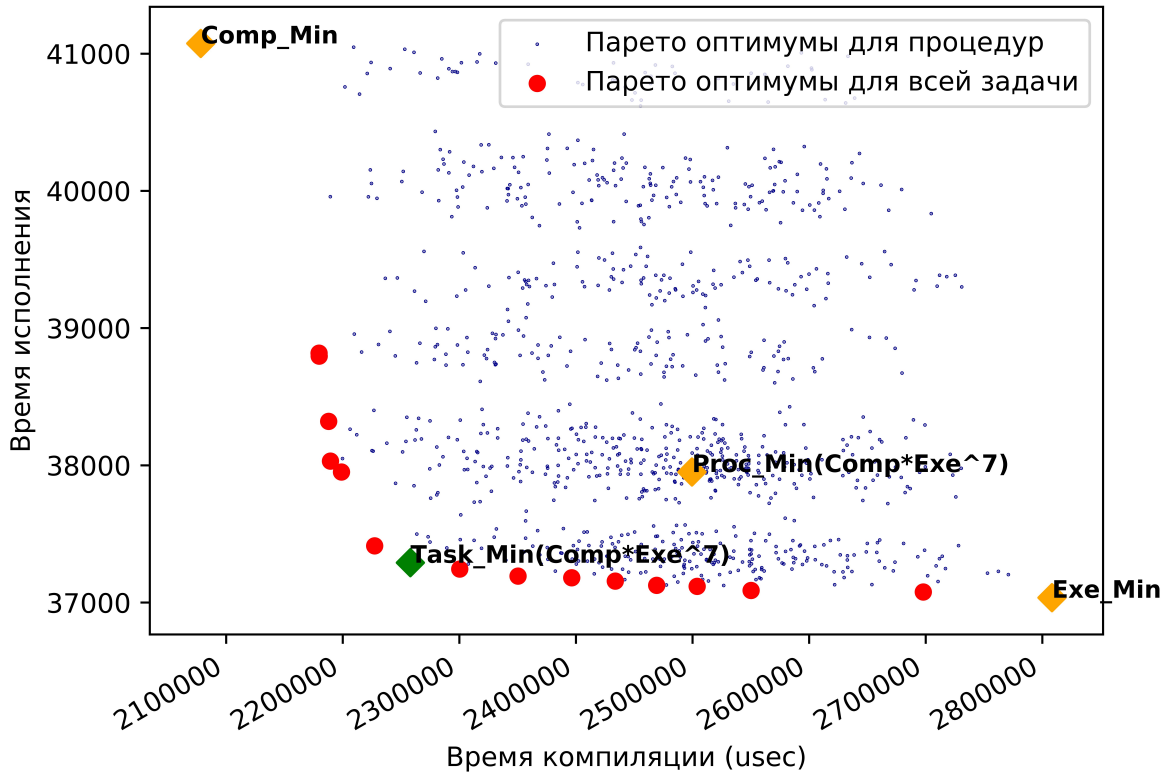

Рисунок 9. Граница Парето для времен компиляции и исполнения

Особенностью обоих примеров является попытка найти некоторые точки на границе Парето-минимумов в соотношении времени компиляции и времени исполнения отдельной процедуры. Однако, с точки зрения пользователя, больший интерес представляет граница Парето для совокупного множества процедур и всего процесса компиляции и исполнения. Существенное отличие этих требований отображено на Рисунке 9 на примере задачи 175.vpr. Под «Парето оптимумами для процедур» подразумевается суммарное время компиляции и исполнения задачи, при котором для каждой процедуры используется некоторая точка из ее границы Парето-минимумов; оптимумы же

для всей задачи выбираются на совокупности точек суммарного времени исполнения и компиляции всех процедур приложения. Для наглядности указаны точки Comp_Min, где для каждой процедуры выбрана самая быстрая по времени компиляции линейка, и Exe_Min, где для каждой процедуры используется, которая приводит к самому быстрому исполнению кода. Кроме того, вычислены и указаны точки минимумов некоторого соотношения между временами компиляции и исполнения, подсчитанные для задачи и для отдельных процедур. Как видно, использование независимых попроцедурных характеристик приводит к неудовлетворительному результату для всей задачи.

То есть,

Свойство 4:

> Для решения задачи многокритериальной оптимизации необходимо использовать числовые характеристики компиляции, зависящие от назначения оптимизационных последовательностей сразу всем процедурам приложения.
>
> То есть минимум функционала $l^*_{min} = \min\limits_{l^*: P \to L} F\left(l^*(p_1) \ldots l^*(p_n)\right)$
>
> не представим в виде $l^* = \left(\min\limits_{l \in L}(F_1(l)), \ldots \min\limits_{l \in L}(F_n(l))\right)$

Замечание:

Свойство 4 является более сильным, чем показанное свойство 2) для случая одного критерия. Так, при минимизации только времени исполнения Свойство 4) не выполняется, поскольку минимальное время исполнения каждой процедуры даст минимальное время исполнения их суммы.

Пример. В таблицах 1 и 2 построен пример взаимной зависимости назначения, возникающей в случае минимизации времени исполнения в условиях наложения ограничения на время компиляции. Использовать лучшую по исполнению линейку для обеих процедур при заданном ограничении

невозможно, поэтому требуется решение совместной задачи.

	Exe line 1	Exe line 2	Comp line 1	Comp line 2
Procedure 1	500	800	60	40
Procedure 2	400	500	60	40

Таблица 1: Время компиляции и исполнения процедур 1 и 2 при использовании линеек 1 и 2

For Procedure1 /Procedure2	Line 1/line 1	Line 1/ line 2	Line 2/ line 1	Line 2/ line 2	Best Choise
Exe time	900	1000	1200	1300	1000
Comp	120	100	100	80	100

Таблица 2: Выбор в условиях взаимной зависимости назначения – выбор минимального суммарного времени исполнения при условии ограничения общего времени компиляции значением Comp=100

Обозначение: $f_i\left(l^*(p_1)\ldots l^*(p_n)\right)$ - числовые характеристики компиляции для всего множества назначений, их также можно рассматривать как функции $f_i: L^* \to R$.

3.4.1 Теорема о минимуме многокритериального функционала качества

Попробуем теперь перейти от задачи многокритериальной оптимизации

$F(x)=(f_1(x),\ldots,f_k(x))\to min, x\in L^n, F(x): X\to R^k$

к минимизации обобщенного числового критерия $f(x)\to min$, где $F(x)$ представляет собой некоторую многопараметрическую функцию $F\left(f_1(x),\ldots,f_k(x)\right)$, или же к многокритериальному функционалу качества $F\left(l^*(p_1)\ldots l^*(p_n)\right)$. Такой переход называется скаляризацией задачи многокритериальной оптимизации [52] [53].

Теорема 3: Если многокритериальный функционал качества представляет из себя монотонную строго возрастающую функцию от нескольких

характеристик, то его минимум достигается на на границе Парето-минимумов для этих характеристик.

Замечание: Отношение порядка для двух точек из R^k определено только в том случае, когда отношение всех несовпадающих координат одинаково. Поэтому только для этих точек и будет выполняться условие монотонности функционала.

Доказательство:

Пусть F - монотонно возрастающий функционал от k переменных, X - множество всех точек, $f_1(x), \dots, f_k(x)$ определены всюду на X. И пусть была найдена точка x_0, в которой достигается минимум F на X, то есть

$$F\left(f_1(x_0), \dots, f_k(x_0)\right) = \min_{x \in X}\left(F\left(f_1(x), \dots f_k(x)\right)\right).$$

Допустим, что существует точка x_1, нарушающая оптимальность по Парето для x_0, то есть $\forall i : f_i(x_1) \le f_i(x_0)$ и $\exists j : f_j(x_1) < f_j(x_0)$.

Т.к. F — монотонно возрастающая функция и $f_1(x_1) \le f_1(x_0)$, то

$F\left(f_1(x_1), \dots, f_k(x_1)\right) \le F\left(f_1(x_0), \dots, f_k(x_1)\right)$, аналогично и для всех других характеристик, кроме j. Поскольку значение j-ой характеристики для x_0 строго больше значения для x_1, то есть $f_j(x_1) < f_j(x_0)$, то

$F\left(f_0(x_1), \dots, f_k(x_1)\right) < F\left(f_0(x_0), \dots, f_k(x_0)\right)$

То есть значение F в x_1 меньше значения F в x_0, то есть мы получили противоречие с тем, что x_0 является минимумом функционала F. Значит, допущение неверно.

Строение такого функционала задаст рассматриваемое как оптимальное соотношение между указанными характеристиками. Стоит отметить, что выбор этого соотношения не является однозначным и зависит от решаемой задачи.

Пример 1: Если требуется получить максимально быстрый код, то, при

указании времен компиляции и исполнения в тактах, функционал может иметь вид:

$$F\left(l^*\left(p_1\right),\ldots l^*\left(p_n\right)\right)=\sum_{ki}\mathrm{exe}\left(p_{ki},l^*\left(p_{ki}\right)\right)-1/\left(\sum_{ki}\mathrm{comp}\left(p_{ki},l^*\left(p_{ki}\right)\right)+1\right)$$

В приведенном примере шаг первого слагаемого всегда больше, чем второе слагаемое, поэтому учет времени компиляции будет производиться только при минимальном значении времени исполнения.

Пример 2:

Для задачи динамической компиляции или JIT (Just in Time) время компиляции и исполнения одинаково значимы, поэтому может использоваться функционал вида:

$$F\left(l^*\left(p_1\right),\ldots l^*\left(p_n\right)\right)=\sum_{ki}\mathrm{exe}\left(p_{ki},l^*\left(p_{ki}\right)\right)+\sum_{ki}\mathrm{comp}\left(p_{ki},l^*\left(p_{ki}\right)\right) \quad (7)$$

3.4.2 Теорема о сохраняющих минимумы при растяжениях функциях

Для уточнения структуры функционала для статической компиляции введем дополнительное пользовательское требование.

Свойство 5: Поскольку заранее число исполнений кода неизвестно, то естественно таким образом выбрать линейки оптимизаций для процедур, чтобы они оставались эффективными при разном его значении. То есть растяжение одной из координат не должно менять результат сравнения функционала в двух точках пространства критериев.

Опишем указанное свойство более формально.

Определение: Пусть F - определенная всюду на R_+^{k} функция от k переменных. Свойство *сохранения минимума подмножества при растяжениях* определим как: $\forall X\subset R_+^{\mathrm{k}}$, из $F\left(x_1,\ldots x_j,\ldots x_k\right)=\min\limits_{\vec{x}\in X}F\left(\vec{x}\right)$ следует, что

$\forall a>0,\forall j$, $F\left(x_1,\ldots a*x_j,\ldots x_k\right)=\min\limits_{\vec{x}\in X_j^a}F\left(\vec{x}\right)$ (8), где множество X_j^a

построено из X умножением j-ой координаты всех его элементов на a .

Чтобы описать все множество подходящих скаляризаций, далее автором будет определен весь класс строго монотонных непрерывных функций, обладающих указанным свойством.

Для начала сформулируем более удобное для дальнейшего использования свойство.

Определение: Определенная на R_+^k функция F удовлетворяет свойству сохранения *порядка при растяжениях*, если $\forall \langle x_{11}, \dots x_{1k} \rangle \in R_+^k$, $\forall \langle x_{21}, \dots x_{2k} \rangle \in R_+^k$ из $F(x_{11}, \dots x_{1j}, \dots x_{1k}) \le F(x_{21}, \dots x_{2j}, \dots x_{2k})$ следует, что

$$\forall a > 0, \forall 1 \le j \le k: \quad F(x_{11}, \dots a * x_{1j}, \dots x_{1k}) \le F(x_{21}, \dots a * x_{2j}, \dots x_{2k}) \tag{9}$$

Лемма 1: Пусть F - определенная всюду на R_+^k функция от k переменных. Выполнение для нее свойства сохранения минимума подмножества при растяжениях равносильно свойству сохранения порядка при растяжениях.

Доказательство:

Пусть сохраняется минимум при растяжениях. Рассмотрим точки $x_1 \in R_+^k$ и $x_2 \in R_+^k$, для которых $F(x_1) \le F(x_2)$. Чтобы проверить выполнение свойства (9) для x_1, x_2 достаточно рассмотреть подмножество $X \subset R_+^k$, состоящее только их них. В обратную сторону утверждение следует из определения минимума F на подмножестве $X \subset R_+^k$.

Лемма 2:

Пусть $f(x, y)$ — монотонно возрастающая (монотонно убывающая) функция от двух переменных, определенная всюду на R_+^2 и представимая в виде $f(x, y) = g(x^p \cdot y^q)$, где $p \ge 0, q \ge 0$, тогда

1) $g(x)$ - монотонно возрастающая (монотонно убывающая) функция от

одного параметра,

2) $f(x, y)$ удовлетворяет свойству (9) сохранения монотонности при растяжениях.

Доказательство:

Пусть для определенности $f(x, y)$ - монотонно возрастающая.

1) Пусть $x_1 \le x_2$, то есть $\forall y: f(x_{1,} y) \le f(x_{2,} y)$. Пусть для определенность p>0 (иначе рассмотрим q). Тогда $f\left(x_1^{\frac{1}{p}}, 1\right) \le f\left(x_2^{\frac{1}{p}}, 1\right)$ из монотонности, то есть $g(x_1) \le g(x_2)$.

2) Рассмотрим теперь $x_{1,} x_{2,} y_{1,} y_2: f(x_{1,} y_1) \le f(x_{2,} y_2)$. Тогда $g\left(x_1^p \cdot y_1^q\right) \le g\left(x_2^p \cdot y_2^q\right)$ и, следовательно, из 1), $x_1^p \cdot y_1^q \le x_2^p \cdot y_2^q$. В то же время, $f(a \cdot x_{1,} b \cdot y_1) = g\left(a^p \cdot b^q \cdot x_1^p \cdot y_1^q\right)$, $f(a \cdot x_{2,} b \cdot y_2) = g\left(a^p \cdot b^q \cdot x_2^p \cdot y_2^q\right)$. И, поскольку из $x_1^p \cdot y_1^q \le x_2^p \cdot y_2^q$ следует $a^p \cdot b^p \cdot x_1^p \cdot y_1^q \le a^p \cdot b^q \cdot x_2^p \cdot y_2^q$, то $f(a \cdot x_{1,} b \cdot y_1) \le f(a \cdot x_{2,} b \cdot y_2)$

Замечание:

Аналогично можно показать, что Лемма 2 справедлива для функций любого количества переменных.

Теорема 4:

Пусть $f(x, y)$ — определенная всюду на R_+^2 строго монотонная непрерывная функция от двух переменных, удовлетворяющая свойству (9) на всей области определения. Тогда она представима в виде:

$$f(x, y) = g(x^q \cdot y), \text{ где } q \in R, q > 0$$

Доказательство:

1) Покажем, что через любую точку $(x_{0,} y_0)$ проходят линии уровня функции $f(x, y)$ вида:

$$y \cdot x^q = y_0 \cdot x_0^q, q > 0 \quad (10)$$

Поскольку $f(x,y)$ - непрерывна и строго монотонна, то $f(x_0-\epsilon, y_0)<f(x_{0,}y_0)<f(x_{0,}y_0+\epsilon)$ для $\epsilon>0$. По теореме Больцано-Коши на отрезке $[(x_0-\epsilon, y_0),(x_{0,}y_0+\epsilon)]$ существует точка $(x_{1,}y_1)$, в которой $f(x_{1,}y_1)=f(x_{0,}y_0)$. Точка $(x_{1,}y_1)$ не совпадает с концами отрезка $[(x_0-\epsilon, y_0),(x_{0,}y_0+\epsilon)]$, поэтому $x_1<x_{0,}y_1>y_0$. Рассмотрим $a=\frac{x_1}{x_0}, b=\frac{y_1}{y_0}$, то есть $f(x_{0,}y_0)=f(a\cdot x_{0,}b\cdot y_0)=C$, при этом $a\neq 1, b\neq 1$.

Из свойства (8) следует, что одновременно выполняется $f(a\cdot x_{0,}b\cdot y_0)\leq f(a^2\cdot x_{0,}b^2\cdot y_0)$ и $f(a\cdot x_{0,}b\cdot y_0)\geq f(a^2\cdot x_{0,}b^2\cdot y_0)$, то есть $f(a\cdot x_{0,}b\cdot y_0)=f(a^2\cdot x_{0,}b^2\cdot y_0)$, аналогично, $f(a^n\cdot x_{0,}b^n\cdot y_0)=C\,\forall\, n\in N$.

Теперь можно показать, что:

$$f(a^t\cdot x_{0,}b^t\cdot y_0)=C\,\forall\, t\in R \quad (11)$$

Для этого достаточно построить сходящуюся последовательность к каждой промежуточной степени разбиением отрезков пополам. Докажем равенство для середины между любыми уже проверенными точками:

Пусть $f(a^{0.5}\cdot x_{0,}b^{0.5}\cdot y_0)>f(x_{0,}y_0)$, тогда $f(a\cdot x_{0,}b\cdot y_0)>f(a^{0.5}\cdot x_{0,}b^{0.5}\cdot y_0)>f(x_{0,}y_0)$ - противоречие, аналогично доказывается противоречие для предположения $f(a^{0.5}\cdot x_{0,}b^{0.5}\cdot y_0)<f(x_{0,}y_0)$.

Равенство (11) означает, что кривая $\gamma(t)=(x(t),y(t))$, где $x(t)=a^t\cdot x_0$, $y(t)=b^t\cdot y_0$, - линия уровня функции f. Выражение *y* через *x* для этой кривой дает:

$$y=\frac{y_0}{x_0^{\log_a(b)}}\cdot x^{\log_a(b)} \quad (12)$$

То есть через любую точку $(x_{0,}y_0)$ проходит линия уровня вида (12). Поскольку $f(x,y)$ - строго монотонная, то $\log_a(b)<0$. Обозначим

$q=-\log_a(b)$, тогда линия уровня принимает вид $y\cdot x^q=y_0\cdot x_0^q, q>0$.

2) Покажем теперь, что если есть две линии уровня $y\cdot x^q=a, q>0$, $y\cdot x^r=b, r>0$, то $r=q$. Предположим, что это не так и попробуем найти точку пересечений этих линий, то есть надо найти $\frac{x^q}{a}=\frac{x^r}{b}$. Если $r\neq q$, то решение существует: $x=\frac{a}{b}^{1/(q-r)}$. То есть указанные линии уровня — для одинакового значения $f(x,y)$. При этом все точки между этими линиями принимают то же значение из-за монотонности. Получили противоречие со строгой монотонностью.

3) Из 1) и 2) следует, что через любую точку проходит линия уровня вида $y\cdot x^q=y_0\cdot x_0^q, q>0$ с одинаковым q для всей области определения. Докажем, что $f(x,y)$ представима в виде $g(x^q\cdot y)$. Для этого достаточно показать, что $\forall x_{1,}y_{1,}x_{2,}y_2$ из $x_1^q\cdot y_1=x_2^q\cdot y_2$ следует, что $f(x_{1,}y_1)=f(x_{2,}y_2)$. Но поскольку $x_1^q\cdot y_1=x_2^q\cdot y_2$, то они лежат на одной линии уровня функции f, то есть $f(x_{1,}y_1)=f(x_{2,}y_2)$.

Мы определили весь класс функций, которые удовлетворяют условию (8) *сохранения минимума подмножества при растяжениях.* Однако, можно показать, что минимумы таких функций могут быть найдены путем минимизации меньшего класса функций.

Лемма 3:

Пусть $X\subset R_+^{\mathrm{k}}$, а $F:R_+^k\to R$, определенная всюду на X, представима в виде $F(x_{1,}\ldots x_k)=g\left(x_1^{j1}\cdot\ldots\cdot x_k^{jk}\right)$, для всех $x=\langle x_{1,}\ldots x_k\rangle\in X$, где q - строго возрастающая функция. Тогда, если существует $\min\limits_{x\in X}(F(x))$, достижимый в

некоторой точке $\langle x_{1,} \dots x_k \rangle \in X$, то он достигается в точке минимума

$$\min_{x=\langle x_{1,} \dots x_k \rangle \in X} \left(x_1^{j1} \cdot \dots \cdot x_k^{jk} \right) .$$

Доказательство:

Указанный минимум является так же и минимумом функции g , достижимым в точке $x_1^{j1} \cdot \dots \cdot x_k^{jk}$. Допустим, что для некоторой точки $z = \langle z_{1,} \dots z_k \rangle \in X \quad z_1^{j1} \cdot \dots \cdot z_k^{jk} < x_1^{j1} \cdot \dots \cdot x_k^{jk}$, тогда $g(z) < g(x)$. Получили противоречие.

Утверждение: Из доказанных теорем 3 и 4, лемм 1-3 следует, что минимумы всех скаляризованных целевых функций процесса компиляции с двумя критериями могут быть найдены путем минимизации функционалов вида:

$$F\left(l^*(p_1), \dots l^*(p_n)\right) = \left(\sum\nolimits_i \operatorname{exe}\left(l^*(p_i)\right)\right)^r * \left(\sum\nolimits_i \operatorname{comp}\left(l^*(p_i)\right)^j\right) \quad (13)$$

В случае *K* задач, используя (5), получаем:

$$F_K\left(l^*(p_1), \dots l^*(p_n)\right) = \underset{K}{geomean}\left(\left(\sum\nolimits_{ki} \operatorname{exe}\left(l^*(p_{ki})\right)\right)^r * \left(\sum\nolimits_{ki} \operatorname{comp}\left(l^*(p_{ki})\right)\right)^j\right) \quad (14)$$

где $geomean(x_{1,} \dots x_K) = \sqrt[K]{\prod\nolimits_K (x_k)}$

Замечание: Выбор коэффициентов r и j позволяет регулировать уровень оптимизации, контролируя пропорцию между допустимым замедлением производительности и одновременным сокращением времени компиляции. В соответствии с доказанным, $p = \frac{j}{r} \in R_+$ может быть любым, однако для ускорения вычислений при использовании функционала предполагается рассматривать только значения $p = \frac{j}{r} \in Q_+$.

Помимо учета времени компиляции дополнительно ко времени исполнения использование функционала позволяет регулировать и другие требования к оптимизации. Так, можно в явном виде задать требование на изменение размера получающегося кода всей задачи или уменьшать количество используемых регистров и так далее. Однако, на указанные дополнительные критерии не распространяется требование сохранения монотонности при растяжениях, поэтому оптимальный функционал будет иметь несколько иной вид. К примеру, самое естественное жесткое ограничение размера всей задачи значением SizeMax имеет вид:

$$F\left(l\left(p_1\right),\ldots l\left(p_n\right)\right)=\underset{K}{geomean}\left(\sum\nolimits_{ki}\operatorname{exe}\left(l\left(p_{ki}\right)\right)\right)^{r}*\underset{K}{geomean}\left(\frac{1}{SizeMax-\sum\nolimits_{ki}\operatorname{size}l\left(p_{ki}\right)}\right)$$

Стоит также отметить, что в неявном виде учет размера кода производится и при контроле исключительно времени компиляции, связано это с тем, что время работы многих оптимизирующих фаз как правило линейно или квадратично зависит от числа операций в процедуре.

3.4.3 Минимумы функционала качества и Парето-минимумы

Через каждую точку с положительными координатами проходит бесконечное число различных линий уровня для функционалов из условия теоремы 4, то есть кривых вида

$$x^{q}\cdot y=c\,,q>0 \quad (15)$$

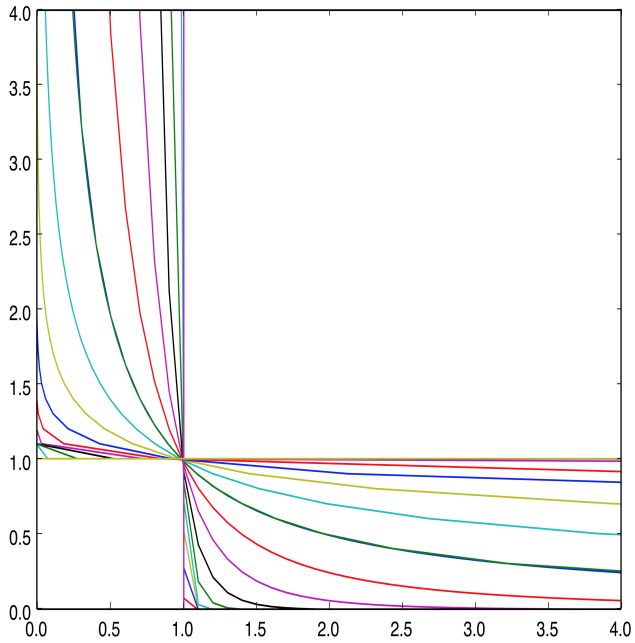

Рисунок 10. Функции вида $y^p \cdot x = 1, y \cdot x^p = 1, p = 2^k, k \in N, p < 1000$

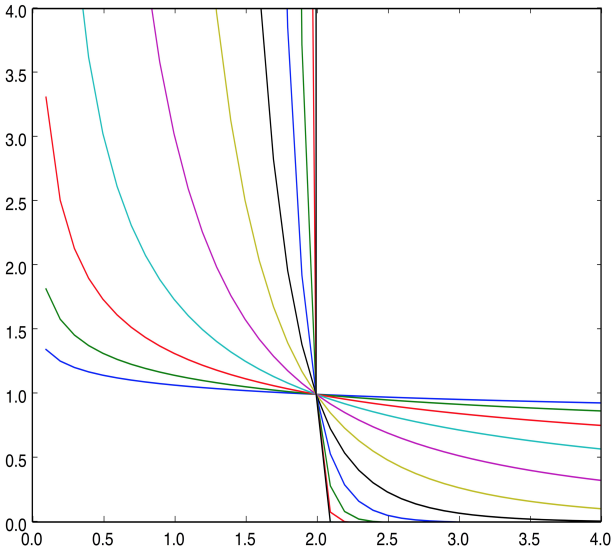

Рисунок 11. Функции вида $y \cdot x^p = c, p = 2^k, k \in N, p < 1000, c = 2^p$

Аналогично, для любого положительного значения c можно построить бесконечно много кривых такого вида. Для наглядности на Рисунке 10 построено несколько таких линий уровня для значения 1, а на Рисунке 11 построены линии уровня различных функционалов, которые проходят через

точку $[x, y]=[2,1]$. Поскольку функционал строго положительный, то минимум какого-то из них будет достигаться в заданной точке $[x, y]$ в том случае, если все остальные точки рассматриваемого множества лежат на его линии уровня, справа, или сверху от нее. При минимизации разных функционалов могут быть найдены разные точки, то есть разные Парето-минимумы. Однако, при этом некоторые точки границы Парето не могут быть вычислены. Следующая лемма описывает соответствие Парето-минимумов и возможным минимумов функционалов вида (15). Утверждения леммы проиллюстрированы на Рисунке 12.

Лемма:

1) Через любые две точки границы Парето $[x_{1,} y_1]$ и $[x_{2,} y_2]$ можно провести кривую вида (15), то есть $\exists c>0, q>0 : x_1^q \cdot y_1 = c, x_2^q \cdot y_2 = c$
2) Пусть для определенности в 1) $x_1 < x_{2,} y_1 > y_2$, а построенная кривая - $g(x, y)$. И пусть $[x_{3,} y_3]: x_1 \le x_3 \le x_2, y_1 \ge y_3 \ge y_2$. Тогда, если $g(x_{3,} y_3) > g(x_{1,} y_1) = g(x_{2,} y_2)$, то точка $[x_{3,} y_3]$ не может являться минимумом функции вида $f(x, y) = x^p \cdot y$ ни для какого p.
3) Пусть через каждые две точки проведены кривые вида (15), то есть в них некоторым функциям $\{g_i(x, y)\}$ сопоставлены значения $\{c_i\}$ в этих точках, тогда произвольная точка $[x, y]$ границы Парето является минимум некоторой функции вида $f(x, y) = x^p \cdot y$ тогда и только тогда, когда $\forall x_1 < x_{2,} y_1 > y_2 : x_1 \le x \le x_2, y_1 \ge y \ge y_2$ выполнено $g_i(x, y) \le c_i$ для соответствующей $[x_{1,} y_1], [x_{2,} y_2]$ функции g_i.

Доказательство:

1) Вычислим $r : y_1 \cdot x_1^r = y_2 \cdot x_2^r$, то есть $r = \log_{\frac{x_1}{x_2}}(\frac{y_2}{y_1})$, поскольку для границы

Парето верно $\{x_1 < x_2, y_1 > y_2\}$ или $\{x_1 < x_2, y_1 > y_2\}$, то $r > 0$

2) Указанное неравенство означает, что при некотором p $x_1^p \cdot y_1 < x_3^p \cdot y_3$ и $x_2^p \cdot y_2 < x_3^p \cdot y_3$. Поскольку $x_1 \le x_3 \le x_2$, то первое сравнение не поменяет знак при увеличении p, а второе при его уменьшении. То есть при любом p одно из неравенств будет выполнено, поэтому функция не будет достигать своего минимума в $[x_3, y_3]$.

3) Первая часть утверждения доказана в 2). Покажем, что если 2) не верно ни для какой пары точек относительно $[x, y]$, то найдется p, при котором в ней будет достигнут минимум. Выбросим все точки, на которых минимум не достигается ни для какого p. В этом множестве будут по крайней мере две точки с минимальным x и минимальным y при $p \to \infty$ и при $p = 0$. Рассмотрим теперь две соседние точки с рассматриваемой $[x, y]$ и построим функцию, соответствующую проходящей через них кривой из 1). Поскольку для этих точек выполнено 2), то значение функции в остальных точках множества не меньше значения в них, поэтому из выполнения 2) для $[x, y]$, на функции этой кривой достигается минимум в точке .

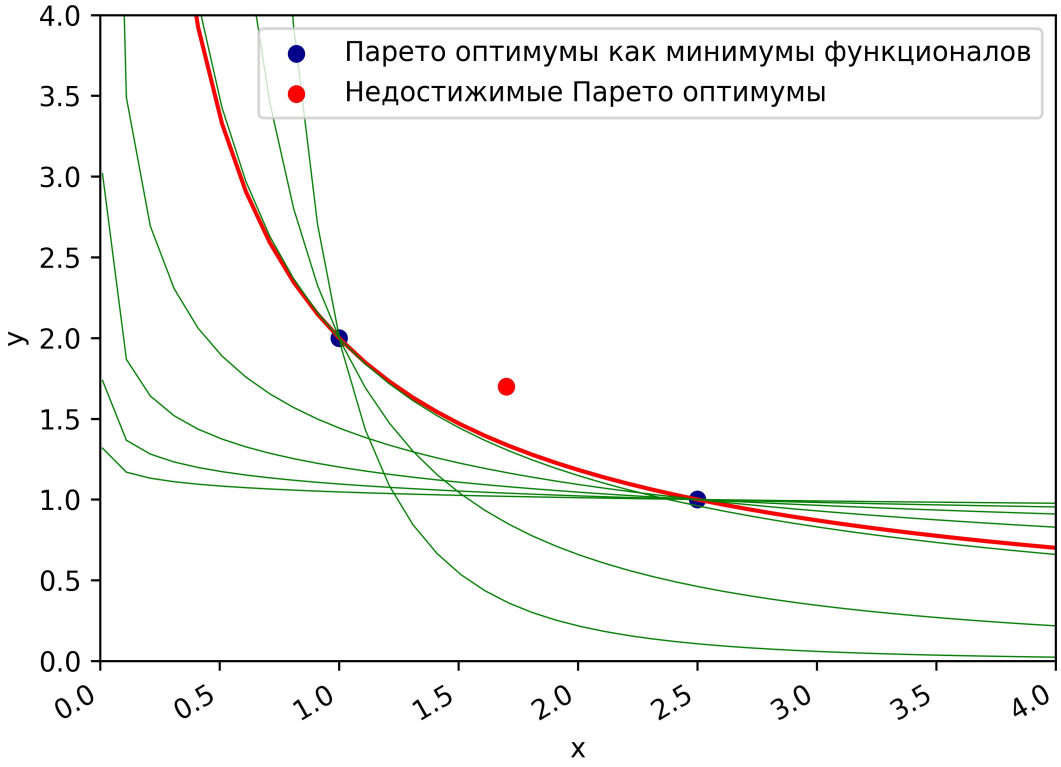

Рисунок 12. Достижимые как минимумы функционалов Парето оптимумы.

3.5 Результаты применения минимума функционала

В качестве тренировочной базы были использованы приложения пакетов CINT spec2000, CFP spec2000. Чтобы полностью исключить воздействие процедур друг на друга при поиске апостериорного минимума функционала, необходимо собрать времена исполнения приложений для всех возможных вариантов сборки входящих в них процедур. Те есть для n процедур и k оптимизационных последовательностей требуется произвести n^k запусков. Это технически не осуществимо за приемлемое время, поэтому были проведены запуски приложений, при компиляции которых для всех процедур использовалась одна и та же оптимизационная последовательность.

Время исполнения каждой процедуры было замерено с помощью инструмента dprof. В этом случае через равные промежутки времени

сохраняется номер исполняющегося такта машинного кода приложения, то есть присутствует небольшая вероятностная ошибка. Времена компиляции отдельных процедур вычислялись с момента начала попроцедурной оптимизации с помощью внутренней утилиты в микросекундах. Поскольку межпроцедурная часть не разделима для процедур и одинакова для разных оптимизационных последовательностей, то в соответствующих функционалах ее время было добавлено к сумме времен попроцедурной компиляции. То есть, полученные данные представляют собой таблицы размера $n*k$ для времен исполнения и $(n+1)*k$ для времен компиляции процедур.

Чтобы получить решение по имеющемуся функционалу, нужно найти его минимум в пространстве назначения линеек. Для поиска минимума функционала по таблицам использовался метод координатного спуска. Чтобы уменьшить вероятность попадания в локальные ямы, дополнительно проводился эксперимент, при котором каждые несколько шагов осуществлялся шаг в рандомном направлении. Кроме того, проводились спуски из крайних точек конфигурационного пространства назначения линеек, то есть из точек, в которых всем процедурам назначается одна и та же линейка. Во всех случаях результат решения совпадал и достигался не более, чем за *3*n* шагов.

Наиболее интересные результаты были получены в точке минимума функционала (14) при значениях параметров <r=7, j=1> . На Рисунке 13 показано сравнение теоретического и реального сокращения времени исполнения при использовании последовательностей соответствующей точки минимума функционала, а сравнение изменения расчетного и актуального изменения времени компиляции проиллюстрировано на Рисунке 14.

На задаче 179.art было получено ускорение в 3,76 и 3,81 раз соответственно, при иллюстрировании оно было ограничено значением 2 (100%) для наглядности результатов остальных задач. Среднее ускорение приложений составило 15,2% и 12,3% соответственно. В некоторых работах по

итерационным решателям отмечается, что замеренное влияние изменения компиляции остальных процедур задачи оказалось незначительным. В целом, вычисленные по таблицам результаты и результаты реального запуска, действительно, соответствуют друг другу, но разница все же заметна. Часть неточности связана с погрешностью профилирования утилитой dprof. Однако, проведенный анализ показал, что большее значение имеет отличие использования кэш-памяти при различной компиляции других процедур. Этот эффект сильней всего проявился на одной из процедур приложении 189.lucas, и был связан пограничными характеристиками для работы памяти.

Основные отличия в оценках изменения времени компиляции связаны с тем, что для большого количества процедур время их компиляции на отдельных этапах мало (1-5 микросекунд), поэтому вычисляется с погрешностью.

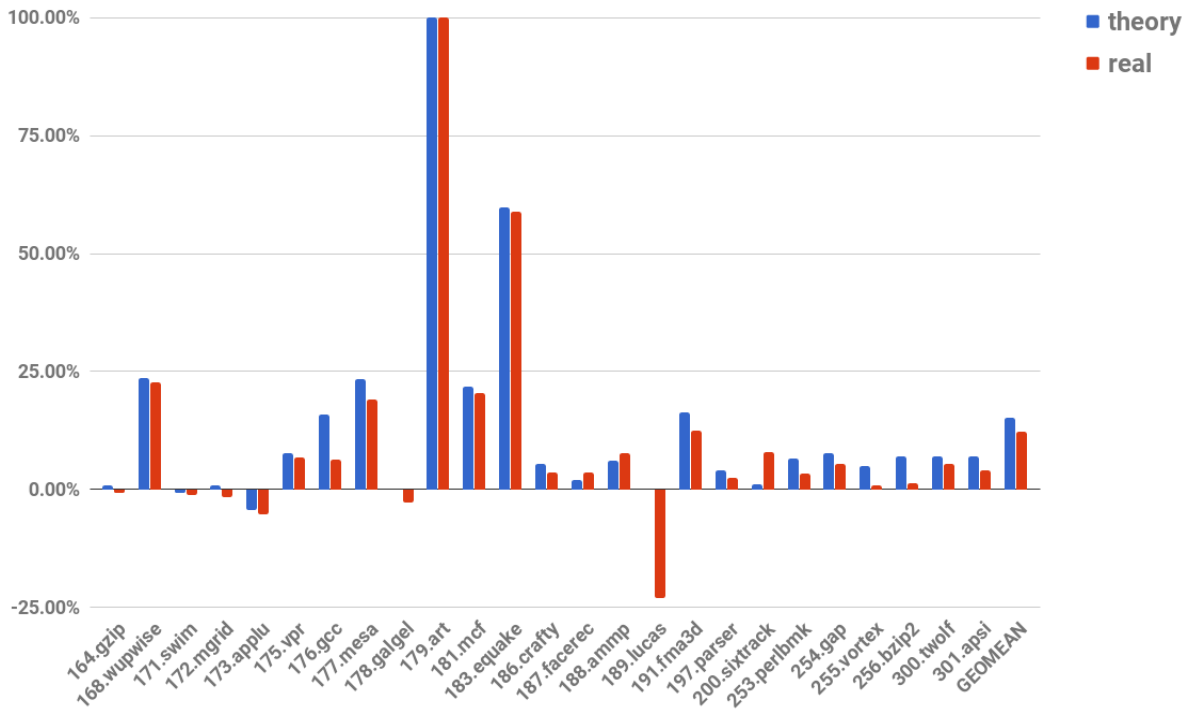

Рисунок 13. Теоретическое и реальное изменение времени исполнения (*), функционал (14), <r=7, j=1>, 100% означает ускорение времени компиляции в 2 раза.

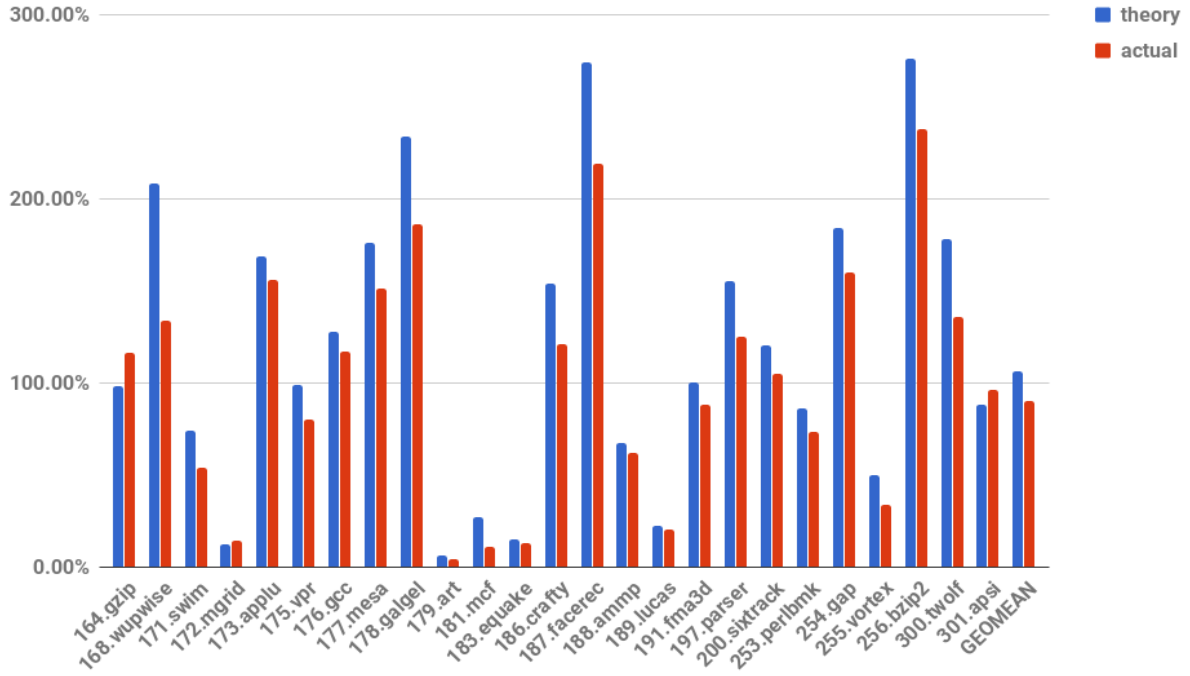

Рисунок 14. Теоретическое и реальное изменение времени компиляции, (14), <r=7, j=1>.

3.6 Выводы

В Главе 3 решена задача формирования числовой характеристики, или функционала, позволяющего оценить качество компиляции. Предлагаемая характеристика применима как для задачи апостериорного выбора, то есть для точного выбора по уже имеющимся оценкам или значениям характеристик компиляции, так и для итоговой оценки качества априорного выбора, то есть осуществленного в условиях отсутствия значений соответствующих характеристик по другим признакам. Для апостериорного выбора построенный функционал качества компиляции был использован как оценка при преобразовании подстановки процедур [29]. В случае же задачи машинного обучения по выбору оптимальной последовательности оптимизации его использование предлагается для априорного выбора.

При построении функционала качества компиляции обоснованы три важных свойства для случая поиска оптимума по одной характеристике

оптимизации и два свойств многокритериальной оптимизации. Для многокритериальной оптимизации доказаны две теоремы, одна из которых позволяет осуществить переход с поиска точек на границе Парето к поиску минимума функционала, а другая позволяет однозначно определить вид функционала, как функции от нескольких критериев. Кроме того, описано множество точек Парето-минимумов, достижимых для какого-то функционала из построенного семейства.

На примере пакетов приложений CINT spec2000 и CFP spec2000 показана работа построенного функционала качества.

Глава 4 Машинное обучение для неаддитивной по объектам ошибки

4.1 Задача минимизирующей классификации по функционалу

Машинное обучение представляет собой выявление общих закономерностей по частным эмпирическим данным [54], [55].

Данные содержат множество объектов *X*. Объекты при этом описываются набором признаков, то есть фиксируется некоторое количество K показателей, измеряемых для всех объектов. Можно считать, что каждый признак это отображение $f : X \to D_f$, где D_f - множество допустимых значений признака. То есть признаковое описание объекта $x \in X$ — это вектор $(f_1(x), \ldots f_r(x))$ размерности K. При этом каждый признак может быть числовым $D_f = R$, бинарным $D_f = \{0,1\}$ или номинальным.

В зависимости от задачи может требоваться либо выявить зависимость между объектами, то есть решить задачу обучения без учителя (unsupervised learning), либо может быть поставлена задача обучения с учителем (supervised learning) [56] [57] [58]:

1) Обучение с учителем (supervised learning)

В этом случае помимо множества объектов X определено множество допустимых ответов Y, при этом есть целевая функция

$$y^*: X \to Y ,$$

значения которой известны только на конечном подмножестве объектов. Соответствующие пары $(x_i, y^*(x_i))$ называют обучающей выборкой X^l. Задача машинного обучения в этом случае заключается в том, чтобы по обучающей выборке построить решающую функцию, имеющую эффективную компьютерную реализацию, или алгоритм, $a: X \to Y$, которая приближала бы целевую функцию не только на объектах обучающей выборки, но и на всем X.

Для оценки качества построенного алгоритма используют функционал качества [58], который определяется, как среднее функций потерь, то есть ошибок алгоритма на объектах выборки:

$\lambda(a, x) \geq 0$ - функция потерь или ошибка алгоритма a на x (loss function) [59], при $\lambda(a, x) = 0$ алгоритм называется корректным на x,

$$Q(a, X^l) = \frac{1}{l} \sum_{x \in X^l} \lambda(a, x)$$

- функционал средних потерь или эмпирический риск.

Классический метод обучения заключается в минимизации эмпирического риска (ERM), то есть в поиске алгоритма, обеспечивающего минимальную среднюю потерю. Обычно с этой целью выбирается некоторая модель алгоритмов с регулируемыми параметрами. После чего производится процесс настройки параметров по обучающей выборке. Чаще всего анализируется не одна, а несколько моделей, из которых выбирается наиболее эффективная.

Задачи обучения с учителем обычно разделяют на следующие основные категории:

а) Классификация на непересекающиеся классы [60] - требуется выбрать один наиболее подходящий ответ из конечного множества допустимых

объектов.

б) Классификация на пересекающиеся классы — требуется для каждого класса определить, принадлежит этому классу объект или нет.

в) Регрессия - требуется указать для объекта действительное число или числовой вектор.

2) Обучение без учителя (unsupervised learning). К этому типу задач относятся как задачи обучения, так и разного рода задачи по удалению избыточных и недостоверных обучающих данных. Основное отличие указанных методов обучения без учителя заключает в отсутствии множества ответов и соответственно отсутствии значений целевой функции. Анализируются только сами объекты по имеющимся признакам.

Для решаемой задачи выбора набора оптимизаций с настройками имеется следующее :

Пусть P - *множество всевозможных процедур,*

$L=(l_1,...l_k)$ - *конечное множество наборов оптимизаций с настройками, или линеек,*

$L^*:\{P\to L\}$ - *всевозможные отображения из множества процедур в множество линеек, и есть функционал качества компиляции,*

$F(l^*,P):L^*\to R$, *значения которого известны на некотором конечном подмножестве процедур* $P_t\subset P$.

Требуется найти алгоритм $a:P\to L$ *такой, который приближал бы* F *к минимальному значению не только на* P_t , *но и на всем множестве* P .

Качество алгоритма оценивается по его работе на обучающей выборке $P_t\subset P$: $F(a,P_t)$.

Замечание: 1) Поскольку значение функционала качества всегда вычисляется для конечного подмножества множества процедур $p_{1,}\dots p_n$, то его удобно представлять как $F:L^n\to R$ - функционал на множестве всевозможных вариантов назначения ответов L объектам из P, где

$F(l^*(p_1),\dots l^*(p_n))=F(l^*,\{p_{1,}\dots p_n\})$.

2) Процедура $p\in P$, помимо собственного кода, определяется вызывающим ее приложением и входными данными для его исполнения, поскольку при их отличии соответствующие значения функционала будут разными.

Поскольку L конечно, то выбор наиболее эффективного для процедуры набора можно рассматривать как задачу классификации. При этом, как и в задаче обучения с учителем, требуется минимизировать функцию потерь, которую можно определить как

$$Q(a,P)=F(a,P)-F_{min}.$$

Построенная функция потерь будет принимать значение 0 как раз в точке минимума функционала.

Однако все равно остаются существенные отличия от обычной задачи обучения с учителем, связанные *с не аддитивностью по объектам* имеющейся функции потерь.

Во-первых, не определены верные классы для объектов. Если бы функция потерь была аддитивной, то минимумы функционала определяли бы и верные классы для объектов. В случае же неаддитивной функции могут быть найдены вектора, соответствующие назначению классов всем объектам одновременно, но не класс для каждого объекта в отдельности.

Пример 5: Чтобы это проиллюстрировать, достаточно рассмотреть одно приложение с двумя процедурами, $L=\{l_{1,}l_2\}$, функционалом (14) с r=1, j=1:

$$F\left(l^*(p_1),\, l^*(p_2)\right)=$$
$$geomean((exe(p_{1,}l^*(p_1))+exe(p_{2,}l^*(p_2))),(comp(p_{3,}l^*(p_3))+comp(p_{4,}l^*(p_4))))$$

и таблицей

	$comp(-,l_1)$	$comp(-,l_2)$	$exe(-,l_1)$	$exe(-,l_2)$
p_1	10	1	1	10
p_2	10	1	1	10

Таблица 5: Пример разных оптимальных наборов оптимизаций.

Тогда в точках $(l_{1,}l_1)$ и $(l_{2,}l_2)$ будет достигаться минимум функционала, а в точках $(l_{1,}l_2)$ и $(l_{2,}l_1)$ - нет. То есть каждая линейка может быть оптимальной для процедуры, но только при определенном выборе линейки для другой процедуры.

Во-вторых, нельзя вычислить ошибку алгоритма для отдельного объекта. То есть принятие решения на одном объекте меняет веса ошибок на других объектах. Указанное отличие является значительным, поскольку не позволяет без модификаций применять описанные в литературе модели минимизации функционала потерь - метод опорных векторов (SVM, support vector machine), Байесовские сети [61], линейные классификаторы, такие как бустинг, нейросети. Параметры соответствующих моделей настраиваются с целью уменьшения значения функции потерь для каждого класса или даже для каждого объекта в отдельности.

Сформулируем описанную задачу *минимизирующей классификации по не аддитивному функционалу потерь* в обозначениях задачи машинного обучения с учителем.

Определение: Определим задачу *оптимизирующей классификации* следующим образом. Пусть дано множество объектов *X,* множество допустимых ответов *Y,* а $Y^*: X \to Y$ - всевозможные отображения из *X* в *Y*. И пусть есть функционал

$F: X \times Y^* \to R$, значения которого известны только на обучающей выборке $X^l \times Y^* \subset X \times Y^*$. Требуется найти алгоритм $a: X \to Y$ такой, чтобы значение функционала F минимизировалось на всем множестве X. Для оценки качества построенного алгоритма используется функционал потери, означающий отклонение функционала от минимума на объектах обучающей выборки $X^l \subset X$, то есть значение:

$$F(a) = F\left(X^l, a\left(X^l\right)\right) - \min_{(y^* \in Y^*)}\left(F\left(X^l, y^*\right)\right).$$

То есть решаемая задача машинного обучения с целью выбора эффективной оптимизационной последовательности представляет собой задачу оптимизирующей классификации и ее решение разбивается на две основные части:

1) Адаптировать существующую или разработать альтернативную модель классификации, которая позволяет минимизировать неаддитивный по отдельным объектам функционал потерь.

2) Выделить показательный набор характеризующих процедуры признаков, доступных на раннем этапе компиляции, то есть сформировать их признаковое описание.

4.2 Классификация с неаддитивной по объектам функцией потери

В этом разделе будет предложен алгоритм *построения последовательности закономерностей*, позволяющий решить задачу классификации с неаддитивной по объектам функцией потери.

4.2.1 Логические алгоритмы классификации и относительная информативность правил

Предлагаемый далее метод можно отнести к логическим алгоритмам классификации [62]. При машинном обучении такого типа разрабатывается

набор правил, то есть логических формул или предикатов, позволяющий отделять объекты одного класса от объектов других классов. То есть набор отображений вида:

$$\varphi(x): X \to \{0, 1\} \textit{ на множестве объектов } X$$

Говорят, что φ *покрывает* объект x, если $\varphi(x)=1$. Чтобы оценить информативность предиката φ по отношению к некоторому классу объектов с а Y в логических алгоритмах, как правило, используются соотношения следующих характеристик [63]: Pc — число объектов класса с в выборке P; $pc(\varphi)$ — из них число объектов, для которых выполняется условие $\varphi(p)=1$; Nc — число объектов всех остальных классов в выборке P; $nc(\varphi)$ — из них число объектов, для которых выполняется условие $\varphi(p)=0$.

Чтобы оценивать качество правила при минимизации функционала, определим *относительную информативность правила по функционалу*, то есть информативность правила в условиях уже имеющегося разбиения на классы, следующим образом.

Определение: Пусть все объекты уже отнесены к некоторым классам, тогда *относительная информативность* правила или предиката φ для класса c вычисляется как изменение значения функционала F при назначении класса c объектам, которые покрываются правилом φ. Для использования относительной информативности необходимо назначить всем объектам некоторый исходный класс, для определенности будем его называть *базовый класс*.

Замечание: Для задачи назначения последовательности оптимизаций естественно в качестве базового класса выбрать оптимизационную последовательность, соответствующую -O3, как наиболее оптимальную в среднем.

Чтобы построить предикаты или правила φ по имеющимся количественным или номинальным признакам, требуется их бинаризовать [62].

Каждый признак порождает целое семейство предикатов, которые проверяют попадание значения признака в некоторое подмножество допустимых значений признака *D.* Самые типичные предикаты такого рода:

- Если f — номинальный признак:

  $\varphi(x)=[f(x)=d], d\ a\ D$;

  $\varphi(x)=[f(x)\in D'], D'\subset D$.

- Если f — количественный (или порядковый) признак:

  $\varphi(x)=[f(x)\le d], d\ a\ R(или\ N)$

  $\varphi(x)=[d\le f(x)\le d'], d\ a\ R(или\ N), d'\ a\ R(или\ N), d<d'$.

Большинство закономерностей ищутся в виде конъюнкций указанных выше элементарных правил.

Самый простой логический алгоритм классификации с помощью правил — это *решающий список*. Решающий список — это алгоритм классификации *a: X* → *Y*, который задаётся набором предикатов $\varphi_{1,}\dots\varphi_T$, приписанных к классам $c_{1,}\dots,c_T\ a\ Y$, который последовательно проверяет принадлежность к соответствующим классам. Основной проблемой построения алгоритма в виде решающего списка является задача выбора предикатов для классов. Использование только конъюнкций элементарных предикатов не позволяет построить эффективный решающий список, а поиск их дизъюнкций является значительно более сложной вычислительной задачей.

Построение *бинарных деревьев* позволяет неявным образом искать дизъюнкции элементарных конъюнкций. В этом случае каждой внутренней вершине $v\in V$ приписан $\varphi_i: X\to\{0,1\}$, а каждому листу дерева (конечной вершине) $v\in V$ приписано имя класса $cv\in Y$. При классификации объект $x\in X$ проходит по дереву путь от корня до некоторого листа. У бинарных деревьев есть ряд недостатков, связанных с безвозвратным разделением

объектов на области. Основной из них — это проблема переобучения, то есть ситуации, когда начинают формироваться правила не под настоящие, статистически значимые, закономерности, а фактически под отдельные объекты выборки [64][65].

Более эффективным подходом, сочетающим преимущества деревьев и решающих списков, является алгоритм взвешенного голосования, позволяющий объединять конъюнкции с помощью весовой функции [66][67]. В этом случае для каждого объекта и правил каждого класса вычисляется значение выражения

$$W(c, x) = \sum a_i \cdot \varphi_i(x) ,$$

где $a_i > 0$ - числовые коэффициенты, φ_i – закономерности класса с. После чего производится выбор класса, соответствующего наибольшему значению суммы.

Одним из наиболее эффективных алгоритмов этого типа является бустинг (boosting) [67], исходно предложенный для объединения результатов нескольких алгоритмов машинного обучения, решающих общую задачу. Идея бустинга заключается в том, что на каждом шаге производится поиск дополнительного правила, уменьшающего ошибку при выделении объектов класса. При этом с целью, чтобы новое правило выявляло новые объекты, искусственным образом увеличиваются веса объектов, на которых была допущена ошибка и уменьшаются веса объектов, на которых было получено верное решение.

4.2.2 Алгоритм классификации для неаддитивного функционала ошибки, теорема сходимости

Для задачи минимизации произвольного неаддитивного функционала можно построить алгоритм, похожий на взвешенное голосование в том смысле, что производится повторный выбор класса для объектов, уже отнесенных к определенному классу. Однако, в этом случае нет возможности менять стоимость ошибки на объектах в зависимости от корректности работы

алгоритма, поэтому новое правило добавляется не с коэффициентом, а просто переопределяет класс для некоторой области. Такой подход можно рассматривать как взвешенное голосование с той особенностью, что вес каждого следующего правила больше, чем суммарный вес всех предыдущих правил.

Чтобы искать новое правило для имеющегося разбиения, предлагается для него искать правило с положительной, желательно максимально высокой, *относительной информативностью по функционалу*.

Алгоритм 2. Построение последовательности закономерностей при классификации по функционалу качества (КФК):

1: Назначить всем объектам базовый класс

2: Подсчитать значение функционала F для обучающей выборки

3: **Повторять** пока значение функционала F не достигло минимума F_{min}

4: найти предикат φ_k с самой высокой положительной относительной информативностью (*) для некоторого класса C, если не удалось найти такой предикат, то **остановка**

5: добавить φ_k в последовательность закономерностей $\varphi_{1,\dots}\varphi_{(k-1)}$

6: назначить всем объектам, для которых $\varphi_k(x)=1$, класс C

7: вычислить новое значение функционала F

8: k++

(*) - предлагаемый алгоритм поиска закономерностей будет описан далее

В результате работы Алгоритма 2 будет сформирована последовательность предикатов $\varphi_{1,\dots}\varphi_K$. Для того, чтобы определить класс объекта x надо найти:

$$\varphi = max\{\varphi_k : \varphi_k(x)=1\}_{k\le K}$$

И назначить объекту x класс C, соответствующий предикату φ.

Опишем теперь алгоритм поиска правила с высокой относительной информативностью. Поскольку полный перебор совокупности конъюнкций всевозможных элементарных правил практически не осуществим из-за их слишком большого количества, то обычно используют различные техники сокращения поиска. Так, для поиска хороших закономерностей в задачах логической с аддитивным по объектам способом вычисления ошибки классификации существует ряд алгоритмов - градиентный, генетические [68], КОРА [69], ТЭМП [70].

В результате исследований и анализа результатов был адаптирован и использован метод градиентного поиска закономерности с двоичным уменьшением размера шага, достаточно быстрый и показавший себя эффективным на решаемой задаче.

Определение: Для описания предлагаемого алгоритма поиска закономерности введем понятие *матрицы частичных ошибок,* содержащую отклонения функционала от минимального значения при изменениях координат одной из точек достижения минимума. Поскольку множество объектов в выборке и множество ответов конечны, то обозначим $|X|=n$, а $|Y|=m$, $x_{1,}\dots x_n$ - все элементы множества X. Для удобства перейдем к векторному описанию отображения y^*:

$$y^*(X)=(y^*(x_1),\dots y^*(x_n))=(c_{1,}\dots c_n)\in Y^n .$$

Без ограничения общности можно считать, что $F_{min} > 0$, в противном случае, т.к. F_{min} вычисляется на конечном множестве точек, все ее значения можно сдвинуть на требуемую величину. По обучающей выборке вычислим одну из точек, на которой функционал качества достигает минимума $F_{\min} >0$:

$F_{min}=F(b_{1,}\dots b_n)=min\limits_{(c_{1,}\dots c_n)\in Y^n}(F(c_{1,}\dots c_n))$, где $(b_{1,}\dots b_n)\in Y^n$ - оптимальный вектор классов, назначенных элементам X. Пронумеруем $y_{1,}\dots y_m\in Y$ - все элементы

множества Y .

Тогда элемент (i, j) матрицы размера $n \times m$ вычисляется как:

$$\partial_{i,j} = \partial(x_i, y_j) = \log\left(\frac{F(b_{1,\dots}, b_{(i-1)}, y_j, b_{(i+1)}, \dots b_n)}{F_{min}}\right)$$

Замечание: Логарифм от отношения использован для того, чтобы при изменении выбора линеек на отдельных процедурах в разных задачах сумма частичных ошибок совпадала частичной суммарной ошибкой. Поскольку в функционале (14) задачи связаны между собой произведением в geomean, то под логарифмом в $\partial_{i,j}$ остается только отношение части функционала для задачи с изменившейся компиляцией одной процедуры и части функционала для нее в его минимуме. Поэтому если процедуры $x_{i1}, x_{i2}, i2 > i1$ принадлежат разным задачам, то

$$\partial(x_{i1}, y_{j1}) + \partial(x_{i2}, y_{j2}) =$$
$$\log\left(\frac{F(b_{1,\dots}, b_{(i1-1)}, y_{j1}, b_{(i1+1)}, \dots, b_{(i2-1)}, y_{j2}, b_{(i2+1)}, \dots b_n)}{F_{min}}\right) = \partial(\langle x_{i1}, x_{i2}\rangle, \langle y_{i1}, y_{j2}\rangle)$$

Следущая лемма требуется для осуществление шага алгоритма.

Лемма: В каждой строке матрицы частичных ошибок есть нулевой элемент:

$$\forall i \exists j : \partial(x_i, y_j) = 0 .$$

Доказательство:

Для доказательства достаточно рассмотреть класс, соответствующий координате объекта x в векторе классов, для которого была построена матрица частичной ошибки.

Алгоритм 3. Двоичный градиентный поиск закономерностей:

1: Пусть $(c_1, \dots c_n) \in Y^n$ текущее разбиение на классы множества X , где c_s соответствует назначению класса элементу номер s . Выбрать объект x_s с

максимальной текущей ошибкой $\partial(x_s, c_s)$, не помеченный флагом. Если таких нет, то **остановить** поиск. Далее, выбираем правило для класса $y_l \in Y : \partial(x_s, y_l) = 0$. Для определенности обозначим текущее значение функционала F_{start}.

Далее правило ищется в виде конъюнкции предикатов $\varphi_i = (th \geq t) \wedge (tl \leq t)$, где t — значение некоторого признака объекта, a th и tl — границы признака для области.

2: **Цикл** по признакам

Вычислить dh как разницу между максимальным значением признака th и его значением ts для x_s. Вычислить dl как разницу между минимальным значением признака tl и его значением ts для x_s.

3: **Цикл** пока уменьшается F

4: Шаг двоичного поиска верхней границы признака — сравнить значение F при использовании в предикате th, (th+dh/2) и (th-dh/2). Присвоить границе th значение, соответствующее минимальному значению F при условии наличия Q объектов в выделенной области.

5: Шаг двоичного поиска нижней границы признака — сравнить значение F при использовании в предикате tl, (tl+dl/2) и (tl-dl/2). Присвоить границе tl значение, соответствующее минимальному значению F при условии наличия Q объектов в выделяемой предикатом области.

6: Присвоить dh значение dh/2, присвоить dl значение dl/2.

7: Для найденного предиката $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_r$ сравнить по значению F его работу для всех классов. Выбрать класс y_{step}, соответствующий получению минимального значения $F_{step} = F$.

8: Если $F_{step} < F_{start}$, то правило φ и класс y_{step} найдены. Иначе отметить объект x_m флагом и **перейти** в 1.

Были также опробованы и подходы с построением нескольких вариантов классов, с выбором лучшего, как это делается, к примеру, в ТЭМП:

1) На каждом шаге строятся правила для нескольких стартовых процедур с наибольшими ошибками, оставляется лучшее.

2) Для каждого класса подбирается свое правило, выбирается лучшее.

На практике результаты таких разбиений с точки зрения скорости уменьшения значения функционала оказались практически идентичны описанному однопроходному выше.

Чтобы уменьшить возможное переобучение, при реализации поиска закономерностей дополнительно в процессе на шаге 4 и 5 ограничивается снизу число объектов, которые покрываются получившемся правилом, некоторым числом Q. Кроме того, на шаге 9 происходит отказ от правила, если уменьшение значения функционала (или *W*) ниже некоторого процента от исходного значения:

$$\frac{dF}{F(c_{start}, \ldots c_{start})} < eps ,$$

то есть построенное правило имеет очень низкий *уровнь значимости*.

Покажем, что предложенные алгоритмы останавливаются и что они решают поставленную задачу классификации.

Теорема 5: Алгоритм КФК с предложенным алгоритмом поиска правил останавливается. Если на каждом шаге получится подобрать правило с положительной относительной информативностью по функционалу, то алгоритм КФК за конечное число шагов приведет к разбиению на классы,

соответствующему нулевой потере.

Доказательство:

Поскольку количество элементов в обучающей выборке конечно и количество классов тоже конечно, то число значений, которое может принимать функционал *F* тоже конечно. Пусть оно равно Nf.

1) Двоичный градиентный поиск правила останавливается, тк на каждом шаге его цикла требуется уменьшение значения F, то есть число итераций цикла не больше Nf.

2) Если на некотором шаге алгоритма КФК не удается найти правило, уменьшающее значение *F*, то алгоритм останавливается, поэтому этапов поиска правил будет не больше Nf. На каждом шаге алгоритма поиска правила значение функционала уменьшается, либо стартовый элемент отмечается флагом и не рассматривается для дальнейшего поиска правила, поэтому шагов алгоритма поиска правила не более n, где n — число объектов обучающей выборки.

3) Пусть на каждом шаге алгоритма КФК удается найти правило, уменьшающее значение *F*. Все получающиеся значения различны, поэтому не больше, чем за Nf шагов функционал достигнет своего минимума.

4.2.3 Переход к аддитивной по объектам частичной ошибке

Рассмотрим теперь перехода к аддитивному функционалу потери, позволяющему искать минимум функционала потери. В этом случае можно было бы использовать обычные методы обучения. Кроме того, такой функционал можно использовать для выхода из локальных минимумом при работе алгоритма КФК.

Далее будет построен аддитивный функционал на основе матрицы частичной ошибки, определенной в предыдущем разделе, и доказан для него ряд утверждений.

Пусть объектам $x_k \in X$ назначен вектор классов $(c_{1,} \ldots c_n)$. И пусть

выбрана точка, в которой функционал F достигает своего минимума. Определим новую ошибку классификации как сумму частичных ошибок W:

$$W(c_{1,}\dots c_n)=\sum \partial(x_l, c_l)\ .$$

Тогда функционал потерь для алгоритма может быть вычислен как:

$$Q(a, X)=\sum \partial(x_l, a(x_l))$$

Чтобы свести задачу к обычной задаче обучения с учителем, достаточно в качестве целевой функции рассмотреть $y^*=\underset{(z^*\in Y^*)}{argmin}\left(F(X^l, z^*)\right)$.

Лемма 4.1: Если минимум единственный, то точная минимизация суммы частичных ошибок совпадёт с результатом точной минимизации функционала.

Доказательство:

Поскольку W=0, то $\forall l, k$ получившегося разбиения на классы $\partial(x_l, c_k)=0$, то есть $\log\left(\frac{F(b_{1,}\dots, b_{(l-1)}, c_k, b_{(l+1)}, \dots b_n)}{F_{min}}\right)=0$, и, соответственно, $F_{min}=F(b_{1,}\dots, b_{(l-1)}, c_k, b_{(l+1)}, \dots b_n)$. Из условия единственности минимума следует, что $c_k=b_k$.

Лемма 4.2: Если минимум неаддитивного функционала потери F единственный и для всех объектов выборки отличается их признаковое описание, то при применении Алгоритма 1 к частичной ошибке возможно выбрать такие предикаты для классов, что за конечное число шагов будет достигнут минимум F.

Доказательство:

С учетом Теоремы 5 и Леммы достаточно показать, что выполним шаг 6 Алгоритма 1, то есть что $\forall x_m, \partial(x_m, c_m)>0$ существует предикат, назначение на котором некоторого класса позволит уменьшить текущую сумму частичных ошибок W. Поскольку по условию признаковые описания всех объектов

отличаются, то для любых двух объектов можно найти признак и его значение, их разделяющее, то есть $\forall x_m, x_l \exists \varphi_l : \varphi_l(x_m)=1, \varphi_l(x_l)=0$. Рассмотрим $\varphi_m = \varphi_1 \wedge ... \wedge \varphi_{m-1} \wedge \varphi_{m+1} ... \wedge \varphi_n$ и назначим ему класс b_m точки минимума. Тогда новая сумма частичных ошибок будет равна W_{new}, где $W - W_{new} = \partial(x_m, c_m) > 0$.

Замечание:

1) Если минимум функционала не единственный, то минимум частичной ошибки может не быть минимумом функционала. Пусть таблица построена для минимума $(y_{1,} y_1)$, тогда частичная ошибка в точке $(y_{2,} y_2)$ равна $W(y_{2,} y_2) = \partial(x_{1,} y_2) + \partial(x_{2,} y_2) = 0$, однако $F(y_{2,} y_2) = 2 > 1$.

$F(y^*(x_1), y^*(x_2))$	y_1	y_2
y_1	1	1
y_2	1	2

Таблица 6. Пример матрицы частичных ошибок. Столбец соответствует значению y^* для x_1, строка — для x_2.

Такой проблемы можно избежать, если скорректировать таблицу, поставив вместо нулей, не соответствующих основному минимуму, некоторые положительные значения. Однако, это может привести к тому, что исходно важные признаки окажутся неинформативными.

2) Стоит отметить, что при этом отдельный шаг классификации в сторону уменьшения частичной ошибки может приводить к увеличению значения функционала потери. Однако это свойство позволяет временно заменять исходный функционал на частичную ошибку в алгоритме КФК, чтобы выйти из локальных минимумов.

4.3 Создание и отбор признаков процедур

Достаточно часто объекты исходно имеют признаковое описание, то есть начальная информация представляет из себя таблицы числовых значений. В противном случае первоочередной задачей является выбор и вычисление стартового набора числовых характеристик объектов. Качество ее решения значительно влияет на качество итоговой модели машинного обучения, в то же время, она является в значительной мере творческой и не имеет четких методологических описаний. Как правило, лучших результатов позволяет добиться хорошее знакомство с предметной областью, для которой строится модель [71].

Доступная для анализа информация о процедуре в процессе компиляции представляет собой ее промежуточное представление — граф управления с узлами, дугами перехода между ними, как прямыми, так и обратными, операции, разного рода зависимости между операциями. Кроме того, имеется информация о профиле исполнения процедуры, то есть значения счетчиков числа исполнения узлов и дуг управляющего графа. Поскольку основной целью является выбор набора оптимизаций, которые были бы эффективны для анализируемого контекста, а также отказ от сложных оптимизаций на простых или редких по исполнению контекстах, то, помимо частных характеристик кода, есть основания рассмотреть несложно вычислимые характеристики, используемые рядом преобразований для принятия решения о применении.

В связи с этим были предложены следующие характеристики кода:

1) Количество операций в процедуре $ProcLen = \sum_i Opers(Node_i)$.

2) Оценочное время исполнения процедуры или размер процедуры.

Самый простой вариант такой оценки — это подсчет числа операций с учетом значений счетчиков числа исполнения линейных участков (или узлов управляющего графа.

$$ProcSize = \sum_i Counter(Node_i) * Opers(Node_i)$$

Так же была проанализирована более сложная уточненная оценка для *ProcSize,* которая была разработана в компиляторе Эльбрус для оценки эффективности проведения оптимизации подстановки процедур (*inline*). Она вычисляется по большему набору характеристик кода и учитывает некоторые дополнительные возможности архитектуры [29].

3) Плотность процедуры

$$ProcDensity = ProcSize / ProcLen$$

4) Количество различных операций (типов опреаций) в процедуре - число вызовов процедур, количество операций с плавающими вычислениями, количество операций взятия адреса, количество операций чтения и записи.

5) Количество различных операций из 4) с учетом значений счетчиков их числа исполнения.

6) Максимальная глубина вложенности циклов.

7) Среднее количество операций в узлах управляющего графа процедуры, которое указывает на степень его ветвлений:

$$AverOpers = ProcLen / NodesNum$$

где NodesNum — число узлов в процедуре

В случае, когда числовые характеристики уже имеются, создание и отбор признаков подразумевает под собой подбор показательных функций их подмножеств и отсев лишних признаков (features selection) [72] [73]. Под лишними признаками подразумеваются такие, которые либо значительно коррелируют с другими признаками и потому не несут дополнительной информации, либо обнаруживают слабую зависимость с целевой функцией, то есть малоинформативны.

Для отсева похожих, то есть коррелирующих признаков, достаточно подсчитать корреляционную матрицу их значений на тренировочной выборке.

Зависимость с целевой функцией определялась как качество классификации с точки зрения минимизации функционала по отдельному признаку.

4.4 Оценка качества предложенного алгоритма

С целью обучения компилятора использовались пакеты spec2000 CINT и spec2000 CFP [18], использующиеся для тестирования производительности процессоров. В совокупности они представляют собой минимальную подборку различного типа реальных приложений, таких как компилятор, задача сжатия, обработка текста, игра в шахматы, метеорологическая задача распределения загрязнений, задачи обработки и распознавания изображений, задача жидкостной динамики, задача теории чисел проверки числа на простоту и так далее. Такой состав пакета позволяет сделать предположение о возможности качественного обучения для дальнейшего применения полученного решения на различных задачах пользователей. В то же время, смысловая минимальность пакета не позволяет делить его на части с целью обучения на одной части и проверки на другой. В связи с этим большинство результатов были получены на одном, на момент проведения экспериментов самом современном доступном пакете такого типа. Дополнительная проверка проводилась с использованием предыдущих пакетов тестирования производительности spec95 CINT, spec95 CFP, решающих отличающиеся задачи на похожие тематики. С целью уменьшения возможности переобучения было использовано небольшое число признаков для обучения (7), построено лишь небольшое количество областей в пространстве параметров, кроме того, вводилось ограничение на минимальное количество процедур в каждой выделенной (выпуклой) области.

С целью оценки качества классификации было проведено сравнение исполнения теоретического оптимального решения и исполнения задач, которые были скомпилированы в соответствии с последовательностями, получающимися после выделения классификаций первых 7-ми областей в

пространстве признаков процедур. Для сравнения был использован функционал (14) при <r=1, j=0>, то есть функционал, который учитывает только время исполнения. Как показано на Рисунке 15, проведенная классификация с последовательным применением 7-ми правил, на обучающей выборке позволила получить большую часть эффекта (88%) по приросту производительности.

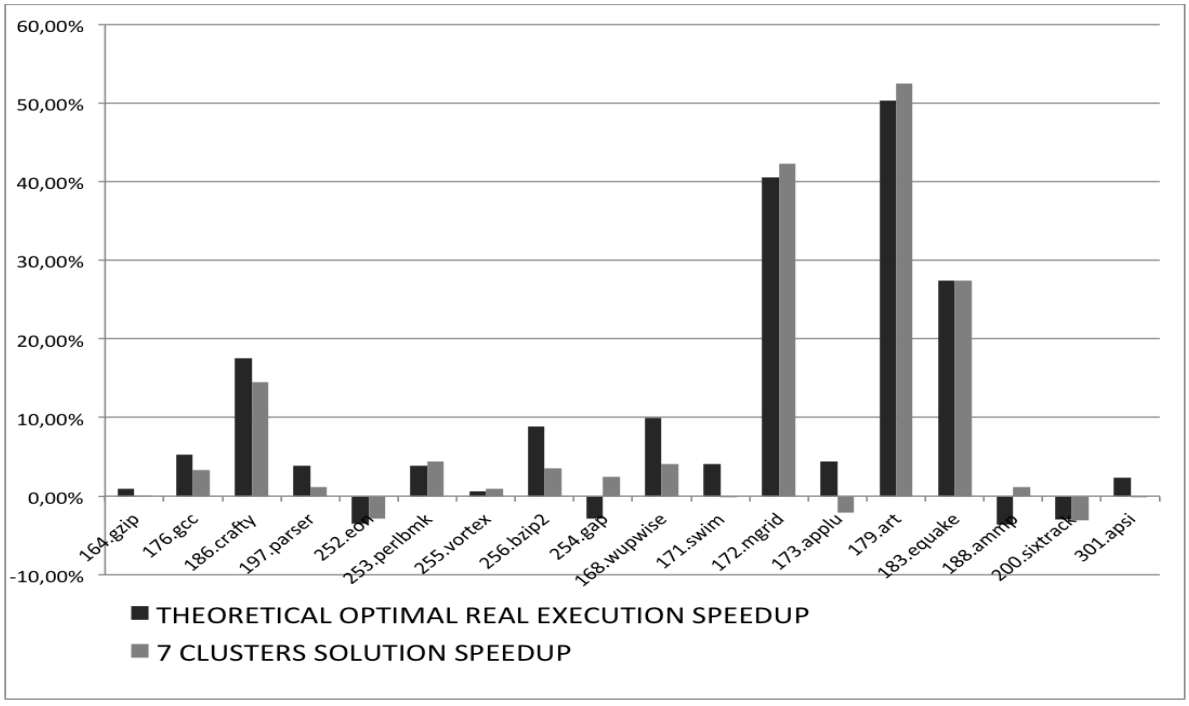

Рисунок 15. Сравнение исполнения в точке теоретического минимума функционала и при классификации алгоритмом.

Получить настолько близкий эффект при одновременной минимизации времени компиляции не удается, что связано с особенностями обучающей выборки. Так, по табличным данным чуть менее 90% процедур обучающей выборки имеют время исполнения 0 из-за того, что они или не вызываются при заданных входных данных, либо имеют настолько небольшое время исполнения, что в соответствующие им такты не попадает утилита dprof. В результате, при поиске теоретического минимума к ним была применена последовательность с самой быстрой компиляцией. Однако, как показал анализ кода приложений, достаточно часто отсутствие вызовов процедур связано не с

тем, что процедура предполагается не исполняемой, как процедуры обработки ошибки, а с тем, что входные данные не обеспечивают полное покрытие вариантов исполнения приложения. Поэтому, с целью уменьшения влияния не полной представительности входных данных, при обучении были использованы только те процедуры, для которых время исполнения при компиляции хотя бы одной последовательностью оказалось положительным.

4.5 Экспериментальные результаты многокритериальной классификации процедур

Задача многокритериальной минимизирующей классификации решалась с использованием функционала (13) при значениях параметров r=7, j=1. С помощью описанного алгоритма выделения областей для фиксации значений линеек было построено 10 областей с дополнительным требованием наличия не менее q=10 элементов в каждой. На Рисунке 16 показан результат применения СНОП, представленный в сравнении с базовым режимом компиляции.

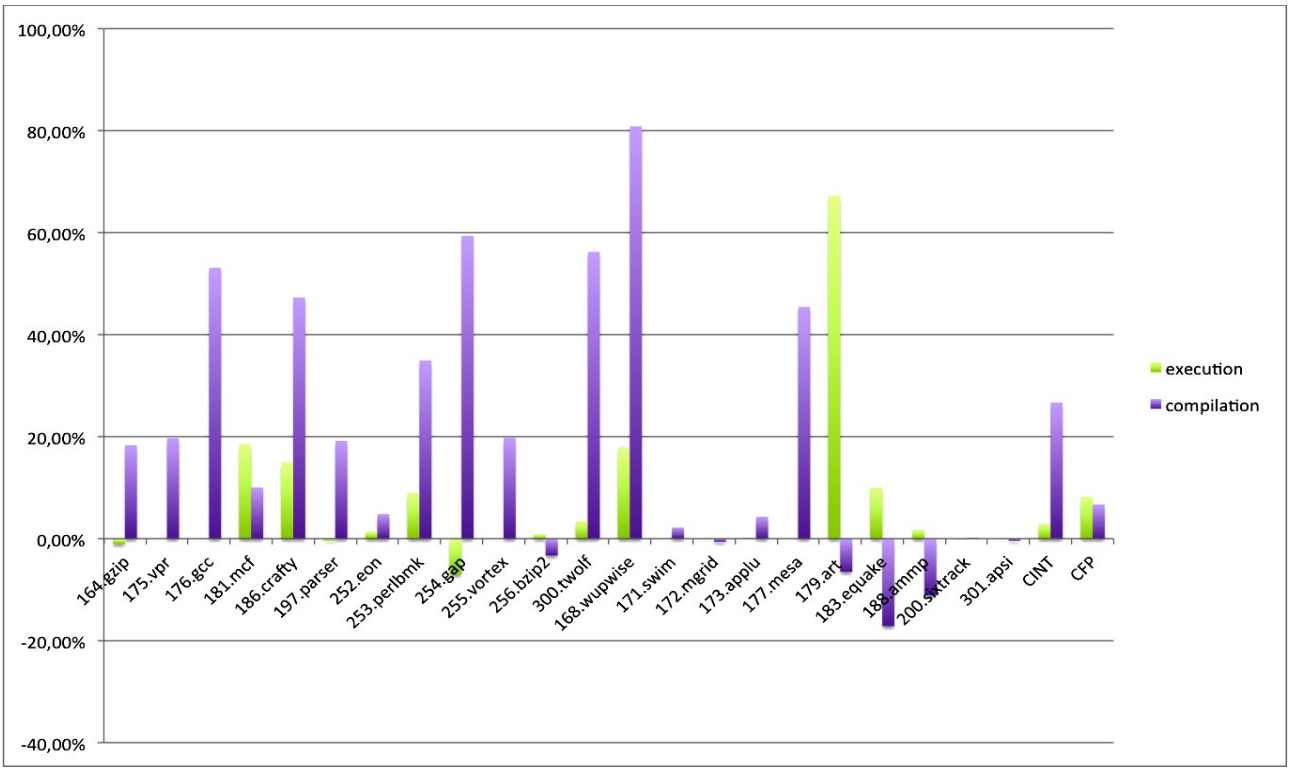

Рисунок 16. Сравнение режима компиляции СНОП с базовым режимом компиляции: execution – время исполнения, compilation – время компиляции, CINT – среднее для задач с целыми вычислениями, CFP – среднее для задач с плавающими вычислениями

Для задач с целыми вычислениями (пакет CINT) было получено среднее ускорение времени исполнения на 2,9% и времени компиляции на 26,7%. Основной эффект достигался за счет сокращения объема работы, увеличивающей размер кода и повышающей спекулятивность оптимизаций на больших процедурах с большой степенью ветвлений. Задачи пакета CFP, т.е. задачи с плавающими вычислениями, в среднем ускорились на 8,2% по исполнению, причем среднее время компиляции сократилось на 6,7%. В этом случае основное ускорение связано с полученной возможностью применить оптимизации работы памяти, которые увеличивают время планирования, но позволяют существенно уменьшить количество блокировок по чтению.

Опубликованный в работе [6] эксперимент, проиллюстрированный на Рисунках 17 и 18, был поставлен с проверкой на пакете spec95, не включенном в тренировочную базу. В этой работе для всего пакета приложений был использован функционал (13) с r=3, j=1. Как для обучения, так и для проверки необходимо использовать взвешенный пакет разносторонних приложений. Такой подход к проверке отсутствия переобучения равносилен применению техники LOO (Leave One Out), но подходит для еще меньших обучающих данных. В результате применения алгоритма классификации на разработанных линейках на основе процедур пакета spec2000 были выделены 5 областей для фиксации значений линеек. Применение линеек, соответствующих областям, дало 17% ускорения компиляции и 8,5% ускорения производительности в среднем на задачах spec2000. В результате использования выделенных областей и линеек для компиляции проверочного пакета spec95 в среднем на 16% ускорилась компиляция и на 3% уменьшилось время исполнения. Меньшее ускорение исполнения приложений пакета связано с меньшим потенциалом пакета по ускорению с точки зрения рассматриваемого компилятора и архитектуры. Однако важным является полученное значительное ускорение компиляции при одновременном ускорении производительности. Стоит отметить, что отличие результатов этого эксперимента на пакете spec2000

связано с отличающимися версиями компилятора, использовании другого функционала и другими оптимизационными линейками.

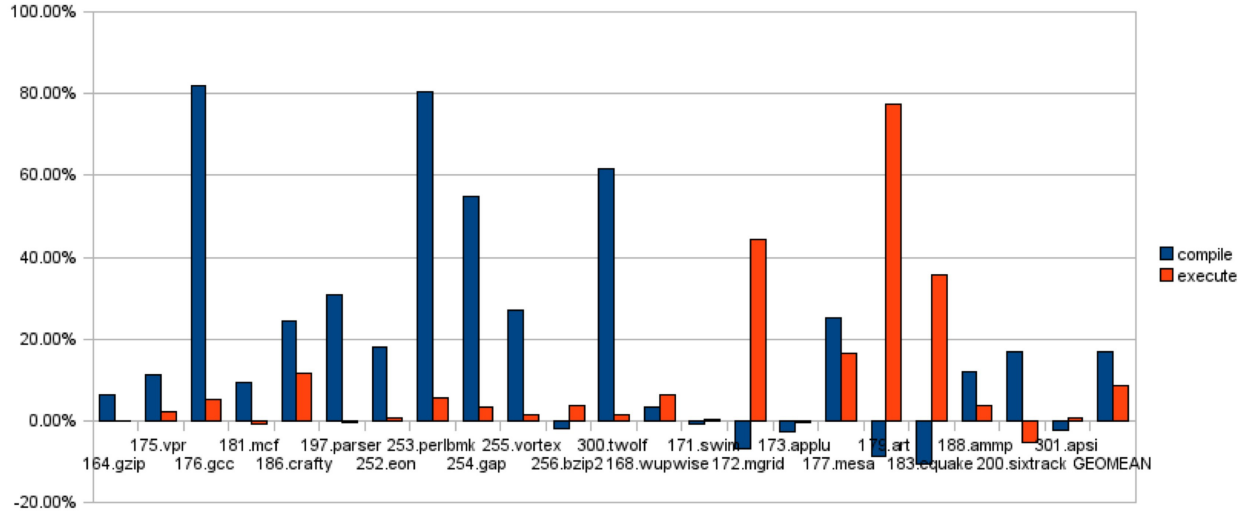

Рисунок 17. Использование линеек для выделенных алгоритмом КФК для частичной ошибки 5-ти областей на тренировочном пакете spec2000.

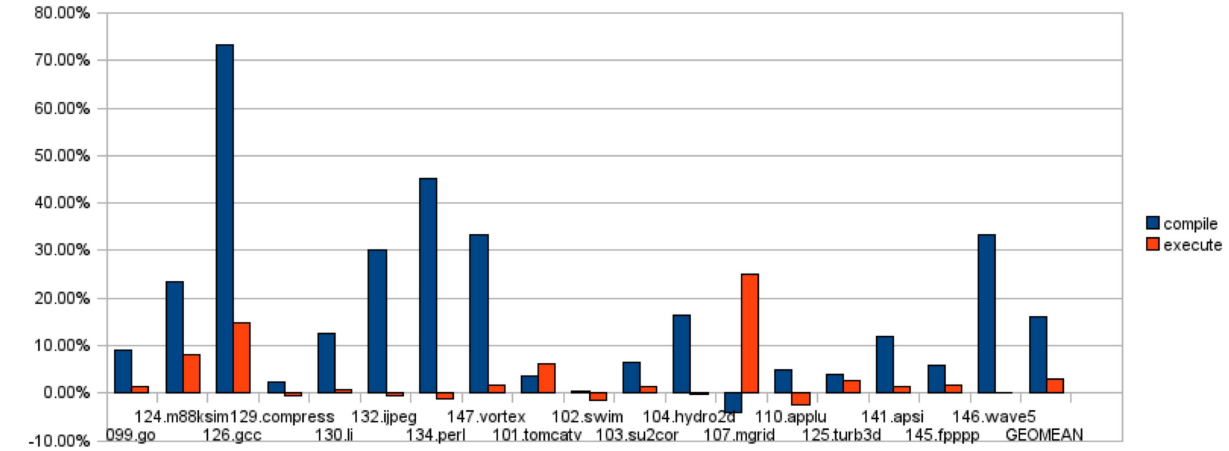

Рисунок 18. Использование линеек для выделенных алгоритмом КФК для частичной ошибки 5-ти областей на проверочном пакете приложений spec95.

4.6 Выводы

В главе 4 ставится задача поиска алгоритма машинного обучения с целью минимизации неаддитивного по объектам функционала качества, каким, как было показано в главе 3, является функционал качества компиляции.

Проведенный анализ описанных в литературе моделей машинного обучения показал, что без модификаций они не могут быть использованы для ее решения. В связи с этим предложены две альтернативные модели - алгоритм построения последовательности закономерностей при классификации по функционалу качества, улучшающий на каждом шаге решение, и алгоритм минимизации частичной ошибки, позволяющий продолжать поиск решения даже в случае локальных ям. Для обоих алгоритмов доказаны теоремы об их сходимости.

Предложенная модель была использована при построении алгоритма назначения оптимизационной последовательности по вычислимым на раннем этапе компиляции признакам процедур. Его применение привело к уменьшению времени исполнения приложений пакета spec2000 CINT в среднем на 2,9%, а приложений пакета CFP spec2000 - в среднем на 8,2%. При этом время компиляции приложении указанных пакетов сократилось на 26,7%. и 6,7% соответственно. Режим компиляции с использованием предложенного алгоритма автоматического выбора оптимизационной последовательности был внедрен в компилятор для архитектуры Эльбрус в качестве одного из базовых режимов.

Поставленная задача машинного обучения, как минимизация неаддитивного функционала качества, и построенные модели ее решения могут быть использованы в различных задачах классификации, для которых принятие решения на одном объекте меняет вес ошибки на других объектах.

Заключение

В диссертационной работе рассматривается проблема приближения качества компиляции в режиме по умолчанию к пиковой настройке компиляции для отдельных приложений с одновременным контролем времени компиляции и других характеристик.

Для ее решения предложено машинное обучение компилятора с целью попроцедурного выбора последовательности оптимизаций с настройками.

Однако описанный в научной литературе вероятностный подход машинного обучения по результатам исполнения приложений целиком, показал отрицательный результат для попроцедурного варианта. Кроме того, проведенный анализ ряда методов пиковой настройки компиляции, необходимых для построения эффективных оптимизационных последовательностей, показал их неприменимость для режима без профиля.

В процессе исследования и в ходе решения поставленных задач были получены следующие основные результаты:

1. Предложены преобразования частичной раскрутки критических путей и длин рекуррентностей цикла, позволяющие лучше спланировать сложные циклы с разновесными ветвлениями. Для обоих методов доказаны утверждения об оценке их эффективности.
2. Предложены неагрессивные методы оптимизации работы с памятью, применимые в режиме по умолчанию: метод увеличения дистанции от чтений с ограничением планируемого времени исполнения цикла и метод построения подкачки структур в циклах.
3. Предложен механизм автоматического выбора оптимизирующей последовательности преобразований для каждой процедуры на раннем этапе компиляции.
4. Предложена числовая многокритериальная оценка качества компиляции, учитывающая попроцедурный характер оптимизации. Показано ее преимущество по сравнению с вероятностной при решении задач машинного обучения компиляторов.
5. Сформулирована и доказана теорема о представлении всех функций двух переменных, обладающих свойством сохранения порядка при растяжениях. Доказано утверждение о представлении функционала качества компиляции, учитывающего время исполнения и время компиляции.

6. Сформулирована задача минимизирующей классификации, соответствующая задаче машинного обучения компилятора с построенным функционалом качества. Предложены методы решения задачи минимизирующей классификации, доказаны утверждения об их сходимости.

Представленные методы использованы при разработке оптимизирующего компилятора с языков C, C++ и Фортран для микропроцессоров «Эльбрус». Суммарное ускорение от совокупности примененных методов в базовом режиме компиляции составило в среднем около 6,1% для задач с целыми вычислениями и около 15% для задач с плавающими вычислениями. При этом время, требующееся для компиляции приложений с целыми вычислениями, сократилось больше, чем на 20%, а с плавающими - больше, чем на 6%.

Разработанные методы оптимизации могут быть адаптированы и использоваться в компиляторах для различных микропроцессоров. Построенный функционал качества компиляции применим для апостериорного и априорного сравнения результатов оптимизации кода по нескольким критериям одновременно.

Постановка задачи обучения, как минимизирующей классификации, и предложенные алгоритмы ее решения могут быть использованы в различных задачах классификации, для которых принятие решения на одном объекте меняет вес ошибки на других объектах.

Список литературы

[1] Четверина О.А., "Сравнение и выбор эффективной стратегии компиляции". Труды 56-й научной конференции МФТИ, Радиотехника и кибернетика. - М.:МФТИ, 2013. С. 66-68.

[2] Четверина О.А., "Статический выбор эффективной стратегии оптимизации кода". Национальный Суперкомпьютерный Форум НСКФ-2013, http://2013.nscf.ru/nauchno-prakticheskaya-konferenciya/tezisy-dokladov/, 2013.

[3] Четверина О.А., Нейман-заде М. И., Степанов П. А. "Система направленной оптимизации (СНОП)". I Всероссийская научно-техническая конференция "Расплетинские чтения", 2014. С. 190-191.

[4] Четверина О.А., Степанов П. А., Нейман-заде М. И., "Автоматическая направленная оптимизации процедур (СНОП)". Вопросы радиоэлектроники. 2015. № 3. С. 64-76.

[5] В.Ю. Волконский, А.В.Брегер, А.Ю.Бучнев, А.В.Грабежной, А.В.Ермолицкий, Л.Е.Муханов, М.И.Нейман-заде, П.А.Степанов, О.А.Четверина, "Методы распараллеливания программ в оптимизирующем компиляторе". Вопросы радиоэлектроники. 2012. Т. 4. № 3. С. 63-88.

[6] O.A. Chetverina, “Procedures classification for optimizing strategy assignment”. Proceedings of the Institute for System Programming. Volume 27 (Issue 3), 2015.

[7] Четверина О.А., «Сравнение использования различных функционалов в пространстве назначения стратегий оптимизации», II Международная конференция Инжиниринг & Телекоммуникации — En&T 2015, Тезисы докладов, 2015. С. 201-202.

[8] А. В. Ермолицкий, М. И. Нейман-заде, О. А. Четверина, А. Л. Маркин, В. Ю. Волконский, «Агрессивная инлайн-подстановка функций для VLIW-архитектур». Труды Института системного программирования РАН. 2015. Т. 27. No 6. С. 189-198.

[9] Четверина О.А., «Методы коррекции профильной информации в процессе компиляции». Труды Института системного программирования РАН, Т. 27, № 6, 2015. С. 49-66.
[10] Четверина О.А., «Повышение производительности кода при однофазной компиляции». Программирование, 2016, № 1. С. 51-59. (O. A. Chetverina, Alternatives of profile-guided code optimizations for one-stage compilation, Programming and Computer Software,January 2016, Volume 42, Issue 1, pp 34–40.)
[11] Четверина О.А., Нейман-заде М.И., Степанов П.А., Линчик М.И. "Система выбора и использования индивидуальных последовательностей оптимизаций для компиляции процедур оптимизирующим компилятором под архитектуру Эльбрус". Свидетельство о государственной регистрации программы для ЭВМ № 2018666284 от 13.12.2018.
[12] Jan Hunicka, Profile driven optimizations in GCC, Proceedings of the GCC Developers' Summit June 22nd-24th, 2005, Ottawa, Ontario Canada
[13] Patrick W. Sathyanathan, Wenlei He, Ten H. Tzen, Incremental whole program optimization and compilation, CGO '17 Proceedings of the 2017 International Symposium on Code Generation and Optimization, Pages 221-232
[14] Kulkarni., W.Zhao, H.Moon, et al. ―Finding Effective Optimization Phase Sequence‖, [A]. Proc. of ACM SIGPLAN 2003 Conference on Languages, Compilers and T ools for Embedded Systems, US:2003.
[15] Keith D. Cooper, Alexander Grosul, T imothy J. Harvey, Steven Reeves, Devika Subramanian, Linda T orczon, T odd Waterman. ― ACME: adaptive compilation made efficient‖, LCTES '05 Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, Pages 69
[16] Prasad A. Kulkarni , David B. Whalley , Gary S. Tyson, Evaluating Heuristic Optimization Phase Order Search Algorithms‖, Proceedings of the International Symposium on Code Generation and Optimization , p.157-169, March 11-14, 2007
[17] M. Haneda , P. M. W. Knijnenburg , H. A. G. Wijshoff, ― Generating new

general compiler optimization settings‖, Proceedings of the 19th annual international conference on Supercomputing, June 20-22, 2005, Cambridge, Massachusetts
[18] 19. Standard Performance Evaluation Corporation, http://www.spec.org/
[19] Кузьминский М.Б. Куда идет «Эльбрус». – «Открытые системы», 2011, №7.
[20] K. D. Cooper, D. Subramanian, L. Torczon, Adaptive optimizing compilers for the 21st century, The Journal of Supercomputing, vol. 23, no. 1, pp. 7–22, 2002.
[21] M. Wolf, D. Maydan, and D. Chen, “Combining loop transformations considering caches and scheduling,” in Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 274–286, December 1996.
[22] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, David I. August, Compiler optimization-space exploration. Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, March 23-26, 2003, San Francisco, California
[23] В.Ю. Волконский, А.В.Брегер, А.Ю.Бучнев, А.В.Грабежной, А.В.Ермолицкий, Л.Е.Муханов, М.И.Нейман-заде, П.А.Степанов, О.А.Четверина "Методы распараллеливания программ в оптимизирующем компиляторе" // Вопросы радиоэлектроники, серия ЭВТ, выпуск 3, 2012
[24] Ким А.К., Волконский В.Ю., Груздов Ф.А., Михайлов М.С., Парахин Ю.Н., Сахин Ю.Х., Семенихин С.В., Слесарев М.В., Фельдман В.М. Микропроцессорные вычислительные комплексы с архитектурой «Эльбрус» и их программное обеспечение. – «Вопросы радиоэлектроники», сер. ЭВТ, 2009, вып. 3, с. 5-37.
[25] Галазин А.Б., Грабежной А.В. Эффективное взаимодействие микропроцессора и подсистемы памяти с использованием асинхронной предварительной подкачки данных. – «Информационные технологии», 2007, №5.
[26] Галазин А.Б., Грабежной А.В., Нейман-заде М.И. Оптимизация размещения данных для эффективного исполнения программ на архитектуре с

многобанковой кэш-памятью данных. – «Информационные технологии», 2008, №3, с. 35-39.
[27] Galazin A., Neiman-zade M. An Unsophisticated Cooperative Approach to Prefetching Linked Data Structures. – Proceedings of the Eighth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-8). April 24, 2010, Toronto, Canada.
[28] Yong, S.H., Horwitz, S., Reps, T.: Pointer analysis for programs with structures and casting. SIGPLAN Not. 34(5), 91–103 (May 1999), http://doi.acm.org/10.1145/301631.301647
[29] А. В. Ермолицкий, М. И. Нейман-заде, О. А. Четверина, А. Л. Маркин, В. Ю. Волконский. Агрессивная инлайн-подстановка функций для VLIW-архитектур , Труды ИСП РАН, 2015, том 27, выпуск 6, страницы 189–198 (Mi tisp193), DOI: https://doi.org/10.15514/ISPRAS-2015-27(6)-13
[30] Steven S. Muchnick, Advanced Compiler Design & Implementation, 2003
[31] Chang P. P., Mahlke S. A., Hwu W. W. Using profile information to assist classic compiler code optimizations. Software Practice and Experience, V. 21, No12. -1991.-P. 1301-1321
[32] W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W. -M. Hwu, R. Hank and J. Gyllenhaal., Using profile information to assist advanced compiler optimization and scheduling, 1993
[33] Krebbers, R.: Aliasing Restrictions of C11 Formalized in Coq. In: CPP. LNCS, vol. 8307, pp. 50–65 (2013)
[34] InternationalOrganizationforStandardization:ISO/IEC9899-2011:Programming languages – C. ISO Working Group 14 (2012)
[35] R. P. J. Pinkers , P. M. W. Knijnenburg , M. Haneda , H. A. G. Wijshoff, Statistical Selection of Compiler Options, Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04), p.494-501, October 04-08, 2004

[36] Ермолицкий А., Шлыков С. Автоматическая векторизация выражений оптимизирующим компилятором. – Приложение к журналу «Информационные технологии», 2008, №11.
[37] Волконский, В., Ермолицкий, А., Ровинский, Е. Развитие метода векторизации циклов при помощи оптимизирующего компилятора. Информационные технологии и вычислительные системы, № 8:pp 34-56. — 2005
[38] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, Michael O'Boyle, Milepost GCC: Machine Learning Enabled Self-tuning Compiler // International Journal of Parallel Programming, 2011
[39] Suresh Purini, Lakshya Jain. "Finding good optimization sequences covering program space". Transactions on Architecture and Code Optimization (TACO), January 2013.
[40] M. Lam. Software pipelining: An e ective scheduling technique for VLIW machines // Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI 88), July 1988. Also published as ACM SIGPLAN Notices. V. 23. I. 7. P. 318 328.
[41] Aho A.V., Lam M., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools. 2nd ed. Addison-Wesley, 2007.
[42] Louis-Noel, Pouchet, Uday Bondhugula, Cedric Bastoul, Albert Cohen, J. Ramanujam, Ponnuswamy Sadayappan, Nicolas Vasilache. Loop transformations: convexity, pruning and optimization // 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, 2011, Pages: 549-562
[43] Qingfeng Zhuge, Zili Shao, Edwin H.-M. Sha. Optimization of Nest-Loop Software Pipelining // 2012, Department of Computer Science University of Texas at Dallas, Submitted paper http://www.utdallas.edu/ edsha/

[44] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, Guang R. Gao. Single-dimension software pipelining for multidimensional loops // ACM Transactions on Architecture and Code Optimization (TACO), Volume 4 Issue 1, March 2007, Article No. 7
[45] M. Fellahi and A. Cohen. Software Pipelining in Nested Loops with Prolog-Epilog Merging // Lecture Notes in CS, SpingerLink, 2009
[46] Dmitry M Maslennikov, Vladimir Y Volkonsky: Compiler method and apparatus for elimination of redundant speculative computations from innermost loops. Elbrus International October 9, 2001: US06301706
[47] Д.С.Иванов «Распределение регистров при планировании инструкций для VLIW-архитектур», Программирование, 2010, №6.
[48] Jaekyu Lee, Hyesoon Kim, Richard W. Vuduc. When Prefetching Works, When It Doesn't, and Why // ACM Transactions on Architecture and Code Optimization, vol. 9, no. 1, Mar. 2012.
[49] Четверина О.А., Методы коррекции профильной информации в процессе компиляции // Труды Института системного программирования РАН. 2015. Т. 27. № 6. С. 49-66.
[50] Galazin A., Neiman-zade M. An Unsophisticated Cooperative Approach to Prefetching Linked Data Structures. // Proceedings of the Eighth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-8). April 24, 2010. Toronto, Canada.
[51] Goh, C. J.; Yang, X.Q., Duality in Optimization and Variational Inequalities, Optimization theory and applications, v. 2, London : Taylor and Francis, 2002.
[52] Silva A.P., Stam A., "On multiplicative priority rating methods for the AHP", European Journal of Operational Research, 145 (2003), 92–108.
[53] [41] Jahn J, Scalarization in Multi Objective Optimization, 1985
[54] Hastie T., Tibshirani R., Friedman J. The Elements of Statistical Learning. Springer, 2014.
[55] К. В. Воронцов, Математические методы обучения по прецедентам

(теория обучения машин) // Курс лекций, http://www.machinelearning.ru/wiki/images/6/6d/Voron-ML-1.pdf, дата обращения 20.12.2017

[56] Vapnik, V. (2000). The Nature of Statistical Learning Theory. Information Science and Statistics. Springer-Verlag. ISBN 978-0-387-98780-4.

[57] Encyclopedia of Machine Learning, Claude Sammut, Geoffrey I. Webb (Eds.), Springer Science+Business Media, LLC 2011

[58] Principles of Risk Minimization for Learning Theory, V. Vapnik AT &T Bell Laboratories Holmdel, NJ 07733, USA

[59] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, A. Verri, "Are loss functions all the same?", Neural Comput., vol. 16, no. 5, pp. 1063-1076, Mar. 2004.

[60] Донской В. И. Алгоритмические модели обучения классификации: обоснование, сравнение, выбор. – Симферополь: ДИАЙПИ, 2014. – 228 с. ISBN 978–966–491–534–9

[61] Judea Pearl, Stuart Russell. ― Bayesian Networks‖. UCLA Cognitive Systems Laboratory, T echnical Report (R-277), November 2000.

[62] Лекции по логическим алгоритмам классификации К. В. Воронцов 24 июня 2010 г.

[63] Fu¨rnkranz J., Flach P. A. Roc ‘n’ rule learning-towards a better understanding of covering algorithms // Machine Learning. — 2005. — Vol. 58, no. 1. — Pp. 39–77. http://dblp.uni-trier.de/db/journals/ml/ml58.html#FurnkranzF05.

[64] Breslow L. A., Aha D. W. Simplifying decision trees: a survey // Knowledge Engineering Review. — 1997. — Vol. 12, no. 1. — Pp. 1–40.

[65] Esmeir S., Markovitch S. Lookahead-based algorithms for anytime induction of decision trees // Proceedings of the 21st International Conference on Machine Learning (ICML-2004). — 2004.

[66] Cohen W. W., Singer Y. A simple, fast and effective rule learner // Proc. of the 16 National Conference on Artificial Intelligence. — 1999. — Pp. 335–342.

[67] Freund Y., Schapire R. E. A decision-theoretic generalization of on-line

learning and an application to boosting // European Conference on Computational Learning Theory. — 1995. — Pp. 23–3

[68] Гладков Л. А., Курейчик В. В., Курейчик В. М. Генетические алгоритмы. — М.: Физматлит, 2006. — С. 320.

[69] Вайнцвайг М. Н. Алгоритм обучения распознаванию образов "кора" // Алгоритмы обучения распознаванию образов / Под ред. В. Н. Вапник. М.: Советское радио, 1973. С. 110–116.

[70] Лбов Г. С. Методы обработки разнотипных экспериментальных данных. — Но- восибирск: Наука, 1981.

[71] Isabelle Guyon , André Elisseeff, An introduction to variable and feature selection // The Journal of Machine Learning Research, 3, 3/1/2003

[72] I. Cohen, Q. Tian, X. Zhou, T Huang, Feature selection using principal feature analysis // Proc. of the 15th international conference on Multimedia. Urbana-Champaign; Universety of Illinois, 2007. P. 301—304

[73] Peng H., Long F., Ding C. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundency // IEEE Transactions on pattern analysis and machine intelligence. 2005. Vol. 27. No. 8. P. 1226—1238

Список иллюстраций

Список таблиц

использованные при построении набора оптимизационных линеек.

Таблица 4. Построение кода подкачки.

Таблица 5. Пример разных оптимальных наборов оптимизаций.

Таблица 6. Пример матрицы частичных ошибок.