Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук

Варданян Мамикон Ашотович

МЕТОД АВТОМАТИЧЕСКОГО ПОДБОРА ЭФФЕКТИВНЫХ ОПТИМИЗАЦИЙ КОМПИЛЯТОРА ПО НЕСКОЛЬКИМ КРИТЕРИЯМ НА ОСНОВЕ ПАРЕТО-ДОМИНИРОВАНИЯ

Специальность 05.13.04 —
«Математическое и программное обеспечение математических машин, комплексов, систем и сетей»

Диссертации на соискание ученой степени
кандидата технических наук

Научный руководитель:
академик РАН, доктор физ.-мат.наук
Иванников В.П.

Ереван 2015

Содержание

Введение

Актуальность работы. Современные компиляторы[18], как правило, включают сотни различных оптимизаций, которые, работая на разных этапах компиляции, влияют на качество сгенерированного кода. При этом набор оптимизаций (опций компилятора), при которых генерируется оптимальный объектный код зависит как от целевой архитектуры машины, так и от специфики выбранного приложения. Обычно разработчики и пользователи приложений используют параметры компилятора по умолчанию (например -O2 в компиляторах GCC и LLVM), определяющие базовые наборы оптимизаций. Такие наборы определяются разработчиками компиляторов известных наборов тестов (как правило, SPEC[43]) и обеспечивают компиляцию оптимального кода в среднем для данного набора. Обычно, для разных приложений оптимальные наборы опций компилятора могут различаться, в том числе, в зависимости от целевой платформы. При этом, для конкретного приложения можно значительно увеличить производительность только за счет правильного подбора параметров компилятора. Задача усложняется еще тем, что критериями оптимальности могут являться наряду с производительностью, также размер кода, энергопотребление, время компиляции и т.д. Кроме того, нередко требуется оптимизировать одновременно по нескольким критериям (например, по производительности и размеру кода). В некоторых случаях критерии могут не согласовываться между собой, и тогда приходится использовать компромиссные подходы.

Подбор опций компилятора и значений параметров компиляции для конкретного приложения требует глубокого знания инфраструктуры компилятора

и особенностей целевой платформы, поэтому для разработчиков и пользователей приложений, не специализирующихся в области компиляторных технологий, актуальны автоматические методы подбора параметров компиляции. В настоящее время имеются некоторые современны инструменты автоматического подбора опций компиляции (такие, как OpenTuner, COLE, ACOVEA, и т.д.). Однако, существующие инструменты автоматического подбора оптимизаций компилятора (в том числе с поддержкой многокритериального поиска) не удовлетворяют одновременно таким актуальным требованиям пользователей и разработчиков программного обеспечения как, например, поддержка подбора числовых параметров компиляции, многокритериальный поиск, параллельная кросс-компиляция и запуск на нескольких тестовых устройствах с целевой архитектурой.

Обеспечить улучшение эффективности работы приложения можно также посредством корректировки исходного кода. Корректировка исходного кода библиотеки может повысить уровень применимости ряда межмодульных оптимизаций (например, открытая вставка между модулями) компилятора на этапе компоновки, так как многие библиотеки (особенно более старые) при написании не были рассчитаны на такие возможности компилятора. Некоторые библиотеки содержат некорректные объявления области видимости своих функций, что приводит к тому, что все глобальные функции самой библиотеки автоматически экспортируются в качестве ее интерфейса. Это препятствует выполнению межмодульных оптимизаций на этапе компоновки, так как эти функции могут быть вызваны из других библиотек во время выполнения кода. Корректировка соответствующих объявлений в библиотеках позволяет существенно повысить применимость оптимизаций на этапе компоновки и соответственно улучшить производительность и уменьшить размер кода, при этом не нарушается целостность библиотеки.

Решение перечисленных задач проблем связано с существенными трудозатратами и требует высокий квалификации, и необходимо разработать соответствующие средства автоматизации для решения указанных проблем. В Институте системного программирования РАН проводятся исследования и разрабатывается набор инструментов TACT (Toolkitfor Automatic Compiler Tuning), обеспечивающий эффективное использование возможностей компилятора для оптимизации конкретного приложения под выбранную целевую архитектуру. Настоящая работа является составной частью этих исследований.

Целью диссертационной работы является разработка метода автоматического выбора опций компилятора и значений его параметров, обеспечивающих оптимизацию целевой программы в соответствии с выбранным критерием (включая возможность многокритериальной оптимизации) для заданного приложения и целевой платформы. Кроме того, необходимо разработать метод автоматической проверки и корректировки исходного кода библиотек, обеспечивающий выполнение оптимизаций во время компоновки.

Разработанные методы должны быть реализованы в рамках инфраструктуры набора инструментов TACT.

Методы исследования, используемые в диссертационной работе, включают методы поиска, основанные на генетических алгоритмах, методы статистической обработки данных и методы разработки программного обеспечения.

Научная новизна. В диссертационной работе были получены следующие результаты:

- Разработан метод автоматического подбора оптимизаций компилятора для конкретного приложения и целевой архитектуры. Данный метод поддерживает также многокритериальную оптимизацию на основе Парето-доминирования.
- Исследовано и оценено влияние конфигураций генетического алгоритма на процесс эволюционного поиска во время оптимизации.

- Разработан метод автоматической проверки и корректировки исходного кода библиотек программ, обеспечивающий улучшение эффективности выполнения оптимизаций на этапе компоновки.

Практическая ценность. Разработанные методы реализованы в рамках инструментальной инфраструктуры TACT, учитывая особенности компиляторов GCC и LLVM и характеристики применяемости концепций генетического алгоритма. Автоматическое улучшение качества применяемости компилятора на выбранные приложения позволяет разработчикам и пользователям приложений (при наличии исходного кода) получить прирост производительности и/или уменьшить размер исполняемого кода до нескольких десятков процентов за приемлемое время, не требуя от них знания инфраструктуры компилятора и целевой платформы.

Программная система разрабатывалась в рамках научно-исследовательского проекта корпорации Samsung и с ее согласия находится в открытом доступе (Open-source).

Положения, выносимые на защиту:

- Метод автоматического подбора опций компилятора и значений его параметров, обеспечивающий генерацию эффективного в соответствии с заданным критерием оптимальности целевого кода для конкретного приложения на заданной платформе. Данный метод поддерживает также многокритериальную оптимизацию на основе Парето-доминирования.[1-5]
- Метод автоматической проверки и корректировки исходного кода библиотек, обеспечивающий эффективное выполнение оптимизаций на этапе компоновки[2].
- Инструментальное средство на основе предложенных методов с использованием генетического алгоритма, позволяющее отражать влияние

разных значений конфигураций инструмента на качество результирующего кода.[1-4]

Апробация и публикации работы. Основные результаты и положения диссертационной работы обсуждались и докладывались на семинарах Института системного программирования РАН (2012-2015 гг.), научно-исследовательских семинарах Вычислительного центра РАН (Москва, Россия, 2014) и МФТИ (ГУ) (Москва, Россия, 2014), представлялись на международной конференции по вычислительным наукам и информационным технологиям “Computer Sciences and Information Technologies” (CSIT) 2013 в г. Ереване, Армения и на международной научно-практической конференции «International Conference on Computational Science» (ICCS) 2013 в г. Барселона, Испания.

Результаты работы отражены в пяти публикациях, список которых приведен в конце автореферата.

Структура диссертационной работы. В первой главе производится обзор работ, имеющих отношение к теме диссертации. В начале главы приводятся термины, понятия, которые будут использованы в тексте диссертации. Далее рассматриваются широко используемые компиляторные инфраструктуры с открытым исходным кодом (в первую очередь GCC и LLVM) и особенности выполнения ими оптимизаций. Во-второй главе приводится описание метода автоматического подбора опций и значений параметров компиляции (как для одной целевой функции, так и по нескольким критериям) для определенного приложения под конкретную архитектуру. В третьей главе приводится метод, позволяющий автоматически улучшить эффективность выполнения оптимизаций во время стадии компоновки. В четвертой главе описана реализация инструмента, разработанного в Институте системного программирования РАН на основе предложенных методов в рамках набора инструментов TACT (Toolkit for Automatic Compiler Tuning).

Глава 1

Обзор и анализ материалов, рассмотренных в рамках диссертационной работы. Компиляторные инфратсруктуры.

1.1. Рассматриваемые компиляторные инфраструктуры

В рамках диссертационной работы были исследованы инструменты разработки программного обеспечения с открытым исходным кодом, в том числе компиляторные инфраструктуры. Ричард Столлман в 1984 году основал проект GNU[6-13]. В рамках лицензии GNU в дальнейшем было разработано огромное количество программ с открытым исходным кодом. Его необходимость была обусловлена тем, что в восмедесятые годы было ограничено сотрудничество между компаниями по разработки программного обеспечения и было желательно исключить возможность из-за авторских прав закрывать доступ к ПО. Частью программ под лицензией GNU и является набор инструментов для разработчика, который включается в дистрибутивы операционной системы Linux. Одним из инструментов разработчика является компиляторная инфраструктура GCC, которая в течении долгого времени модернизируется в соответствии с новыми требованиями языков и целевых платформ. Одним из самых широко используемых компиляторных инфраструктур с открытым исходным кодом является также LLVM. В данной работе будут рассмотрены именно компиляторы GCC и LLVM.

1.1.1 Компилятор GCC

Компиляторная инфраструктура GCC (GNU Compiler Collection) в настоящее время поддерживает множество языков, такие как C, C++, Fortran, Objective-C, Objective-C++, Fortran, Java, Ada и работает на широком множестве платформ. Компилятор действует как переводчик. GCC осуществляет преобразование кода с языка высокого уровня на язык ассемблера путём применения последовательности проходов. Компилятор читает набор операторов, записанных на языке программирования высокого уровня, и транслирует его в набор машинных команд двоичного формата для дальнейшего выполнения на целевом компьютере.

Работу GCC можно разделить на три основные части:

1. «Front-End» — этап компиляции программы, при которой производиться синтаксический и семантический анализ и генерируется промежуточный код.
2. «Middle-End» или оптимизатор — этап компиляции, при которой производиться оптимизация промежуточного кода.
3. «Back-End» — этап компиляции программы, при которой оптимизированный промежуточный код транслируется в команды машинного кода целевой машины.

В компиляторной инфраструктуре GCC промежуточное представление (IL) характеризуется следующими понятиями:

1. Generic – внутреннее представление исходного уровня.
2. Gimple[14] – внутреннее представление промежуточного уровня.
3. RTL (Register Transfer Language[15] – внутреннее представление ассемблерного уровня.
   - На первом этапе компилятор выполняет лексический анализ (иначе говоря, положение текста по грамматическим правилам для транслируемого исходного языка). При этом считываются знаки из входного потока и по

тому, как они сгруппированы, определяются составляемые ими программные символы, числа.

- Процесс синтаксического разбора (parsing process) читает поток символов (лексем), передаваемый ему лексическим сканером, и следуя своему набору правил, определяет отношения между ними.
- Конструируется промежуточное представление на языке GIMPLE, которое является языково-независимым и машинно-независимым внутренним представлением для всех языков поддерживаемыми GCC, на GIMPLE делаются много машинно-независимых оптимизаций.
- GIMPLE переводится в псевдо-ассемблерный язык, который называется Языком Регистрового Перехода и зависит от целевой платформы платформы.
- Работа <back-end>-а компилятора начинается с анализа кода на языке RTL и выполнения некоторых действий по его оптимизации. При этом убираются избыточные и не используемые секции кода. Некоторые части кода могут быть перемещены по дереву в другое местоположение, чтобы предотвратить выполнение соответствующего набора инструкций большее число раз, чем это необходимо. Проводится большое количество различных оптимизаций, и некоторые из них проходят по обрабатываемому коду несколько раз.
- На данном этапе работает ассемблер, который транслирует код в объектный файл. Этот файл ещё не имеет выполнимого формата — он содержит исполняемый объектный код, но ещё не в том формате, который годится для его запуска на целевой машине.
- Компоновщик (linker) выполняет связывание и объединие объектных файлов в машинный код, выполняемый на целевой машине.

1.1.1. Компилятор LLVM

LLVM (Low Level Virtual Machine)[16-19] – компиляторная инфраструктура с открытыми исходными кодами написанный на языке C++. Функционирование

системы обеспечивается единым внутренним представлением, которое может быть представлено в текстовом виде, в виде структур данных в оперативной памяти, а также в двоичном виде как биткод. Этот биткод может быть сохранен в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. При этом, возможно использовать все предоставляемые LLVM возможности по обработке внутреннего представления (включая различные анализы, трансформации и т.п.). Поэтому инфраструктура LLVM предоставляет удобную базу для исследований по анализу и трансформации программ.

Функция программы во внутреннем представлении записана с явной разметкой потока управления, в которой каждый базовый блок состоит из трехадресных строго типизированных инструкций LLVM. Внутреннее представление можно перевести в форму с единственным статическим присваиванием (SSA), для чего предусмотрена специальная инструкция, либо оно сразу может находиться в SSA-форме (если при обработке программы на языке высокого уровня такое представление будет сгенерировано). Разница будет заключаться в скалярных временных переменных: в первом случае они будут выделяться на стеке, во втором в качестве таких переменных будут использоваться виртуальные регистры. В LLVM предусмотрены трансформации по переводу внутреннего представления в SSA и обратно. Следующие инструменты LLVM были использованы:

- lli – интерпретатор или JIT-компилятор, исполняющий байт-код LLVM.
- llc – компилятор байт-кода LLVM в ассемблерный код.
- opt – преобразует байт-код LLVM, и применяет либо явно заданные оптимизации, либо заранее заданный набор, соотнесенный с уровнями "O0, O1, O2, O3".
- Clang – компилятор переднего плана для LLVM, собирающий исходный код в байт-код или сразу в машинный код. Уровень оптимизации "O4"

соответствует применению межпроцедурных оптимизаций во время связывания (LTO).

Перечислим основные особенности LLVM:

- Реализована на C++;
- Модульная и расширяемая архитектура;
- Является статическим компилятором, а также имеет возможность динамической компиляции биткода;

Поддерживает несколько различных компиляторов переднего плана:

- C, C++, Objective-C (Clang, GCC);
- Ruby (Rubinius, MacRuby) ;
- Python (unladen swallow);

Поддерживает множество целевых архитектур:

- ARM, Alpha, Intel x86, Microblaze, MIPS, PowerPC, SPARC, …;

Промежуточное представление (LLVM IR) играет центральную роль в процессе компиляции. Все оптимизации реализованы как компиляторные проходы преобразования “LLVM IR to LLVM IR” Анализ кода может быть реализован как отдельный проход, а его результаты могут разделять несколько проходов, трансформирующих код.

1.2. Машинная архитектура ARM

Одним из самых широко распространённых архитектур для встроевоемых устройств (планшеты, смартфоны, устрохства малой мощности) является ARM[20-25](Advanced RISC Machine). Архитектура ARM — является процессорной архитектурой с сокращённым набором команд (RISC), разрабатанная компанией ARM Limited. В настоящее время устройства с

процессороной архитектурой ARM охватывают до 80% всех используемых устройств со встраиваемыми RISC-процессорами, что делает его очень востребованной.

Процессор семейства ARM может находиться в одном из следующих операционных режимов:

- User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.
- Fast Interrupt — режим быстрого прерывания (меньшее время срабатывания)
- Interrupt — основной режим прерывания.
- Supervisor mode — защищённый режим для использования операционной системой.
- Abort mode — режим, в который процессор переходит при возникновении ошибки доступа к памяти (доступ к данным или к инструкции на этапе prefetch конвейера).
- System mode — привилегированный пользовательский режим.
- Undefined mode — режим, в который процессор входит при попытке выполнить неизвестную ему инструкцию.

  Процессор переключается между режимами, когда возникают соответсвующие исключения или при изменении статуса регистров.

В рамках данной работы в качестве тестовых мобильных устройств будут использоваться машины с архитектурой ARM.

1.3. Некоторые из используемых тестовых приложений

Для тестирования оптимизаций компиляторов GCC и LLVM были выбраны широко используемые библиотеки с открытым исходным кодом. Эти библиотеки широко используется на современных встраиваемых системах с процессорами ARM. Примерами таких устройств являются мобильные телефоны и планшеты, пользовательские оболочки которых реализованы с использованием функциональности нижеописанных библиотек. Первая библиотека – libevas. По

результатам профилирования, на выполнение кода этой библиотеки уходит более 95% времени выполнения набора тестов expedite для EFL[26]. Enlightenment Foundation Libraries - набор свободно распространяемых графических библиотек для оконного менеджера Enlightenment. Используемый нами набор тестов expedite был урезан для оптимизации времени тестирования и получения большей стабильности результатов, без уменьшения покрытия всех главных функции libevas, для которых наиболее важно улучшение производительности. Еще одна библиотека для тестирования – sqlite[27]. Она представляет собой легковесную встраиваемую реализацию реляционной базы данных, которая может использоваться для хранения пользовательской информации во встраиваемых системах. В качестве тестов для sqlite использовался набор из 20 тестов, содержащий большую часть операций языка SQL – включая добавление, изменение, поиск и удаление данных, и разнообразное выполнение комбинаций этих операций в виде единой транзакции. Для уменьшения влияния накладных расходов при работе с базой данных использовалась виртуальная файловая система в оперативной памяти. В качестве третьего тестового приложения использовался webkit[28]. Это инфраструктура с открытым исходным кодом для отображения веб-страниц. Сейчас он используется в нескольких браузерах и других программах, нуждающихся в отображении и редактировании содержимого веб-страниц. Для тестирования производительности использовался набор тестов SunSpider[29] для JavaScript, включенный в репозиторий webkit. К сожалению, рассмотренные нами тесты, измеряющие время визуализации HTML-кода, оказались очень нестабильными. Время различалось до 10-15% от запуска к запуску, несмотря на все попытки уменьшить накладные расходы, например организацию чтения страницы из оперативной памяти.

Мы адаптировали все библиотеки вместе с их тестами для использования с разработанным инструментом TACT, который описан в следующем разделе.

1.4. Анализ существующих алгоритмов подбора опций компилятора

В настоящее время имеются разработки ряда алгоритмов, которые дают возможность подобрать опции и параметры компиляции для данного приложения.Такие алгоритмы в основном известны во многих поисковых задачах, в том числе оптимизационных. Приведем некоторые из них.

- Алгоритм полного перебора[30]. Этот алгоритм самый примитивный, так как строит полное пространство решений, используя все возможные в допустимой области комбинаций опций компиляции, и естественно получает глобальный экстремум (как правило минимум по целевой функции). Однако вычислительная сложность данного алгоритма неприемлема, так как поиск усложняетя при наличии большого числа опций, поэтому обычно применяют разные подходы, учитывающие конкретные особенности и цели поиска. Существуют разработки, которые способны в процессе поиска на этапах итерации подобрать значимые наборы опций и их параметров на основе экспертных, опытных или других оценок Но такой подход может иногда подводить, так как задолго до завершения итерации,опробованные значимые опции иногда могут не удовлетворять некоторым требованиям, и этот подход не самый удачный, но часто приемлемый для простых примеров.
- Алгоритм группового исключения[31]. Этот алгоритм представляет интерес, так как заведомо неприемлемые опции, например,те, которые вызывают увеличение времени исполнения программы, не включаются в дальнейший поиск оптимального набора опций. Однако данный алгоритм сильно зависит от того, насколько влияние опции друг на друга значительны и от этого может колебаться производительность. Для определения целесообразности использования какой-либо опции,

применяется проверка, насколько улучшает или ухудшаетт включение, или выключения опций на время выполнения компиляции программы.

- Алгоритм итеративного исключения. В данном алгоритме, в процессе итерации последовательно исключаются те опции, которые во время выполнения существенно увеличивают время компиляции программы. В процессе итераций удаляется опция, которая дала неприемлемый результат. Процессу удаления подвергаются и последующие опции с неприемлемым результатом. Процедура удаления прекращается в случае, когда все оставшиеся опции будут приемлемы.
- Алгоритм комбинированного исключения[32]. Данный алгоритм совмещает в себе идеи алгоритмов группового и итеративного исключения. Этот алгоритм также полагает, что опции компиляции слабо влияют друг на друга. Данный алгоритм исключает опции с неприемлемым значением, как имело место в алгоритме группового исключения. В противном случае он исключает при осуществлении итерационных процедур, как это было в алгоритме итерационного исключения. Данный алгоритм показал более удовлетворительные результаты, чем предыдущие. Однако, при увеличении количества опций подход становится более уязвимым к требованиям ко времени. В настоящее время бурно развиваются и особый интерес представляют разработки генетически алгоритмов и эти исследования последовательно развеваются.
- Генетический алгоритм[33]. Позволяет найти приемлемые решения проблемам поиска удовлетворительных опции и параметров компиляции, с применением механизмов, подобные эволюционным развитиям, посредством последовательного подбора и комбинирования искомых параметров, напоминающие эволюционное развитие, в процессе которой остаются только полезные особи. Алгоритм начинает свою работу с формирования

начальной популяции, то есть начального выбора набора допустимых решений задачи, которые выбираются в основном вероятностными технологиями, случайным образом. На каждом этапе эволюции с помощью селекции, мутации и скрещивания строят новую популяцию, которая.содержит наиболее значимые решения.

- Алгоритм статистического выбора[34]. Алгоритм основан на применения статистических данных, полученных экспериментальным и опытным анализом и гипотезами, которые нередко противоречат друг друга. Кроме того, для их проверки требуется значительное число запусков тестируемой программы с различными наборами опций, значения которых распределены непредсказуемым образом и проверяются экспериментально, используя статистические, вероятностные и контрольные данные. На этой основе определяют, попадает ли опция с данным значением в оптимальный набор.
- Жадный конструктивный алгоритм. Пространство поиска представляется как ориентированный ациклический граф, корнем которого является вершина, представляющая пустое множество. Из корневой вершины графа выходят n дуг, при этом, каждая дуга соответствует определенной оптимизации и из каждой конечной вершины также выходят дуги, которые образуют пространство оптимизационного поиска. n оптимизаций далее с конечных вершин дуги последовательно выходит новое множество дуг, представляющих последовательные приемлемые оптимизации. Применение жадных алгоритмов часто дает хорошие результаты при решении некоторых конкретных проблем.
- Алгоритм случайного поиска. Алгоритм основан на случайных значений при входе и с помощью генератора случайных величин, как правило, равномерно распределенных, стараются улучить результаты выбора,обычно, уровень производительности. То есть,все результаты тоже являются

случайными величинами,которые иногда могут удовлетворять поставленным требованиям.

- Нетерпеливые алгоритмы. Иногда во время поиска срочно важно найти какое-то решение, естественно случайное. Но, чтобы не тратить время на алгоритме случайного поиска, просто совершается локальный запуск по некоторым окружающим точкам. Результат обычно не удовлетворительный, но иногда приемлемо в данное время.

1.5. Анализ существующих инструментов автоматического подбора эффективных оптимизаций компилятора

В настоящее время имеются несколько инструментов автоматического подбора эффективных оптимизаций компилятора. Многие из них основаны на принципах эволюционных алгоритмов, но имеются также методы, основанные на машинном обучении, жадных произвольных алгоритмах и т.д.

Эти инструменты обычно основаны на автоматическом сужении пространства поиска, т.е. фильтрацию параметров и опций, которые не могут повлиять на производительность. Рассмотрим наиболее значимые инструменты, имеющих влияние в процессе разработки данной работы.

1.5.1. MILEPOST

MILEPOST (MachIne Learning for Embedded ProgramS optimization, Машинное Обучение для Оптимизации Встраиваемых Программ)[35-37] - исследовательский компилятор с применением методов машинного обучения. Он реализован как надстройка над оригинальным компилятором GCC, расширяя его функциональность посредством ICI (Interactive Compilation Interface), Интерактивный Интерфейс Компиляции, который добавляет поддержку внешних встраиваемых модулей. MILEPOST также включает в себя ряд плагинов для ICI.

Каждый плагин представляет собой специальную библиотеку, которая содержит процедуры инициализации / завершения и обработчик событий. События должны быть размещены внутри кода GCC, так что для создания нового события его также необходимо добавить в исходный код GCC, и он должен быть перекомпилирован. В настоящее время существуют события для работы с оптимизациями GCC и два соответствующих встраиваемых модуля. Первый модуль может извлечь последовательность проходов в текстовый файл, а второй позволяет запускать проходы оптимизаций в последовательности, взятой из текстового файла. Оба эти модуля работают на уровне функций исходной программы, поэтому для различных функций могут быть использованы различные последовательности проходов оптимизаций.

Другие модули в MILEPOST GCC извлекают статические особенности программ. Характеристики структуры программы используются в алгоритме машинного обучения для нахождения в базе данных подобной по характеристикам программы. Характеристики программы, как правило, обобщают её внутреннее устройство описывают основные аспекты, необходимые алгоритму машинного обучения, для нахождения оптимизаций, которые хорошо работают на данном классе программ.

Для извлечения статических или динамических характеристик программы могут быть вызваны дополнительные проходы в системе плагинов ICI. Во время компиляции программа представлена несколькими структурами данных, реализующими промежуточное представление, граф потока управления, цепочки определение-использование, иерархию циклов и т.д. Доступность структур данных зависит от прохода компиляции, выполняющегося в данное время. Для статистического машинного обучения информация об этих структурах данных кодируется как числовой вектор постоянного размера - этот процесс называется извлечением особенностей и необходим для возможности повторного использования знаний для оптимизации других программ. В GCC реализован

дополнительный проход ml-feat для извлечения статических характеристик программы. Этот проход по умолчанию не вызывается во время компиляции, но может вызываться при помощи плагина extract_program_static_features после любого прохода, когда готовы все данные в GCC, необходимые для получения конкретной характеристики.

В MILEPOST GCC выделение характеристик осуществляется в два этапа. На первом этапе, извлекается реляционное представление программы, а во втором этапе из этого представления вычисляется вектор особенностей. На первом этапе программа характеризуется числом сущностей и отношениями между ними. Пространство сущностей является прямым отображением аналогичных сущностей, определённых в языке программирования, либо строится во время компиляции. Примерами таких сущностей являются переменные, типы, инструкции, базовые блоки, временные переменные и т.д. Отношения над множеством сущностей являются подмножеством их декартова произведения.

Отношения определяют свойства сущностей или связи между ними. Для описания отношений используется логическая нотация, подобная языку Пролог, но с упрощённой семантикой и подходящая для описания отношений и операций между ними. Для извлечения реляционного представления программы обрабатываются внутренние структуры данных компилятора. Эти данные собираются с использованием прохода ml-feat. На втором этапе строится программа на Прологе, вычисляющая вектор характеристик, и в которую встроены логические отношения, извлеченные из внутренних структур данных компилятора на первом этапе. Модуль extract_program_static_features вызывает компилятор Пролог, чтобы выполнить эту программу, результатом её работы является вектор особенностей, которые впоследствии могут быть использованы для создания машиной модели обучения и прогнозирования лучшей последовательности проходов оптимизаций для новых программ. Недавно MILEPOST GCC стал частью большого проекта - Continous Collective Compilation

Framework (CCC), инфраструктура непрерывной коллективной компиляции - совместная инфраструктура со встраиваемыми модулями для выполнения итеративной компиляции (т.е. поиска выгодных комбинаций оптимизаций программ), который направлен на сбор различных статических и динамических данных о профилях оптимизаций в своей коллективной базе данных оптимизаций. Он применяется для автоматической оптимизации программ, используя методы машинного обучения, и поддерживает точную оптимизацию на уровне функций, в случае, если это поддерживается в компиляторе.

Мы опробовали MILEPOST на платформе x86_64. Вначале мы установили MILEPOST GCC вне CCC. Он был в состоянии компилировать тестовые программы, создавать текстовые файлы со списком выполненных проходов для каждой функции и экспортировать статические особенности программы, а также скомпилировал программу, используя порядок проходов, указанный в текстовом файле.

После успешного изучения MILEPOST GCC мы попробовали использовать его внутри CCC Framework. CCC Framework предоставляет различные сценарии работы с компиляторами, поддерживающими ICI (MILEPOST GCC), а также реляционную базу данных, где хранятся результаты. Основная идея инфраструктуры CCC заключается в создании обучающего множества программ на основе флагов компилятора, выбранных случайно, связанными с ними последовательностями проходов, особенностями программ и характеристиками скорости/размера кода, которые хранятся в доступной извне коллективной базе данных оптимизации, Collective Optimization Database (COD). Затем, эта обучающая выборка используется для прогнозирования лучших флагов оптимизации и последовательности проходов для новой программы с учетом ее вектора особенностей. COD доступна при использовании скриптов CCC и веб-службы. В связи с тем, что система CCC находится в настоящее время в стадии разработки, ее документация дописана не до конца, так что даже установка

системы CCC является довольно сложной. Когда конфигурационные файлы CCC и переменные окружения были настроены, мы попробовали провести простой тест - запустить простую тестовую программу много раз, собранную с различными случайными опциями и сохранять время выполнения в базе данных, а затем пробовать с помощью CCC получить лучшие флаги оптимизации для другой программы, которая незначительно отличается от оригинала. Оказалось, что часть, которая предсказывает настройки опций, на данный момент не работает. Мы связались с разработчиком CCC по данному вопросу, и он подтвердил, что основа прогнозирования находится в стадии разработки, и что сейчас она не может быть проверена.

Другой интересной особенностью MILEPOST является его потенциальная поддержка динамических особенностей программы, полученных с помощью профилировщика. Например, одной из таких особенностей может быть значение аппаратного счётчика промахов по кэшу. Чем большее значение он имеет, тем потенциально лучше целевое приложение будет реагировать на оптимизацию локальности кэша. Это может помочь выявить программы, которые выиграют от данного класса оптимизаций.

В целом, несмотря на то, что MILEPOST имеет уникальные особенности, он все еще выглядит незрелым, и его анализ показывает, что еще много предстоит сделать, прежде чем он сможет давать прогнозы по оптимизации. Также в нём отсутствует какой-либо эволюционный алгоритм, по словам его разработчиков, обучение проводится по совершенно случайным опциям, и пока что не понятно как можно делать надёжные прогнозы по оптимизациям. Использование таких эвристик, как евклидово расстояние между векторами характеристик, делает равнозначными все статические характеристики программы, например, число промахов кэша и число обращений к памяти в функции. Обе характеристики могут изменяться произвольно и независимо между приложениями, так что

непонятно, как достоверно определить сходство программ на основе подобных векторов характеристик.

Разработчики считают, что официальная ветвь GCC должна включать в себя этот инструмент, чтобы упростить его использование с различными версиями компилятора. В противном случае потребуются исправления каждой новой версии компилятора каждый раз перед его использованием с данным инструментом, в то время как такие исправления могут быть трудно применимы для новой версии, либо поддержка GCC ICI может быть в будущем прекращена его разработчиками.

1.5.2. ACOVEA

ACOVEA (Analysis of Compiler Options via Evolutionary Algorithm, Анализ Параметров Компилятора через эволюционные алгоритмы)[38] - реализует генетический алгоритм для поиска "лучших" опций для компиляции программ с GCC для языков C и C++. Инструмент позволяет использовать пользовательские скрипты для контролирования процесса эволюции, поэтому он может улучшать производительность, размер кода или любой другой указанный пользователем критерий. Этот инструмент является наиболее развитым среди инструментов, доступных в данный момент, и его исходный код доступен в открытом виде в течение нескольких лет. Он также включен в Debian и Gentoo Linux репозитории, а также используется сообществом Gentoo Linux для настройки опций компилятора для сборки этой системы на конкретных компьютерах.

В работе использоваан ACOVEA на платформе ARM на целевом приложении EFL, для помощи при ручной настройке компилятора GCC, а также для автоматической настройки параметров оптимизаций (встраивания функций, разворачивания циклов, распределения регистров и предзагрузки массивов в циклах).

Однако, оригинальной ACOVEA не хватает многих функций, которые были бы удобны для использования в работе. Во-первых, ей не хватает инфраструктуры для поддержки интерпретации результатов и выделения наиболее значимых

опций, полезность которых бы стабильно воспроизводилась. Во-вторых, она не может использовать параллельное выполнение целевого приложения на нескольких тестовых платах, чтобы ускорить процесс оценки популяции. В-третьих, в настоящее время она не поддерживается.

1.5.3. COLE

COLE (Compiler Optimization Level Exploration)[39] - набор для автоматического поиска уровней оптимизации, оптимальных по Парето, на основе многоцелевого эволюционного поиска. COLE разрабатывается как исследовательский инструмент, ожидается, что он будет доступен с открытым исходным кодом. Он ещё не готов до конца, но автор утверждает, что COLE способен выполнять настройку компилятора с учетом сочетания параметров. Каждая сущность в популяции представляет определенное сочетание параметров, используемых для компиляции программы, но этот инструмент нацелен не на поиск максимального абсолютного значения у каждой сущности в популяции (как правило, оценивается эффективность производимых исполняемых файлов), а на поиск оптимальных по Парето популяций по нескольким параметрам одновременно (например, время выполнения / размер кода).

В своей работе авторы COLE экспериментально доказали, что автоматическое построение оптимальных по Парето уровней оптимизации компилятора осуществимо на практике. Кроме того, используя компилятор GCC и наборы тестов SPEC CPU2000, они показали, что многоцелевой эволюционный алгоритм поиска, предложенный в COLE, превосходит случайный поиск, а также определённые вручную в GCC стандартные уровни оптимизации. В работе было найдено, что только 25% от оптимизаций, используемых в стандартных уровнях оптимизации GCC, присутствуют в одном из оптимальных по Парето уровнях оптимизации, и только несколько оптимизаций присутствуют во всех оптимальных по Парето уровнях оптимизации. Также, авторы использовали COLE

для определения чувствительности наборов тестов по отношению к оптимизациям компилятора, что позволило определить группы тестов с одинаковой чувствительностью к уровням оптимизации компилятора, что даёт разработчикам компиляторов и исследователям способ выбора представительного набора тестов для своих исследований или проекта разработки.

1.5.4. OpenTuner

OpenTuner[40] - расширяемая инструментальная среда (framework), разработанная в MIT и позволяющая пользователю с помощью добавления механизма и конфигурации поиска подобрать значения по выбранным критериям. Инструмент представляет особый интерес, так как предлагает новую открытую среду для создания домен-специфических многоцелевых autotuner-ов, поддерживает настраиваемые представления конфигураций, имеет простой интерфейс для взаимодействия с программой. Имеет также возможность использовать разные методы поиска одновременно и динамично выделять методы, прошедшие многочисленные проверки. Демонстрирована эффективность для различных наборов оптимизаций. Многоцелевой автоматической настройкой можно определить приоритеты между производительностью и другими критериями, такие как энергопотребление и размер кода, а также другие требования. Несмотря на то, что инструмент дает хорошие результаты на некоторых простых тестах (таких как умножение матриц, и т.д.), в нем пока не реализованы некоторые полезные свойства (такие, как многокритериальный поиск, поддержка параллельной кросс-компиляции и запуска на целевой платформе, и т.д.). Эта разработка имеет большие возможности развития и в диссертационной работе для сравнения полезна.

1.5.5. PEAK

PEAK[41] - инструментальное средство, разработанное в Purdue University и позволяющее посредством комбинации нескольких приближенных алгоритмов

поиска за приемлемое время подобрать оптимальный набор опций компилятора. В описании инструментального средства говорится о преимуществах по отношению к рассмотренным автором инструментам, однако в нем отсутствует описание поддержки подбора числовых параметров компиляции, многокритериального поиска и т.д, которые очень важны в рамках данной задачи. При этом инструмент не находится в открытом доступе.

1.5.6. PERI

PERI Auto-Tuning[42] инструмент автоматической настройки компилятора, разработанной в Университете Теннесси. Эта работа описывает инструмент общей стратегии при автоматической настройки компилятора для оптимизации крупномасштабных приложений по критерии производительности, используя архитектурные особенности, чтобы добиться высокой производительности на платформе x86. Концептуальная схема автотюнинга в PERI и некоторые этапы работы показали, что метод фокусируется на трансформации, генерации кода, и офф- лайн поиске.

Оптимизация может произойти во время производственных циклов, особенно для задач или машин, чьи оптимальные изменения конфигурации меняются во время исполнения. Пользование системой автоматической настройки компилятора PERI производится через общие интерфейсы. Инфраструктура инструмента и экспериментальные результаты производительности от применения автотюнинга проводились на примерах из линейной алгебре.

Концептуальная схема автотюнинга в PERI заключается в следующих этапах поиска:

- Сортировка. Этот шаг включает в себя измерение производительности, анализ и моделирование для определения возможности приложения для оптимизации.

- Семантический анализ. Этот шаг включает в себя анализ семантики программы для поддержки безопасного преобразования исходного кода. Анализы включают традиционный компилятор анализ для определения данных и зависимости по управлению и могут использовать семантическую информацию преобразования исходного кода предоставленную пользователем в специфических отраслях.
- Трансформация. Преобразование и изменение исходного кода, чтобы создавать возможность реализации компиляции для генерации лучших преобразований исходного кода.
- Код поколения. Эта фаза производит набор возможных реализаций, которые могут быть рассмотрены.
- Идентификация. Рассматриваются разные варианты генерируемого кода для идентификациипри реализации производительности пользовательского ввода.
- Обучение. Обучение проходит как отдельный шаг выполнения, предназначенный главным образом для получения данных о производительности для обратной связи в процессе оптимизации.

1.6. Выводы

Большинство известных инструментов автоматического подбора опций компилятора работают только с одним критерием оптимальности (оптимизируется либо только производительность, либо только размер исполняемого файла, либо только время компиляции и т.д.). Желательно иметь возможность поиска подходящего набора опций компилятора сразу по нескольким критериям. Обычно в случае многокритериальной оптимизации используются компромиссы, позволяющие объединить несколько критериев в один. Один из основных подходов такого объединения основан на принципе Парето-доминировании. В швейцарском федеральном институте технологии (Цюрих) разработан алгоритм многокритериального поиска данных SPEA2, который является улучшенной

версией известного алгоритма SPEA (Strength Pareto Evolutionary Algorithm) и основан на принципе Парето-доминирования. В публикации, описывающей алгоритм, проводится сравнительный анализ с другими известными алгоритмами многокритериального поиска (такими как, NSGA-II и PESA), где алгоритм SPEA2 обеспечивает лучшую устойчивость и скорость сходимости при выпуклой границе Парето. Этот алгоритм и был взят за основу для разработки метода подбора опций и значений параметров компиляции, описываемого в следующей главе.

Глава 2.

Метод автоматического подбора эффективных оптимизаций компилятора

2.1 Опции компиляторов GCC и LLVM

Большинство современных компиляторов имеют множество оптимизационных опций, которые при включении позволяют компилятору выполнять те или иные оптимизации. Одними из самых популярных и широко используемых компиляторов с открытым исходным кодом являются компиляторные инфраструктуры GCC и LLVM (подробно описаны в первой главе).

Компилятор GCC имеет 200+ оптимизационных опций, а также 70+ численных параметров оптимизаций. Многие из опций уже включены в базовые уровни оптимизаций (-O1, -O2, -O3, -Os) по умолчанию, но часто для конкретных приложений отключив некоторые оптимизации, можно добиться улучшения качества сгенерированного машинного кода.

LLVM может использоваться как оптимизирующий компилятор из полученного байткода в машинный код, так и для интерпретации и JIT-компиляции. LLVM (при помощи различных фронтендов, в том числе сторонних) позволяет компилировать программы, написанные на различных языках, учитывая их специфику. В рамках нашего проекта в качестве фронтенда LLVM был выбран Clang для языков C и C++, использующий llvm в качестве бэкенда. При подборе эффективных оптимизаций для Clang нужно также подобрать оптимизации в бэкенде llvm, поскольку множество оптимизаций выполняются на этапе кодогенерации.

2.2. Представление опций и параметром компилятора в генетическом алгоритме.

Как было сказано в главе 1, для автоматического подбора эффективных оптимизаций компилятора был выбран подход использующий генетический алгоритм, где приведены обозначения терминов и принципов соответствующих терминам используемым в генетике. В генетических алгоритмах каждая особь представляет потенциальное решение некоторой проблемы. В классическом ГА особь кодируется строкой двоичных символов – хромосомой, каждый бит которой называется геном. В нашем случае геном является оптимизационная опция (или параметр) компилятора, а особь - это конечный набор опций (параметров). Набор особей – потенциальных решений составляет популяцию. Начальная популяция обычно заполняется произвольно выбранными особями (наборами опций компилятора). Дальнейший поиск "оптимального" решения задачи выполняется в процессе эволюции популяции, т.е. последовательного преобразования одного конечного множества решений в другое с помощью генетических операторов репродукции, скрещивания и мутации. Генетический алгоритм используют механизмы естественной эволюции, основанные на следующих принципах:

1. Репродукция. Этот принцип основан на естественном отборе. Выживают те особи, которые более “приспособлены”. Обычно это особи с лучшими значениями целевых функций (фитнес), но часто применяются различные методы селекции.
2. Скрещивание (кроссинговер). Данный принцип обусловлен тем фактом, что "хромосома" потомка (особи на следующей итерации алгоритма) состоит из частей "хромосом" родителей (особей из предыдущей итерации).
3. Мутация. Принцип основан на концепции резкого произвольного изменения структуры потомков (н.п. отключить опцию, или поменять значение параметра) с целью повысить разнообразие особей в популяции

Эти три принципа составляют ядро генетического алгоритма. Используя их, популяция (множество наборов опций компилятора) эволюционирует от поколения к поколению и в конечном итоге сходится на каком-то поколении или достигает ограничения на количество поколений.

Особь в генетическом алгоритме может быть описан различными способами. Одним из самых применяемых методов реализации особи в генетическом алгоритме является векторное представление. Под вектором будем понимать массив фиксированной длины, где каждый элемент (ген) имеет булевое, целочисленное или вещественное значение. В контексте оптимизационных опций компиляторов GCC и LLVM гены особи представляют собой булевы значения применяемых оптимизаций (-f/-fno, оптимизация включается или нет) и целочисленные значения оптимизационных параметров (--params). Числовые параметры оптимизации всегда имеют заранее заданные диапазоны значений, а также значение по умолчанию, которое соответствует значению заложенному в базовый оптимизационный набор компилятора.

2.3. Анализ и приспособление принципов генетического алгоритма

Для удовлетворения требованиям разрабатываемого метода автоматического подбора эффективных оптимизаций компилятора были доработаны и приспособлены под нашу задачу следующие концепции генетических алгоритмов:

- Инициализация первого поколения.
- Оператор мутации особей.
- Оператор скрещивания и селекции.
- Поддержка мультипопуляционного подхода и возможность миграции между популяциями.

2.3.1 Инициализация первого поколения

Одним из важнейших этапов в генетических алгоритмах является инициализация популяций в первом поколении. Очень важно по возможности

хорошо подобрать начальный набор особей, так как от этого зависит дальнейший процесс работы генетического алгоритма. В следующих поколениях новые гены могут получиться лишь мутацией особи, но как показывают исследования, слишком большое значение коэффициента мутации может сделать дальнейшую работу генетического алгоритма слишком хаотичной и менее эффективной.

Компиляторы GCC и LLVM имеют множество оптимизационных параметров (--params), которые задают числовые значения применяемости соответствующих оптимизаций. Как правило, эти параметры имеют определенный диапазон получаемых значений, а также значения, которые берутся по умолчанию и соответсвуют настроенному значению в соответствующей оптимизации компилятора.

При инициализации особи используются разные вероятностные методы произвольного выбора его генов. Выбор значения булевых опций компилятора (-foption) обычно осуществляют произвольно включив или отключив опцию (-f|-fno). Для числовых параметров часто возникают определенные затруднения, так как параметр обычно имеет большой диапазон значений и задав плохое значение, можно в дальнейшем получить неэффективный код или ошибку при компиляции или выполнении.

В результате исследования вероятностных методов распределения, для выбора “случайных” значений числовых параметров были выбраны подходы равномерной и нормальной распределений.

При равномерном распределении, вероятность выбора значения параметра одинаковая на всем диапазоне значений. Поскольку параметр получает только целочисленные значения, то функция плотности:

$$f(x)=\frac{1}{n},$$

где n - это целое число, равное разнице максимального и минимального значения диапазона получаемых значений параметра.

В нормальном (или Гауссовом) распределении функция плотности вероятности определяется следующей формулой:

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

где параметр μ — математическое ожидание (в нашем случае совпадает со значением параметра по умолчанию), а параметр σ — среднеквадратическое отклонение (σ^2 — дисперсия) распределения.

Согласно правилу трех сигм (3σ) большинство значений (около 99%) функции плотности (значения нормально распределённых случайных величин) получаются в диапазоне [μ -3σ, μ + 3σ]. Чем больше значение среднеквадратического отклонения, тем больше разброс (дисперсия) случайных значений. На рис. 1 наглядно видно поведения случайных значений в зависимости от σ.

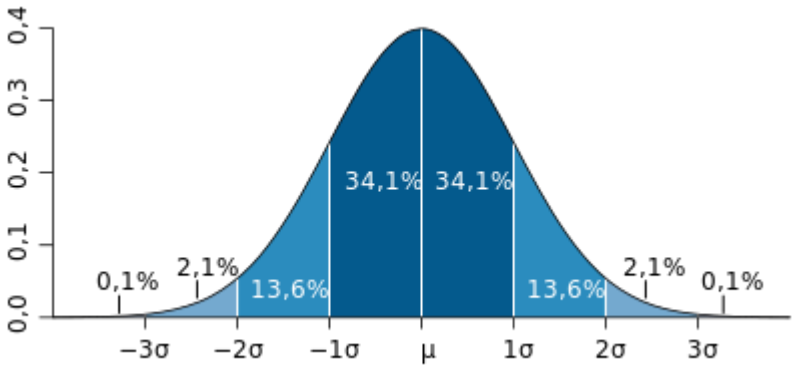


Рис. 1 Нормальное распределение при разных значениях σ.

Пользователь может руководствоваться следующими данными, полученными во время тестирования системы: при σ=1 около 30% параметров при инициализации получают значение, равное значению по умолчанию, 53% с отклонением до 10% от значения по умолчанию, 74% с отклонением до 20% от значения по умолчанию и 97% с отклонением до 50% от значения по умолчанию, а при σ=2 указанные значения равны, соответственно, 24%, 40%, 55% и 85%.

Как показали результаты тестирования, для выбранного тестового приложения лучший сгенерированный код получается, когда значение большинства числовых параметров равно или близко к значению параметра по-умолчанию (как правило эти значения получаются на SPEC) и лишь несколько параметров значительно влияют на получаемый результат (обычно не более пяти параметров на GCC). При оценке особей после инициализации, особи в которых параметры получили значения по нормальному распределению были более эффективные по сравнению с равномерным распределением. Нормальное распределение обеспечивает также лучшую сходимость за счет уменьшения разброса значений. Иногда “лучшие” значения параметра могут быть близки к крайним значениям диапазона и при нормальном распределении вероятность получить эти значения незначительны. Но эти значения также могут быть достигнуты за счет мутации в процессе работы алгоритма.

2.3.2 Оператор мутации

Как выше отметили, одним из операторов генетических алгоритмов является мутация особей. Под мутацией подрозумеваем произвольное (или псевдопроизвольное) изменение значения гена хромосомы. Мутация является основным инструментом, обеспечивающим разнообразие при работе генетического алгоритма, так как новые гены не могут получиться другими операторами генетического алгоритма. Обычно задается коэффициент мутации, от которого зависит степень произвольного изменения данной хромосомы. Чем больше это значение, тем больше модификации подвергается особь. Как показывают исследования, если задать слишком большой коэффициент мутации, то получаем довольно большой разброс значений при выполнении генетического алгоритма и соответственно уменьшается скорость сходимости. Особый интерес представляет мутация числовых параметров оптимизаций. Как и при инициализации, при мутации также могут использоваться как стратегия основанная на равномерном распределении, так и основанная на нормальном

распределении. Как показывают исследования от приложения к приложению результаты выбора стратегии могут отличаться. Как правило, выбор нормального распределения более консервативный, и для тех приложений у которых лучшие результаты получаются при значениях численных параметров оптимизаций (--param) близких к базовым, эта стратегия предпочтительнее. Однако есть и такие примеры, когда лучший сгенерированный машинный код получается при значениях параметров (--params) с большим отклонением от базового значения. В таких случаях равномерное распределение может показать лучший результат и соответственно лучшую сходимость. На Рис. 2 продемонстрированы средние значения производительности на каждом поколении при нормальном и равномерном распределениях на приложении SqLite. Как можно видеть при равномерном распределении средний результат производительности получается лучше поскольку на итоговый результат повлияли численные параметры со значениями близкими к крайним значениям диапазона. Эти значениям могут быть достигнуты и при нормальном распределении, однако вероятность этого существенно меньше.

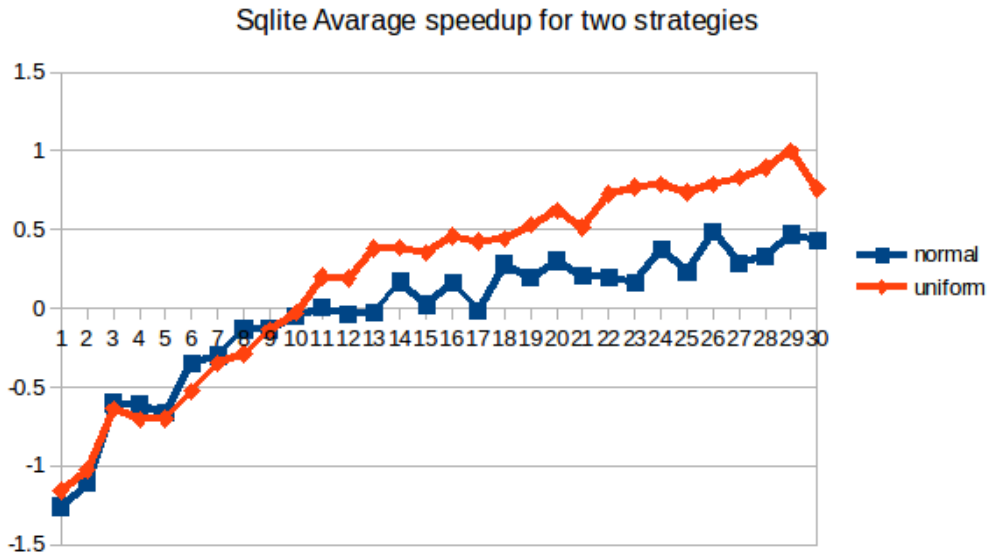


Рис. 2 Сравнительные результаты работы инструмента TACT на приложении SqLite при нормальном и равномерном распределении.

2.3.3 Оператор скрещивания и селекции

Основным инструментом в генетических алгоритмах, обеспечивающий процесс эволюции является концепция скрещивания особей (crossover). В зависимости от применяемых генетических алгоритмов, методы скрещивания могут отличаться, однако основной принцип является одинаковым. При переходе от одного поколения к другой наиболее приспособленные пары особей предыдущего поколения соединяются (скрещиваются) в новую особь, который в себе содержит гены обоих родителей. При векторном представлении особи, это происходит сливая два вектора одинаковой длины в новый вектор, при условии что в новый вектор (особь) войдут одинаковое количество генов от родительских векторов.

Особый интерес представляет также процесс выбора пары приспособленных родительских особей для скрещивания в новый. Этим в генетических алгоритмах занимается оператор селекции. Существуют множество стратегий выполнения селекции, от самого простого (произвольного выбора) до более сложных. Исследования, а также апробация показали, что бинарная турнирная селекция обеспечивает приемлемый уровень выбора (этот подход\также используется также во многих базовых генетитеских алгоритмах).

2.3.4 Метод использующий несколько параллельных популяций и миграция между ними

Существуют генетические алгоритмы, в которых одновременно могут использоваться несколько параллельных популяций[59-61]. В таких генетических алгоритмах в первом поколении одновременно инициализируются несколько популяций независимо друг от друга, работая параллельно от поколения к поколению и в итоговый результат записывается лучшие из всех поколений. Такая концепция обеспечивает больше разнообразия генов и соответственно повышается эффективность работы генетического алгоритма. В данной работе была исследована и реализована именно такая модель. Следует отметить, что применяется также концепция миграции между автономно работающими

популяциями. Под миграцией будем понимать обмен наиболее приспособленных особей между параллельно работающими популяциями и использовании их в процессе скрещивания в рамках конкретной популяции. Как показывают результаты тестирования, такая концепция может обеспечить повышение эффективности метода до нескольких процентов, обеспечивая связь между популяциями.

2.4. Подбор опций и параметров компиляции по нескольким критериям

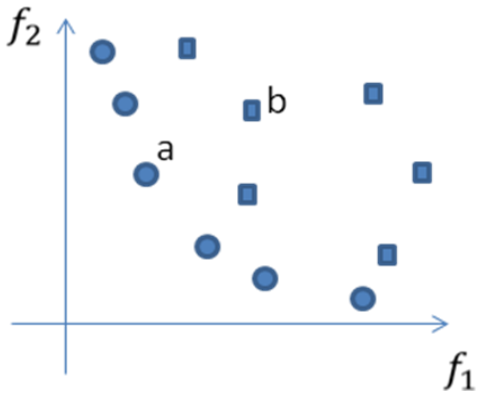
Большинство доступных нам инструментов автоматического подбора эффективных опций компилятора работают только с одной целевой оптимизируемой функцией, такой как производительность, размер кода, время компиляции и т.д. Но часто возникают ситуации, когда необходимо оптимизировать не по одной целевой оптимизируемой функции, а сразу по нескольким. Каждая из таких оптимизируемых функций далее будет называться критерием. Иногда можно найти решение, которое оптимально по всем критериям. Однако гораздо чаще возникает неприятная ситуация, когда критерии не согласуются друг с другом. В таких случаях решением является компромисс, приемлемый для всех критерий.

2.4.1 Существующие алгоритмы многоэкстремального поиска

Основные методы решения многокритериальных оптимизационных задач основаны на следующих принципах.[44-49]

- Выбор главного критерия.
- Правило «близости к идеалу».
- Метод последовательных уступок.
- составные критерия: аддитивный или мультипликативный.
- Правило паритета.
- Использование принципа Парето.

Во многих случаях, как показали апробации и анализ, самыми удобными и эффективными методами являются методы на основе Парето границе, так как наилучшим образом подходят для поиска рациональных решений ввиду отображения на фронте Парето всех возможных вариантов для последующей помощи в выборе стратегии. В этом случае хорошо известны алгоритмы метаэвристического класса, позволяющие, не исследуя полностью область допустимых решений, с определенной вероятностью находить глобальный экстремум в задачах с одним критериям и граница Парето в многокритериальных. Опишем основные принципы Парето-оптимальности[58].



- На плоскости точка a является Парето-доминантом по отношению точки b если ее значение «лучше» на одной из целевых функций a на другой «лучше или равно».
- Парето границей называется множество всех точек для которых не существует другая точка являющееся Парето доминантом по отношению к ней.
- Задача по нахождению Парето-оптимального решения для множества, эквивалентна задаче по нахождению Парето границы для этого множества.

2.4.1 Эволюционное многокритериальное вычисление

Составление множества Парето обычно вычислительно дорогое и зачастую невозможное, так как сложность основных приложений предотвращает точные методы их применения. По этой причине был разработан целый ряд стратегий стохастического поиска, таких как эволюционные алгоритмы, табу поиска, имитация отжига и оптимизация колонии муравьев. Обычно они не гарантируют определения оптимального компромисса, но пытаются найти хорошее приближение, т.е. набор векторов решений, цель, которая не слишком далека от оптимального вектора цели.

Общий стохастический алгоритм поиска состоит из трех частей: I. Оперативная память, которая содержит текущие оптимальные решения, II. Выбор модуля и III. изменение модуля.

В выбор модуля входят этапы оценки решений и экологический отбор. Оценка решений направлен на сбор перспективных решений для изменений и обычно проводится в рандомизированном подходе. Экологический отбор определяет, какие из сохраненных ранее решений, а также вновь созданных, будут храниться во внутренней памяти. Изменение модуля принимает множество решений и систематически или случайным образом изменяет эти решения в целях получения потенциально более эффективных решений. Таким образом, одна итерация стохастического оптимизатора включает последовательные шаги спаривания отбора, изменение модуля, экологический отбор; выполнение этого цикла может повторяться до определенного критерия остановки.

Многие стохастические стратегии поиска были первоначально предназначены для однокритериальной оптимизации, поэтому рабочая память содержит только одно решение. Как следствие: не спаривание отбора необходимо, и изменение осуществляется путем изменения текущих вариантов решения.

Эволюционный алгоритм характеризуется тремя особенностями:

1. ведется выбор кандидатов на решение,
2. производится оценка на этом множестве,
3. несколько решений могут быть объединены в условиях рекомбинации для генерации новых решений.

По аналогии с естественной эволюцией кандидаты на решения называются особями, а множество решений кандидатов называется популяциех. Каждая физическая особь представляет возможное решение, т.е. вектор решения, к текущей проблеме.

Процесс оценки решений обычно состоит из двух этапов: вычисление приспособленности (фитнесс) и взятия проб. На первом этапе особи в текущего поколения оцениваются, а затем им присваивается скалярное значение-фитнес, отражающее их качество. Далее родительский пул создается путем селекции из популяции в соответствии со значениями фитнеса. Например, широко используется метод бинарной турнирной селекции, где из двух особей, случайно выбранные из популяции, особеь лучшим фитнесом, копируется в родительский пул. Эта процедура повторяется до заполнения родительского пула. Тогда эти изменяющие операторы будут применяться в родительском пуле. Генетические алгоритмы, обычно имеют два из этих операторов, а именно рекомбинация и мутация. Рекомбинация это оператор, который занимает определенное число родителей, и создает заранее установленное количество детей путем объединения частей из родителей. Чтобы имитировать стохастическую природу эволюции, вероятность кроссовера связана с этим оператором. С другой стороны, оператор мутация изменяет особей путем изменения мелких деталей в присоединенных векторах по заданной вероятности мутаций. Нужно отметить, что из-за случайных эффектов некоторые особи в родительском пуле не могут быть затронуты изменениями и поэтому просто представляют собой копии ранее сгенерированного решения.

Наконец, экологический отбор определяет, какие особи населения из модифицированного родительского пула формируют новую популяцию. Простейшим способом является использование последнего населения в качестве новой популяции. Альтернативой можно объединить оба набора и так, детерминировано выбрать лучшие особи для выживания. Существуют и другие возможности, которые не будут обсуждаться здесь подробно.

В последние десятилетия генетические алгоритмы получили большую популярность и имеются множество исследований посвещённых им. Множество исследований ведутся также по разработке генетических алгоритмов поиска по нескольким критериям. Разработаны различные подходы поиска решений, оптимальных по Парето в задачах многокритериальной оптимизаций. Опишем некоторые известные алгоритмы многокритериального поиска[63-70].

- SPEA[82] и SPEA2 (Eckart Zitzler, Marco Laumanns, and Lothar Thiele, 2001). Алгоритм использует правило отбора особей популяции основанное на методе Парето силы (Pareto strength). Модернизацией генетического алгоритма SPEA является алгоритм SPEA2, который имеет некоторые преимущества по сравнению с SPEA, например можно выделить использование арихива фиксированной размерности.
- VEGA (Vector Evaluated Genetic Algorithm)[71-75] был разработан в 1984 году Шафером. В предложенном алгоритме для каждого критерия производиться отдельная селекция, тем самым популяция полученная на промежуточном этапе заполняется равными порциями особей, выбранных по каждому критерию.
- FFGA[76-78] был разработан Фонсекой и Флемингом в 1993 году и представляет собой алгоритм многокритериального отбора особей основанный на принципе Парето-доминирования. Ранг каждой особи определяется числом доминирующих его особей.

- MOGA[66] (Multi-ObjectiveGeneticAlgorithm), так же как и FFGA метод предложенный Фонсека и Флемингом в 1993 г. В описанном методе ранг каждой особи определяется числом особей популяции над которыми он доминирует.
- NPGA[79-81] (Niched Pareto Genetic Algorithm) был разработан Хорном в 1994 году отличался от предложенных ранее методов тем, что имеет механизм поддержки. Метод основан на формировании популяционных ниш.
- NSGA[67] и NSGA-II (Fast and Elitist Multiobjective Genetic Algorithm for Multi-Objective Optimization (Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan) разработан в 2002 году и является развитием метода недоминируемой сортировки, в котором при формировании архивов исключают те особи, расстояние которых до остальных особей архива не превышает заданной заранее значения.

Очень часто наиболее эффективными и удобными методами многокритериальной оптимизации являются методы на основе Парето-доминирования, так как подходящим образом применимы для подбора компромиссных решений ввиду отображения на границе Парето всех возможных вариантов для последующей помощи в выборе стратегии. Известны также некоторые метаэвристические методы, позволяющие, не исследуя полностью область допустимых решений, с определенной вероятностью находить решения близкие к глобальному экстремуму в задачах поиска по одному критерию и границу Парето в многокритериальных задачах.

Проводился сравнительный анализ между вышеперечисленными методами многокритериальной оптимизации. Было установлено, что SPEA2 и NSGA-II показывают лучшие результаты. В контексте данной задачи, где формируется

выпуклая граница Парето, алгоритм SPEA2 показывает лучшую сходимость и скорость.

2.4.2.Алгоритм SPEA2

При исследовании доступных алгоритмов многокритериального поиска было решено использовать алгоритм SPEA2 с некоторыми модификациями. Для сохранения ранее полученных эффективных наборов оптимизаций используется архив текущих “лучших” результатов, который с каждым поколением улучшается. Чтобы исключить большую концентрацию точек на границе Парето, был применен метод кластеризации K-средних , преобразованный с целью сохранения крайних точек на границе Парето. В результате применения метода получаем границу Парето, которая содержит не только самые эффективные точки по производительности и размеру бинарного кода, но и решения, являющиеся компромиссными между ними.

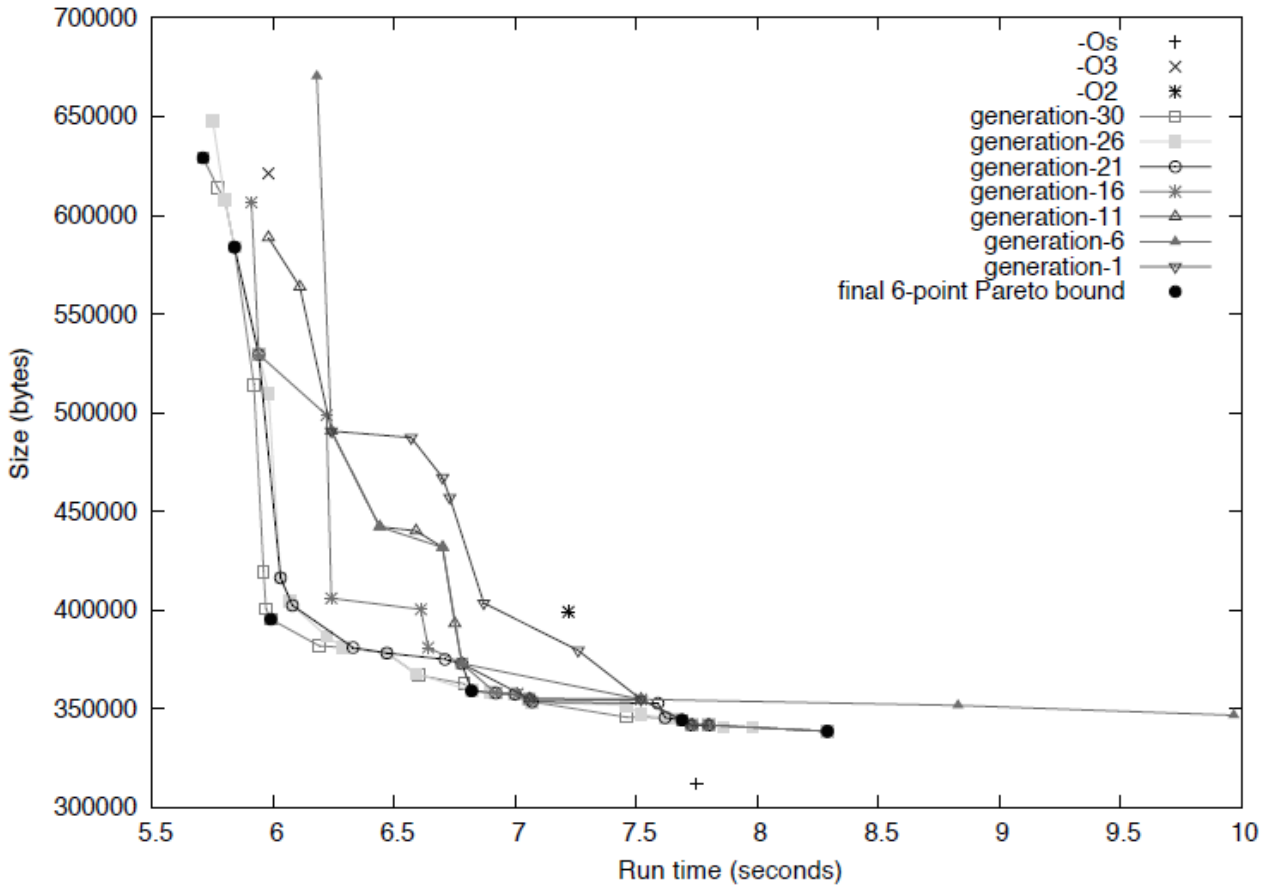


Рис. 3. Улучшение графика Парето с 1-е по 30-е поколение для оптимизации приложения x264 по производительности и размеру кода соответственно.

Эккарт Цитцлер, Марко Ломаннс и Лотар Тиеле разработали алгоритм поиска в многокритерилальном пространстве, использующий архив текущих "лучших" решений. Данный алгоритм основан на понятии Парето доминирования и называется SPEA (Strength Pareto Evolutionary Algorithm). В дальнейшем этот алгоритм был усовершенствован в том числе в плане устойчивости решений и получил название SPEA2. При изучении существующих алгоритмов многокритериального поиска решений, данный алгоритм был выбран в качестве базового для разработанного "метода автоматического подбора эффективных оптимизаций компилятора", так как он имеет лучшую скорость сходимости и более устойчив при выпуклой границе Парето. Описание алгоритма на высоком уровне:

Вход: N (размер популяции), M (размер архива), T (число поколений)

Выход: A (множество недоминированных особей)

1. Инициализация: сгенерировать произвольную популяцию P_0 и пустой архив A_0. Назначить t = 0.
2. Вычисление приспособленности: вычислить значение «fitness»-а для всех особей текущей популяции P_t и архива A_t.
3. Обновление архива: используя текущий архив и популяцию создать архив для следующего поколения A_{t+1}.
4. Завершение: Если t >= T (количество поколений), завершить работу алгоритма и на выходе получить множество P_{t+1}.
5. Размножение: выполнив операции мутации и скрещивания над особами архива A_{t+1} получить популяцию P_{t+1}. Назначить t = t+1. Перейти на шаг 2.

Данный алгоритм имеет некоторые преимущества по сравнению с другими алгоритмами многокритериального поиска, которые в процессе “подбора эффективных оптимизаций компилятора” делают более устойчивым и обеспечивают хороший уровень сходимости.

Некоторые особенности алгоритма SPEA2:

- Улучшенная схема оценки приспособленности (fitness) особи, которая для каждого индивида вычисляет количество особей Парето-доминирующих и Парето-доминированных по отношению к нему.
- Введена техника вычисления плотности особей на основе метода классификации “k ближайших соседей”, которая позволяет точнее управлять процессом поиска.
- Хранение архива фиксированной длины, из которой при переполнении удаляются те особи с текущей границы Парето, имеющие меньший приоритет.

2.4.3 Элитаризм

Одним из возможных инструментов ГА является элитаризм. В генетических алгоритмах с использованием концепции элитаризма наиболее приспособленные особи предыдущего поколения сразу попадают в следующее поколение или сохраняются в специальный контейнер под названием архив. Эти особи называются элитными. Таким образом, повышается использование найденных решений, так как часто хорошее решение полученное в ранних поколениях может быть не достигнуто в последнем итоговом поколении. Также улучшается скорость сходимости ГА, поскольку при генерации популяций нового поколения особи архива обычно имеют больше шансов быть выбраны при селекции, таким образом повышая среднее значение функций приспособленности в пределах данной популяции. На рис. 4 видим, как результат, полученный на 17-ом поколении так и не был достигнут и без концепции элитаризма он был бы потерян.

Элитаризм особенно полезен для ГА с многокритериальным поиском. В генетических алгоритмах с многокритериальным поиском результатом является набор недоминированных особей и сохраняя архив недоминированных особей полученных в ранних поколениях, граница Парето будет постоянно улучшатся (ведь во время скрещивания особей могут быть выбраны индивиды из разных частей границы Парето и их потомство может получится хуже).

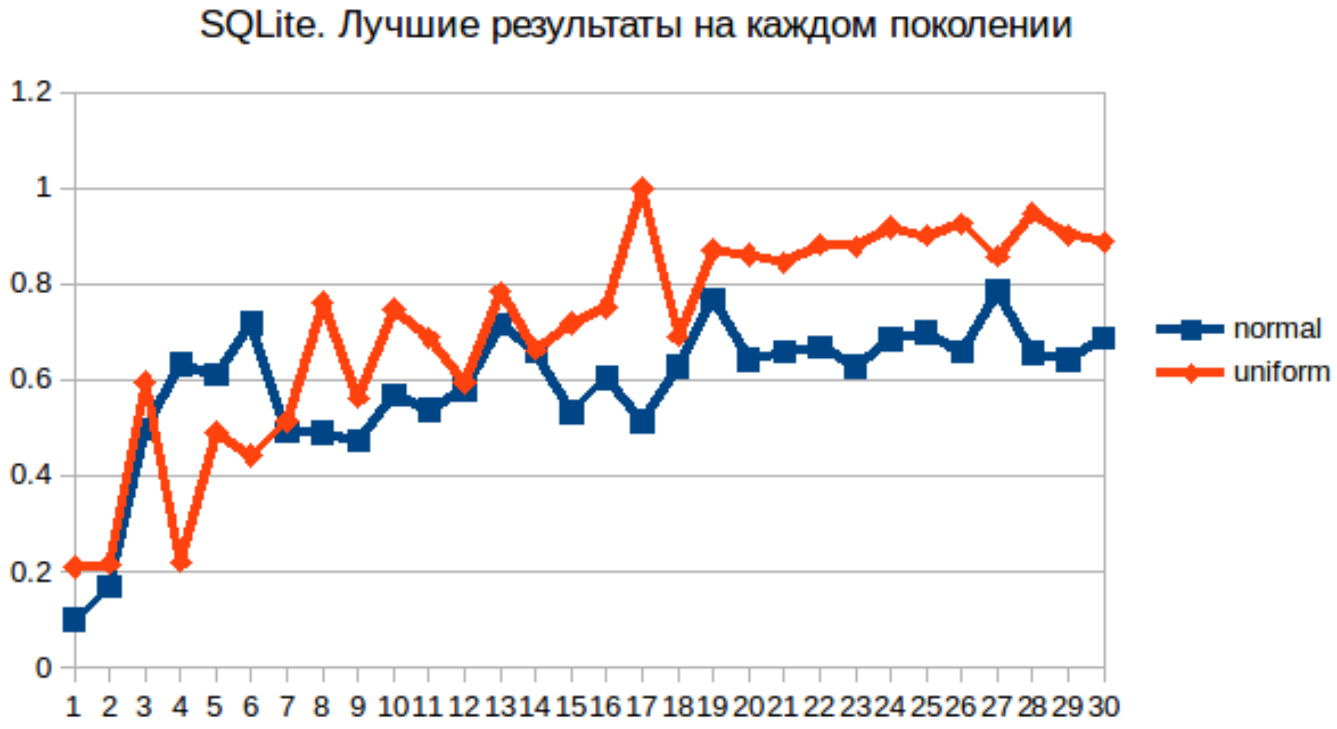


Рис. 4. Лучшие результаты улучшения производительности для каждого поколения при нормальном и равномерном распределениях , 0 соответствует -O2, 1 - максимальной производительности, достигнутой инструментом.

2.4.4 Кластеризация границы Парето

На рис. 3 продемонстрирован процесс улучшения границы Парето на примере приложения x264, с использованием компилятора GCC, работающего на архитектуре ARM. Как видно из рисунка, граница Парето состоит из большого

количества точек (часто несколько десятков), составляющие множество недоминированных особей. Очень часто эти точки могут скопиться в нескольких кучах (близко друг от друга) и для пользователя не имеет практического смысла получать в качестве результата все эти особи, достаточно иметь только одну (точку центра скопления). Для того, чтобы получить из множества Парето значимые особи, используется метод кластеризации k-средних (k-means).

Метод k-средних[83] – это метод кластерного анализа, цель которого является разделение множества данных решений (точек) на пространстве на k кластеров, при этом каждая точка относится к тому кластеру, к центу (центроиду) которого оно ближе всего. В качестве меры близости используется Евклидово расстояние. В контексте нашей задачи, этот метод позволяет из точек на границе Парето выбрать небольшое число самых значимых решений. Поскольку при многокритериальном (производительность/размер кода) подборе опций компилятора крайние точки на границе Парето представляют особый интерес (лучшее производительность и лучший размер кода), алгоритм k-means был модифицирован. В результате корректировки алгоритма, крайние точки Парето границы всегда включены в итоговый “кластеризованный” результат в качестве центра кластеров, независимо от имеющих место скоплений.

2.5. Метод автоматического подбора опций и параметров компиляции

Как отметили выше, в качестве базового генетического алгоритма поиска данных был выбран SPEA2. В представленной работе данный алгоритм был доработан для удовлетворения требованиям как многокритериального поиска, так и поиска по одной целевой функции. Кроме того к этому алгоритму добавлена возможность поддержки нескольких популяций параллельно, что позволяет повысить эффективность метода.

В алгоритме SPEA2 используется концепция элитаризма, при которой наиболее приспособленные особи предыдущих поколений сохраняются в архиве, повышая

использование ранее найденных решений. Размер архива в SPEA2 не изменяется в процессе работы алгоритма.

В данной работе используются некоторые термины, заимствованные из генетических алгоритмов, в том числе из SPEA2, такие как поколение, популяция, особь, скрещивание, мутация, ген и т.д. В контексте предложенного метода подбора опций компилятора гены представляют собой булевы значения применяемых опций (-f/-fno, оптимизация включается или нет) и целочисленные значения числовых параметров компиляции (--params), особи – наборы опций и параметров компиляции заданной неизменной длины, популяция - множество особей определенной размерности.

В разработанном в рамках диссертационной работы методе проведены ряд изменений и модификаций в примененном алгоритме эволюционного поиска SPEA2. Приведем основные особенности и отличия предложенного в методе алгоритма от базового SPEA2.

- Разработан подход, который позволяет использовать несколько параллельных популяций одновременно и применить оператор миграции между ними.
- В отличии от базового алгоритма, данный метод поддерживает поиск как по одному критерию так и по нескольким.
- Разработан механизм объединения архивов популяций для каждого поколения в один общий архив.
- При селекции[84] выбор родительских особей проводится как среди особей архива предыдущего поколения, так и среди особей текущей популяции (в SPEA2 при селекции используются только особи архива).
- Использован метод кластеризации точек на границе Парето при многокритериальном поиске.

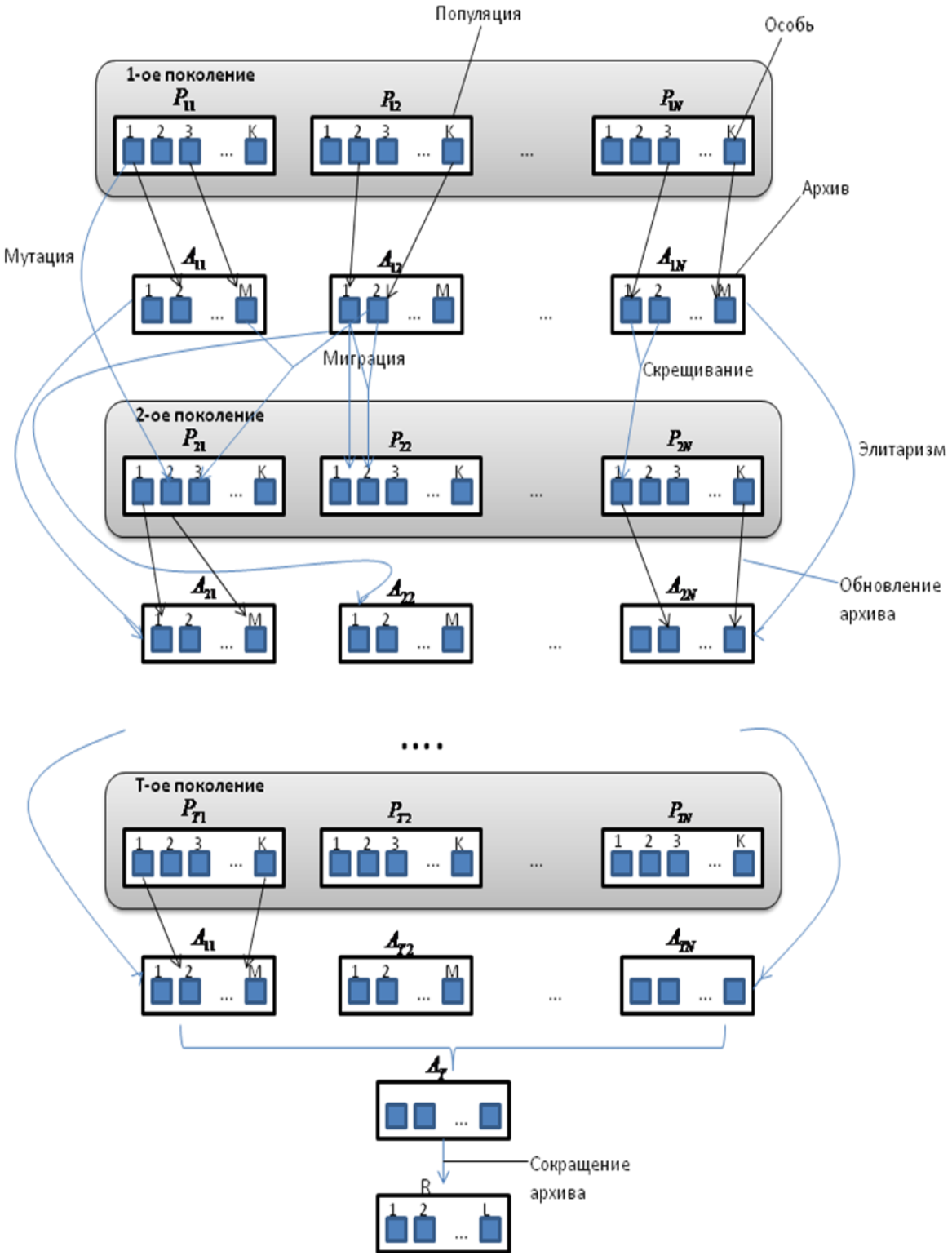


Рис. 5. Схемотическая демонстрация метода подбора опций и параметров компилятора.

Опишем разработанный в данной работе метод автоматического подбора оптимизаций компилятора для конкретного приложения и целевой архитектуры, который схематически продемонстрирован на рисунке 5.

Вход: N (число популяций), K (количество особей в одной популяции), M (размер архива), T (максимальное число поколений), S (стратегия эволюционного поиска), L (выбранное число решений)

Выход: R (множество предлагаемых решений)

1. ***Инициализация первого поколения.*** *Создать произвольные популяции* $P_{11}, P_{12,\dots}P_{1N}$. *Для каждой популяции* P_{1i}, *где* $i = 1 \dots N$ *сгенерировать K наборов опций и параметров компилятора и создать пустой архив* A_{1i}. *Присвоить t = 1.*
2. ***Вычисление приспособленности***. *Вычислить значение приспособленности для всех особей текущих популяций* $P_{t1}, P_{t2}, \dots P_{tN}$. *В случае стратегии многокритериального поиска применить принцип Парето силы, предложенный в SPEA2, а при поиске по одному критерию значение приспособленности соответствующего вычисленному результату (производительность, размер кода и т.д.) на целевой платформе.*
3. ***Обновление архивов.*** *При t> 1, используя текущие популяции* P_{ti} *и архивы предыдущего поколения* A_{t-1i},*где* $i = 1 \dots N$ *обновить архивы* A_{ti} *для текущего поколения. В случае многокритериального подбора включить все недоминированные особи из* $P_{ti} \cup A_{t-1i}$ *в архив* A_{ti}. *Если размер* A_{ti}

меньше M, включить доминированные особи с наибольшим значением приспособленности до полного заполнения. Если размер архива больше M *использовать оператор исключение, предложенный в SPEA2. При поиске по одному критерию включить в* A_{t+1i} *M наиболее приспособленных особей из* $P_{ti} \cup A_{t-1i}$.

4. ***Объединение архивов и уменьшение числа текущих решений.*** *Объединить все текущие архивы* $A_{ti}, где\ i = 1 \dots N$ *в один архив* $\overline{A_t}$. *При многокритериальном поиске использовать метод кластеризации для сокращения числа скоплений точек на границе Парето (соответствующей архиву* $\overline{A_t}$*) до L центров и включить сокращенное множество в* $\overline{R_t}$. *В случае поиска по одному критерию включить в* $\overline{R_t}$ L *наиболее приспособленных особей из архива* $\overline{A_t}$.

5. ***Завершение:*** *Если t>= T (количество поколений) или достигнуты ограничения по другим критериям, завершить работу алгоритма, присвоить* $R = \overline{R_t}$ *и на выходе получить множество R.*

6. ***Создание популяций нового поколения.*** *Для каждой популяции* P_{ti} *и архива* A_{ti}, $где\ i = 1 \dots N$ *с помощью бинарной турнирной селекции выбрать приспособленные особи и применяя над ними операторы мутации и скрещивания, а также используя миграцию из особей архивов других популяций получить новую популяцию* P_{t+1i}. *Присвоив t = t+1, перейти на шаг 2.*

На шаге 1 алгоритм генерирует NxK (количество поколений и популяций) особей и для каждой из них на шаге 2 за конечное время вычисляет значения приспособленности (компиляция, запуск на целевой машине и в зависимости от выбранной критерии поиска вычисление числового значения результата). На шагах 3 и 6 обновляются N архивов и популяций следующего поколения за конечное время, поскольку размер архива и популяции всегда фиксированной длины и новые особи получаются через бинарную турнирную селекцию из конечного множества. На шаге 4 из N архива за полиномиальное время (в том числе, алгоритм кластеризации) вычисляются L наиболее приспособленных особей. И наконец на шаге 5 есть ограничение на количество итераций (T) и поскольку каждая итерация выполняется в конечное время, алгоритм конечный.

Выходной результат (лучшие наборы опций компилятора для конкретного приложения) корректен, так как при работе алгоритма в следующее поколение (и архив) попадают только те особи, которые удовлетворяют ограничениям тестового приложения и компилятора, а результат это набор наиболее приспособленных особей из архива последнего поколения.

2.6. Выводы

1. Построенамодификикация и адаптация известного генетического алгоритма SPEA к методу автоматического подбора опций компилятора и параметров компиляции для конкретного приложения на заданной платформе как для одной целевой функции оптимальности, так для поиска по нескольким критериям.
2. Основные отличия разработанного модификации от базового SPEA2:
   - Одновременное рассмотрение нескольких параллельных популяций и использование оператора миграции между параллельными популяциями для обмена генами.

- Поиск как по одному критерию, так и по нескольким.
- Объединение архивов популяций для каждого поколения в единый общий архив.
- При селекции выбор родительских особей проводится как среди особей архива предыдущего поколения, так и среди особей текущей популяции.
- Кластеризация точек на
  границе Парето при многокритериальном поиске.

3. Приведено доказательство корректности полученного алгоритма.

Глава 3

Методы, позволяющие автоматически улучшить применимость ряда оптимизаций компилятора GCC.

3.1. Оприимизации на этапе компоновки в GCC

В компиляторной инфраструктуре GCC при компиляции с использованием опции –flto [91,92] в объектный файл дополнительно записывается некое промежуточное представление исходного кода (GIMPLE). Эта дополнительная информация используется на стадии компоновки. В итоге имеет место тот же эффект, как если бы весь исходный код находился в одном файле.

Промежуточное представление на стадии компиляции записывается в специальную ELF-секцию объектного файла (посредством библиотеки Libelf). Рассмотрим секции ELF.

- **.rodata** – *Read-Only Data Section* – включает те данные, которые предназначены только для чтения. GCC сохраняет глобальные переменные отмеченные константами в отдельную секцию, которая называется .rodata. .rodata также испоьзуется для хранения строковых констант. .rodata - секция констант и должна размещаться при загрузке в память, которая только для чтения.
- **.text** – *Text Section* - содержится исполняемый код программы.

- **.hash** - хэш символьной таблицы. Хэш таблица используется для ускорения доступа к таблице символов (.dynsym).
- **.data** — содержит те глобальные и локальные переменные, которые инициализированы.
- **.dynsym** - включает все экспортированные и импортированные символы.
- **.dynstr** – содержит таблицу строк.
- **.rel.dyn** – переразмещения для динамически компонуемых функций (только если PLT не используется)
- **.rel.plt** – список элементов в PLT (Procedure Linkage Table), подлежащих переразмещению при динамической компоновке (используется PLT)
- Кодовый сегмент состоит из секций **.init** (процедуры инициализации), **.plt** (секция связок), **.text** (основой код программы) и **.finit** (процедуры завршения).
- **.eh_frame** — информация, необходимая для frame-unwinding во время обработки исключений.
- **.bss** — секция неинициализированных глобальных и локальных переменных, то есть заполненных нулями.
- Секции “.**rel.dyn**” и “**.rel.plt**” являются таблицами переразмещений для тех символов из “**.dynsym**”, для которых вообще необходимо переразмещение при компоновке.

В некоторых случаях в исходном коде приложения в связи с некорректными объявлениями области видимости функций могут возникнуть определенные проблемы во время выполнения оптимизаций на этапе компоновки.

В объектных файлах формата ELF внутренние функции в динамических библиотеках (shared object) имеют скрытую видимость (hidden visibility). Это позволяет внутренним функциям избежать вызовы через таблицу компоновки процедур (PLT), что в свою очередь позволяют компилятору лучше оптимизировать их. PLT (Procedure Linkage Table) — это таблица компоновки процедур. Она присутствует в исполняемых и разделяемых модулях. Это массив заглушек, по одной на каждую импортируемую функцию.

Однако иногда функции могут экспортироваться, но одновременно использоваться внутри библиотеки. Такие функции не могут иметь “скрытую видимость”, в связи с тем, что экспортируются, но одновременно желательно избежать вызовы через PLT. В компиляторе GCC все функции по умолчанию не имеют “скрытую видимость” (экспортируются) и вызовы через PLT могут препятствовать многим межмодульным оптимизациям внутри библиотеки. Чтобы избежать этого разработчики или пользователи приложений должны сами явно указывать те функции, которые будут экспортированы. При компиляции с GCC в заголовочные файлы можно поставить атрибут __attribute__((visibility ("default"))) в объявления тех функций, которые должны быть глобальными. Используя опцию -fvisibility=hidden компилятор GCC сделает все функции со скрытой видимостью, кроме тех которые были явно объявлены этим атрибутом. В GCC сейчас также поддерживается директива #pragma с объявлением области “видимости” функций для участка своего действия:

Существуют также другие ограничения для выполнения оптимизаций на этапе компоновки. На версиях libtool (<2.4) опция -flto не передается компиновшику и соответственно libtool нужно обновить.

3.2. Метод автоматического улучшения выполнимости оптимизаций на этапе компоновки.

Решение вышеперечислинных проблем вручную является трудоемким для больших приложений и для решения данной задачи был разработан метод, на

основе чего реализован инструмент, позволяющий автоматически решить данную задачу. В дальнейшем данный инструмент был интегрирован в разрабатываемое в ИСП РАН набор инструментов TACT.

Опишем метод автоматической проверки и корректировки исходного кода библиотек программ, обеспечивающий улучшение эффективности выполнения оптимизаций на этапе компоновки.

На входе инструмент получает исходный код программы, после чего выполняются следующие этапы. На первом этапе исходная программа компилируется с базовым уровнем оптимизаций –O2 и запускается на целевой машине. Одновременно проверяется версия libtool и при версии <2.4 завершается работа инструмента с ошибкой. Далее исходная программа компилируется с опциями –O2 –flto (включаются оптимизации на этапе компоновки) и запускается на целевой машине. С помощью специального скрипта вычислить те заголовочные (header) файлы, которые содержат объявления экспортируемых (внешних) функций. В данных заголовочных файлах добавляется явная область видимости, с помощью директивы #pragma:

```
#pragma GCC visibility push(default)

// участок кода

#pragma GCC visibility pop
```

На последнем этапе компилируется модифицированная исходная программа с опциями “-O2 -fvisibility=hidden -flto” и выходе получаем исполняемый файл (обычно динамическую библиотеку формата .so), содержащий более эффективный машинный код.

В результате работы автоматического метода на этапе компоновки применяемость оптимизаций значительно улучшается, и в зависимости от приложения,

получается рост производительности и уменьшение размера бинарного кода до нескольких десяток процентов.

3.3. Улучшение применяемости ряда оптимизаций

По результатам работы TACT с компилятором GCC на платформе ARM на используемых тестах были выбраны оптимизации компилятора GCC для дальнейшего исследования и ручного анализа. В данном разделе рассматриваются самые важные улучшения, произведенные в оптимизациях компилятора GCC.

3.3.1. Преобразование ветвлений

Для тестового приложения sqlite инструмент TACT показал что опция -fno-if-conversion дает значительный прирост производительности. Данная опция выключает используемую по умолчанию оптимизацию преобразования ветвлений, при которой условные переходы удаляются, а все команды в базовых блоках, выполнение которых зависело от этого перехода, защищаются соответствующим предикатом[85]. Преобразование применяется только к тем базовым блокам, которые не содержат вызовов функций и длинных цепочек зависимостей по данным. Такое преобразование уменьшает количество сбросов конвейера и дает другим оптимизациям (например, планировщику инструкций) большую свободу в преобразовании кода и нахождении параллелизма на уровне инструкций. Преобразование ветвлений требует поддержки со стороны процессора: необходимо наличие условной формы инструкций, которые выполняются, только если предикат выполнен, и работают как инструкция nop в противном случае. На платформе ARM любая инструкция может быть сделана условной по значению флагов. Например, moveq выполнит mov только если установлен флаг нулевого результата. Были исследованы причины того, что эта оптимизация приводит к снижению производительности на ARM. Изучение получаемого ассемблерного кода показало, что проблему можно свести к следующему примеру:

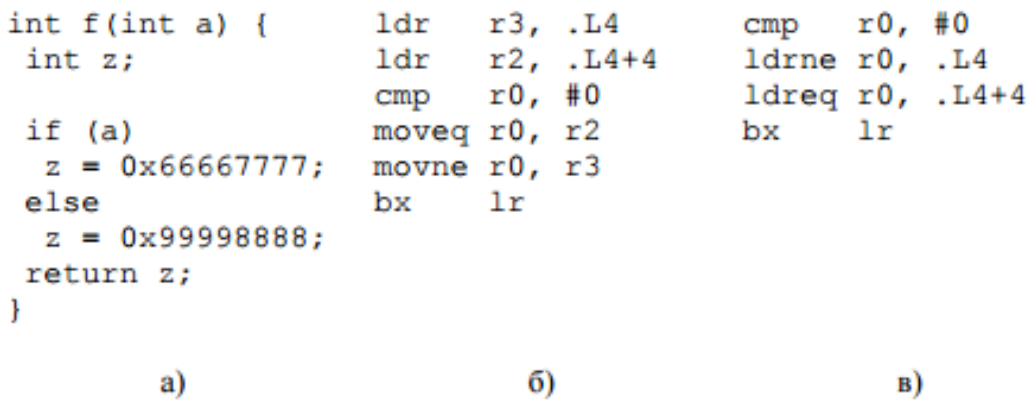


Рис. 6. Пример неоптимального преобразования ветвления.

а) Исходный код на C; б) ARM-ассемблер для -O2; в) ARM-ассемблер для -O2 -fno-if-conversion

В коде ассемблера на рис. 6(б) две загрузки из памяти выполняются всегда (без предикатов), хотя в дальнейшем используется результат только одной из них. В коде на рис. 6(в) такой недостаток отсутствует, поэтому он выполняется быстрее. Причина создания данных двух загрузок – длинные целые константы в исходном коде на C. Опция –fif-conversion включает в GCC оптимизационные проходы ce1 и ce2, которые выполняются до распределения регистров. Аналогичный оптимизационный проход ce3, включаемый опцией –fif-conversion2, выполняется после распределения регистров. Причина изучаемой проблемы была выявлена в оптимизационном проходе ce1. Используется несколько методов преобразования ветвлений, и один из них – это замена присваиваний из базовых блоков ветвления в одно присваивание, которое использует RTL-выражение if-then-else. Впоследствии данное выражение преобразуется в условную команду пересылки, как в примере выше. Компилятор пытается предсказать, будет ли возможно создать условную команду пересылки. Если аргумент – длинная константа, которая не может быть подставлена в инструкцию как непосредственное значение, простая условная команда пересылки не подходит, и используется безусловная загрузка длинной константы с последующей условной пересылкой из регистра. Для вышеописанного примера, RTL-код выглядит следующим образом:

```
r137:SI=0xffffffff99998888
r138:SI=0x66667777
cc:CC=cmp(r135:SI,0x0)
r133:SI={(cc:CC==0x0)?r137:SI
:r138:SI}
```

а)

```
r3:SI=0x66667777
r2:SI=0xffffffff99998888
cc:CC=cmp(r0:SI,0x0)
(!cc:CC) r0:SI=r2:SI
 (cc:CC) r0:SI=r3:SI
```

б)

Рис. 7. RTL-код при преобразовании ветвлений.

а) после прохода ce1 ; б) после прохода split2

В этом коде, выражение “r133:SI={(cc:CC==0x0)?r137:SI:r138:SI}” описывает условную команду пересылки, которая преобразуется в RTL-код на рис. 7(б) оптимизационным проходом split2. Если включен только один оптимизационный проход ce3, работающий после распределения регистров, то к моменту его работы компилятор уже создает необходимые инструкции загрузки длинных констант в регистр и может их преобразовать в условную форму. Очевидным решением кажется отключение ранних оптимизационных проходов преобразования ветвлений и включением только ce3 прохода с помощью опций –fno-if-conversion –fif-conversion2. Однако, данная комбинация отключит часть полезных преобразований, выполняемых оптимизационным проходом ce1. Например, ce1 может преобразовать выражение “x > 0 ? x : 0”, которое обычно требует трех инструкций (сравнение, условное ветвление и пересылка) в инструкцию “bicle r8, r0, r0, asr #31” (ее семантика такова: очистить в регистре r0 те биты, которые равны 1 во втором аргументе, и сохранить результат в регистре r8; здесь второй аргумент равен r0 asr #31, который из определения арифметического сдвига равен 0 либо 0xffffffff, в зависимости от знака числа в регистре r0). Решение о том, следует ли использовать промежуточный регистр при загрузке константы вне условной пересылки, принимается в функции emit_conditional_move. Но для архитектуры ARM, решение иногда оказывается неверным, например в вышеописанном случае. Данный недочет был исправлен, причем были сохранены преимущества оптимизаций выполненных проходом ce1, но с защитой от

создания безусловных загрузок. До вызова функций emit_conditional_move производится проверка необходимости использования промежуточного регистра, и если такая необходимость есть, то преобразование на оптимизационном проходе ce1 отменяется, и только во время работы прохода ce3 загрузки преобразуются в предикатную форму. Описанные изменения включаются с помощью опции -fif-conversion-no-unconditional-moves. Результаты тестирования показали, что с введенной опцией компилятор для некоторых тестов создает улучшенный код, и не было найдено тестов, на которых наблюдается снижение производительности. Таким образом, мы улучшили работу на платформе ARM платформонезависимой оптимизации преобразования ветвлений в компиляторе GCC. Ускорение для тестовых приложений оказалось небольшим, для sqlite оно составило 0.5%. Причина этого в том, что подходящие условные инструкции не встречаются в горячих местах кода в исследуемых тестовых приложениях.

3.3.2. Предварительная загрузка данных

Предварительная загрузка данных (prefetching)[86] – технология, позволяющая ускорить работу программ за счет уменьшения количества промахов кэша. На многих современных архитектурах есть инструкция предварительной загрузки. Эта инструкция фактически является подсказкой процессору, она не влияет на логику работы программы. Предварительная загрузка работает почти как обычная инструкция загрузки из памяти, с той лишь разницей, что данные не попадают ни в один регистр. Запускается процесс передачи данных из памяти в кэш, и через некоторое число тактов (в это время выполняются другие инструкции) данные оказываются в кэше. При последующем обращении к этим данным пропадает необходимость длительного ожидания работы с памятью. Один из вариантов использования предварительной загрузки, предлагаемый в GCC, это использование функции __builtin_prefetch. На платформе ARM ее вызов будет заменен на инструкцию предварительной загрузки pld для адреса, переданного функции в качестве параметра. Также в GCC есть оптимизационный проход,

подключаемый с помощью опции -fprefetch-loop-arrays, который автоматически добавляет операции предварительной загрузки для обращений к элементам массива в циклах. В нем происходит анализ всех обращений в памяти происходящих на одной итерации цикла. В зависимости от множества параметров целевой платформы часть обращений выбирается для предварительной загрузки. Поскольку данные в кэше обновляются блоками по размеру линии кэша, обычно в оптимизационном проходе производится разворачивание цикла, чтобы для одного обращения к памяти в терминах исходного цикла было необходимо выполнить одну инструкцию предварительной загрузки на каждой итерации развернутого цикла.

Оптимизация предварительной загрузки данных в GCC имеет несколько параметров, контролирующих ее работу. Параметры определяют аппаратные характеристики кэша, такие как размер линии кэша, количество одновременно выполняемых загрузок, время задержки при передаче данных из памяти в кэш второго уровня и другие. Анализ результатов тестирования библиотеки libevas с помощью TACT показал, что эти параметры, и их интерпретация в компиляторе для генерации кода, сильно влияют на производительность. В первую очередь, следует отметить два параметра оптимизации предварительной загрузки данных, появившиеся в GCC версии 4.5. Это prefetch-min-insn-to-mem-ratio и min-insn-to-prefetch-ratio. Они были созданы в первую очередь для более консервативной оптимизации предварительной загрузки данных на некоторых тестах набора SPEC CPU 2000 на процессорах архитектуры x86. Первый параметр устанавливает ограничение на минимальное соотношение числа инструкции в цикле к числу обращений к памяти в нем, подразумевая, что если нет инструкций для выполнения на процессоре во время ожидания операций с памятью, то не будет улучшения производительности от предварительной загрузки. Но для многих циклов это не так, например обычный цикл копирования массива в памяти почти не содержит инструкций для долгого выполнения на процессоре, но может быть

значительно ускорен за счет предварительной загрузки данных. Второй параметр отключает предварительную загрузку в циклах с неизвестным числом итераций, если соотношение между числом предварительных загрузок и общим числом инструкций превосходит значение этого параметра. Поскольку без использования профилирования число итераций цикла редко оказывается известно, этот параметр успешно отключает предварительную загрузку на библиотеке libevas, устанавливая слишком высокое ограничение (значение по умолчанию равно 10). Можно сказать, что оба эти параметра оказываются неуместными, по крайней мере, на приложениях, исследованных в данной работе. Тестирование с помощью TACT показало, что оптимальным значением этих двух параметров на платформе ARM является ноль, это позволяет использовать все возможности оптимизации предварительной загрузки.

С помощью TACT было проведено исследование влияния параметра prefetchlatency на производительность тестовых приложений. Данный параметр должен означать количество тактов процессора, которое необходимо для предзагрузки данных из памяти в случае, когда эти данные отсутствуют в кэше. Теоретически, оптимальное значение prefetch-latency должно зависеть от конкретного используемого аппаратного обеспечения, но не от запускаемых программ. Однако, результаты работы TACT показали, что оптимальное значение данного параметра значительно отличается для выбранных тестовых приложений. Оказалось, что в оптимизации предварительной загрузки prefetch-latency используется для вычисления дистанции предварительной загрузки. Дистанцией предварительной загрузки называется, в случае платформы ARM, число байт, добавляемое к аргументу инструкции предварительной загрузки pld по сравнению с обычной загрузкой ldr для обращения к той же области в памяти на той же итерации. Для подсчета дистанции предварительной загрузки компилятор пытается оценить время, требуемое для выполнения одной итерации цикла. Далее, зная шаг у каждого из обращений к памяти и задержку предварительной загрузки

prefetch-latency, вычисляется дистанция предварительной загрузки. Ее максимально точное вычисление очень важно, ведь в случае, когда она слишком мала, данные не успеют оказаться в кэше к нужному моменту времени, а в случае, когда дистанция слишком велика, они могут быть стерты из кэша в результате записи туда других данных. Однако, реализованные в GCC вычисления не очень точны при определении времени выполнения одной итерации цикла. В частности, выполняется оценка времени на исходном (без развертывания итераций) теле цикла, после чего полученное значение умножается на коэффициент развертывания. Получается, что инструкции изменения счетчика цикла и ветвления подсчитываются несколько раз, хотя после развертывания большая часть из них будет удалена. В результате компилятор получает дистанцию предварительной загрузки большую, чем это требуется для архитектуры, особенно на коротких циклах. Это создает излишний дефицит кэша и не позволяет настраивать параметр prefetch-latency, поскольку его реальное влияние зависит от размера и шага конкретного цикла. Механизм оценки дистанции предварительной загрузки был улучшен с помощью подсчета временной оценки для уже развернутого цикла вместо исходного. На простых тестовых циклах выигрыш производительности составляет до 10%. Тем не менее, даже после улучшения, возможно получение более высокой производительности тестов при выборе оптимального значения параметра prefetch-latency с помощью TACT.

При анализе ассемблерного кода, сгенерированного при включенной оптимизации предварительной загрузки, было обнаружено, что дополнительный регистр используется как аргумент в каждой инструкции предварительной загрузки pld, в то время как требуемый адрес предварительно загружаемого места в памяти можно легко вычислять относительно аргументов инструкции загрузки ldr, для которой делается предварительная загрузка. Как следствие, требуется значительно больше регистров, причем разворачивание цикла усугубляет эту

проблему. Увеличение числа используемых регистров легко увидеть на следующем примере:

```
for (i=0; i<n; i++)
  s+=a[i]*b[i];
```

```
.L3:
mov r1, r2, asl
#2
ldr r5, [r3, r4]
ldr r6, [r3, ip]
add r2, r2, #1
cmp r2, r8
add r7, r1, r4
add r1, r1, ip
pld [r7, #116]
pld [r1, #116]
add r3, r3, #4
mla r0, r6, r5,
r0
bne .L3
```

```
.L3:
ldr ip, [r2, #0]
add r1, r1, #1
ldr r4, [r3, #0]
cmp r1, r5
pld [r2, #116]
pld [r3, #116]
add r2, r2, #4
add r3, r3, #4
mla r0, r4, ip,
r0
bne .L3
```

Рис. 8. Пример излишнего регистрового давления.

а) Исходный код на C; б) ARM-ассемблер до исправления; в) ARM-ассемблер после исправления

Удалось выяснить, что источник проблемы – оптимизация отслеживания изменений скалярных переменных (scalar evolution optimization pass), которая анализирует поведение скалярных переменных в циклах. Оказывается, данная оптимизация не позволяет разложить такое выражение, как &a[i] на сумму &a[0] и i, для дальнейшего анализа отдельно каждой части выражения. Этот недостаток является причиной того, что в дальнейшем при оптимизации индукционных переменных цикла оказывается невозможным вычисление адреса &a[i]с использованием того же регистра, что и для обращений к a[i]. Были разработаны улучшения для оптимизации скалярных переменных в циклах. Стало допустимым разложение адресного выражения. Эффект от внесенных изменений не ограничивается влиянием на оптимизацию предварительной загрузки данных, но также позволяет оптимизациям счетчиков цикла находить более полное множество индукционных переменных в циклах, где используются обращения к глобальным массивам или массивам внутри структур. Улучшение оптимизации дает прирост производительности библиотеки libevas на процессоре ARM в

среднем на 3.4% и до 11% на некоторых тестах. Кроме того, на 1% ускоряется набор тестов с использованием чисел с плавающей точкой SPEC FP 2000 на архитектуре x86 на процессоре Core 2 с использованием профилирования и следующих опций оптимизации: -O3 -ffast-math -march=native –fsched-pressure –fschedule-insns -fprefetch-loop-arrays. Для целочисленных тестов SPEC INT 2000 производительность не изменяется.

3.4 Выводы

1. Описаназадача оптимизации на этапе компоновки в компиляторной инфраструктуре GCC.
2. Приведены основные проблемы выполнении оптимизаций при компоновке.
3. Построен метод проверки и корректировки исходного кода библиотеки, позволяющий улучшить эффективность оптимизаций на этапе компоновки.
4. По результатам работы TACT с компилятором GCC на платформе ARM было выделено множество оптимизаций компилятора GCC для дальнейшего исследования и ручного анализа.
5. С помощью запуска выделенных оптимизаций определены самые эффективные улучшения, произведенные с помощью оптимизаций: преобразование ветвлений и предварительная загрузка данных компилятора GCC.

Глава 4

Реализация набора инструментов TACT на основе предложенных методов. Полученные основные результаты.

На основе предложенных выше методов разработан и реализован набор инструментов TACT (Toolkit for Automatic Compiler Tuning), которых состоит из множества модулей и призван для заданной программы и аппаратной архитектуры автоматически улучшить применимость оптимизаций компилятора. TACT поддерживает кросс-платформенность, параллельную компиляцию и выполнение и включает в себя как инструмент автоматического подбора опций компилятора и параметров компиляции (с возможностью выбора критерия поиска), так и инструмент проверки и корректировки исходного кода библиотеки для улучшения эффективности оптимизаций на стадии компоновки.

4.1. Реализация инструмента TACT

Программно-аппаратная система TACT состоит из управляющего модуля, выполняемом на компьютере x86, который используется для кросс-компиляции настраиваемых приложений, и нескольких тестовых плат (например, архитектуры ARM), доступных с управляющего компьютера по протоколу SSH и связанных таким образом, что все устройства имеют один общий каталог через сетевую файловую систему NFS. Общая схема устройства инструмента TACT показана на рис 9.

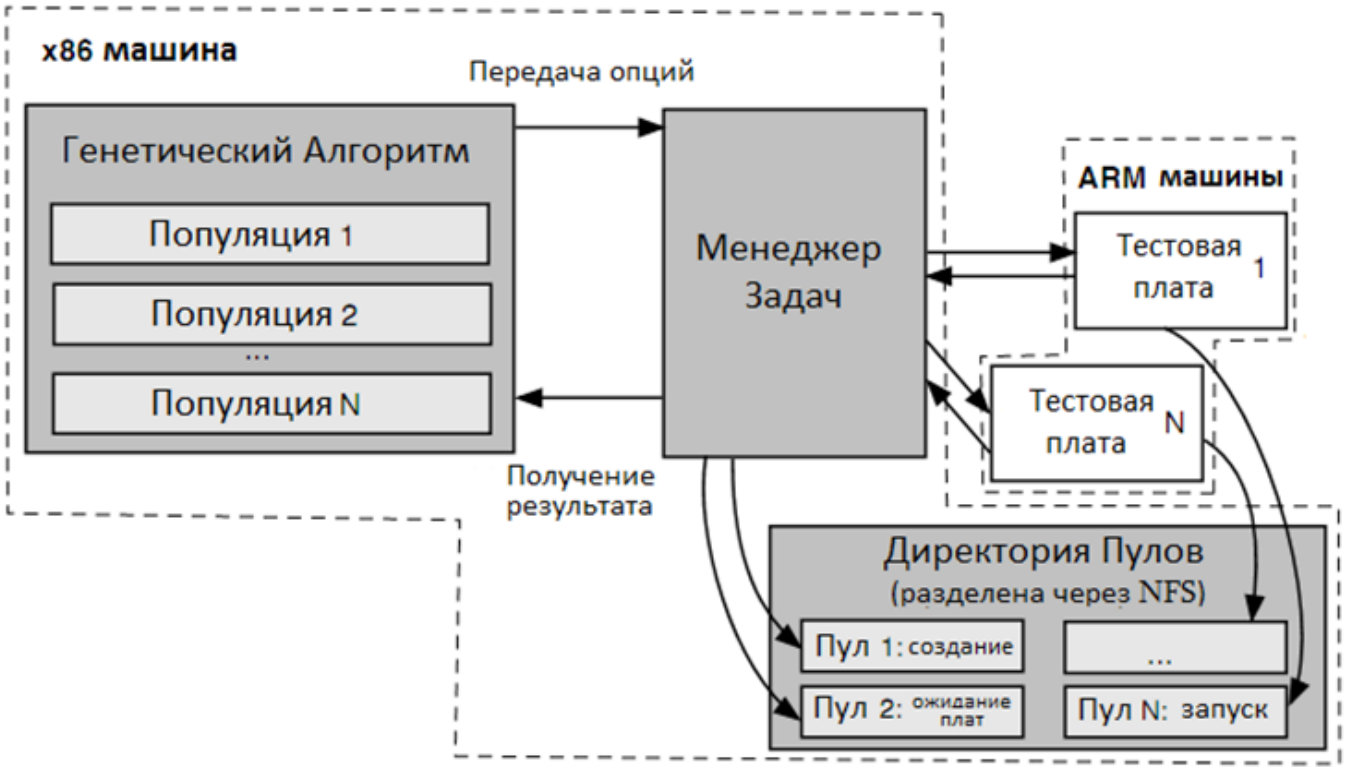


Рис. 9. Общая схема работы инструмента TACT.

Модуль эволюции генерирует с помощью генетического алгоритма различные наборы параметров компилятора. После кросс-компиляции, используя данные параметры на управляющей машине, исполняемый файл отправляется на выполнение на тестовые устройства, используя менеджер задач. Полученная на тестовых устройствах оценка производительности для заданного набора затем используется модулем эволюции для улучшения параметров компиляции в следующем поколении.

4.1.1. Единая структура для развертывания приложений

Инструмент использует определенную структуру каталогов для хранения оригинальных источников приложения, общих ресурсов (библиотек/инструментов, которые сами не являются предметом для оптимизации,

но необходимы целевому приложению для запуска), скриптов и каталогов для сборки, запуска приложения и хранения информации о его профиле (если включен запуск с профилированием). По своей структуре данная организация каталогов имеет аналогию с наследованием в объектно-ориентированных языках программирования, позволяя выносить общие действия и настройки на верхние уровни. Для того, чтобы добавить новые приложения для настройки, пользователь должен скопировать структуру каталогов из шаблона, и настроить файлы конфигурации и сценариев для конкретного приложения. Такая структура облегчает добавление новых тестов для настройки и позволяет реализовать часто выполняемые задачи в системе настройки.

4.1.2. Параллельная сборка и выполнение

Поддержка параллельной компиляции и выполнения значительно ускоряет процесс настройки. Параллелизм используется на двух уровнях. Во-первых, он позволяет компилировать приложения в то же время, как инструмент ждет результатов выполнения ранее скомпилированного приложения. Во-вторых, он позволяет использовать несколько тестовых плат в процессе настройки так, чтобы запускать целевые приложения одновременно на всех них. Чтобы организовать удаленное выполнение задач и синхронизацию между хостом и тестовыми платами, потребовалось реализовать простую систему управления очередью.

Ещё одной особенностью является то, что тестовые платы, используемые для настройки, не обязательно должны быть одной и той же модели, одного производителя или с одинаковой скоростью процессора. Параллелизм на уровне плат осуществляется на уровне популяций генетического алгоритма, и только результаты производительности, полученные на одной тестовой плате сравниваются друг с другом, так что эволюция происходит независимо в каждой популяции. Однако, при этом заложена возможность миграции между

популяциями, которые позволяют лучшим комбинациям опций GCC из каждой популяции распространять себя в другие популяции, таким образом, продолжая их конкуренцию с "местными видами" из этой популяции. Для оптимизации нагрузки количество популяций должно быть кратно числу используемых тестовых плат. В наших тестах мы провели настройку приложений на трёх различных тестовых платах (odroid, Panda board и xue, все с процессором архитектуры ARM v7) и результаты, полученные по итогам оптимизации, были корректны и воспроизводимы на любой из трёх плат. Для получения максимальной выгоды от параллельной сборки и выполнения, число каталогов для сборки (пулов) должно быть минимум в два раза больше количества тестовых плат, используемых для настройки, для некоторых приложений эффективное время компиляции может ещё больше улучшиться от увеличения числа пулов, в основном, это справедливо для больших приложений, у которых время сборки превышает время запуска тестов.

4.1.3. Поддержка компиляции с профилированием

Структура инструмента позволяет с его помощью настраивать приложения с учетом оптимизаций с профилированием, а не только со статическими оптимизациями. Оптимизации с профилированием включаются в GCC следующим образом: сначала приложение компилируется с опцией -fprofile-generate, затем при исполнении оно сохраняет информацию о профиле приложения (например, считает выполнения циклов, вероятности выполнения базовых блоков/путей потока управления и т.д.) в .gcda файлы, которые, как правило, находятся в каталоге для сборки приложения. После этого приложение перекомпилируется с опцией -fprofile-use, используя ранее собранную информацию о профиле. В автоматических инструментах оптимизация с профилем приводит к затратам двойного времени, необходимого для оценки того же количества особей. Задача системы управления очередью в инструменте -

позволить чередование выполнения двух этапов (сбора профиля и окончательной оценки) в различных каталогах сборки так, чтобы свести к минимуму время простоя тестовых плат, потому что обычно время запуска теста превышает время сборки приложения.

4.1.4. Быстрая перекомпиляция

Чтобы перекомпилировать приложение от запуска к запуску, используется скрипт rebuild-pool, который принимает два параметра через переменные среды - имя пула, который необходимо перекомпилировать, и набор опций GCC, используемых при компиляции. Этот скрипт может выглядеть очень просто: "make clean && make CFLAGS=”$NEW_CFLAGS” && make install", но, как правило, стоит изучить профиль выполнения приложения и определить файлы, содержащие функции, которые работают большую часть времени выполнения. Обычно количество таких файлов достаточно мало по сравнению с размером всего приложения. Например, для Webkit время компиляции всего приложения на четырехъядерном процессоре x86 составляет около 15 минут, в то время, как перекомпиляция только необходимых файлов занимает 3 минуты, обеспечивая практически те же результаты оптимизации. Кроме того, время компиляции может быть дополнительно снижено путем увеличения числа пулов и применением конвейерной компиляции на многоядерных CPU. Этот подход является предпочтительным, для его применения необходимо запускать make с опцией -jN, где N равно количеству ядер, однако не все приложения содержат достаточно возможностей для параллельной компиляции (например, SQLite состоит из одного .c исходного файла), но, как правило, для больших приложений это приводит к существенному выигрышу по времени.

4.1.5 Выбор опции и параметров для настройки

Для помощи в выборе опций и параметров настройки мы создали шаблонный конфигурационный файл формата XML, где они расположены по категориям, например, опции, управляющие инструкциями планировщика, опции и параметры для настройки встраивания функций или опции для распределения регистров. Пример такого файла приведен на рис. 10.

Рис. 10. Пример конфигирационного файла.

Данный файл может обновляться при помощи программного скрипта, который извлекает опции и параметры GCC для настройки непосредственно из исходных текстов компилятора, также нами был создан "черный список" опций GCC (например --help или отладочные опции) для фильтрации тех из них, которые не подходят для настройки. Используя этот скрипт, список опций может быть обновлен при переходе на новый компилятор GCC. Кроме того, в работе разработана программа для выполнения отсечения предположительно незначительных опций, но впоследствии убедились в неэффективности её первоначальной реализации. Для заданного целевого приложения, эта программа фильтровала множество опций путем сравнения исполняемых файлов для включенной/выключенной каждой опции, в то время как все другие опции одновременно включены или выключены. Если при изменении значения опции на противоположное не меняются созданные бинарные файлы, независимо от того, включены другие опции или нет, то она, скорее всего, не повлияет на генерацию кода, поэтому опция исключается из рекомендуемого набора для оптимизации. Позже мы обнаружили, что такой подход к фильтрации неэффективен: некоторые опции влияют только при включении одновременно с определенной комбинацией других опций, однако GCC не способен работать в случае, когда включены все опции одновременно. Эти два факта показали ненадёжность предварительной фильтрации опций для настройки. Однако подобная фильтрация в случае со значимыми опциями значительно облегчает анализ.

4.2. Анализ выбираемых конфигураций поиска

Как показывают результаты исследования, в зависимости от конкретного приложения и целевой архитектуры “лучшие” значения данных конфигураций могут отличаться. Поэтому задача по подбору “правильных” конфигураций не имеет однозначного ответа и зависит от выбранного приложения. Рассмотрим некоторые сравнительные результаты выбора конфигураций.

Как было сказано выше, в генетических алгоритмах при формировании нового поколения используются операторы мутации и скрещивания.

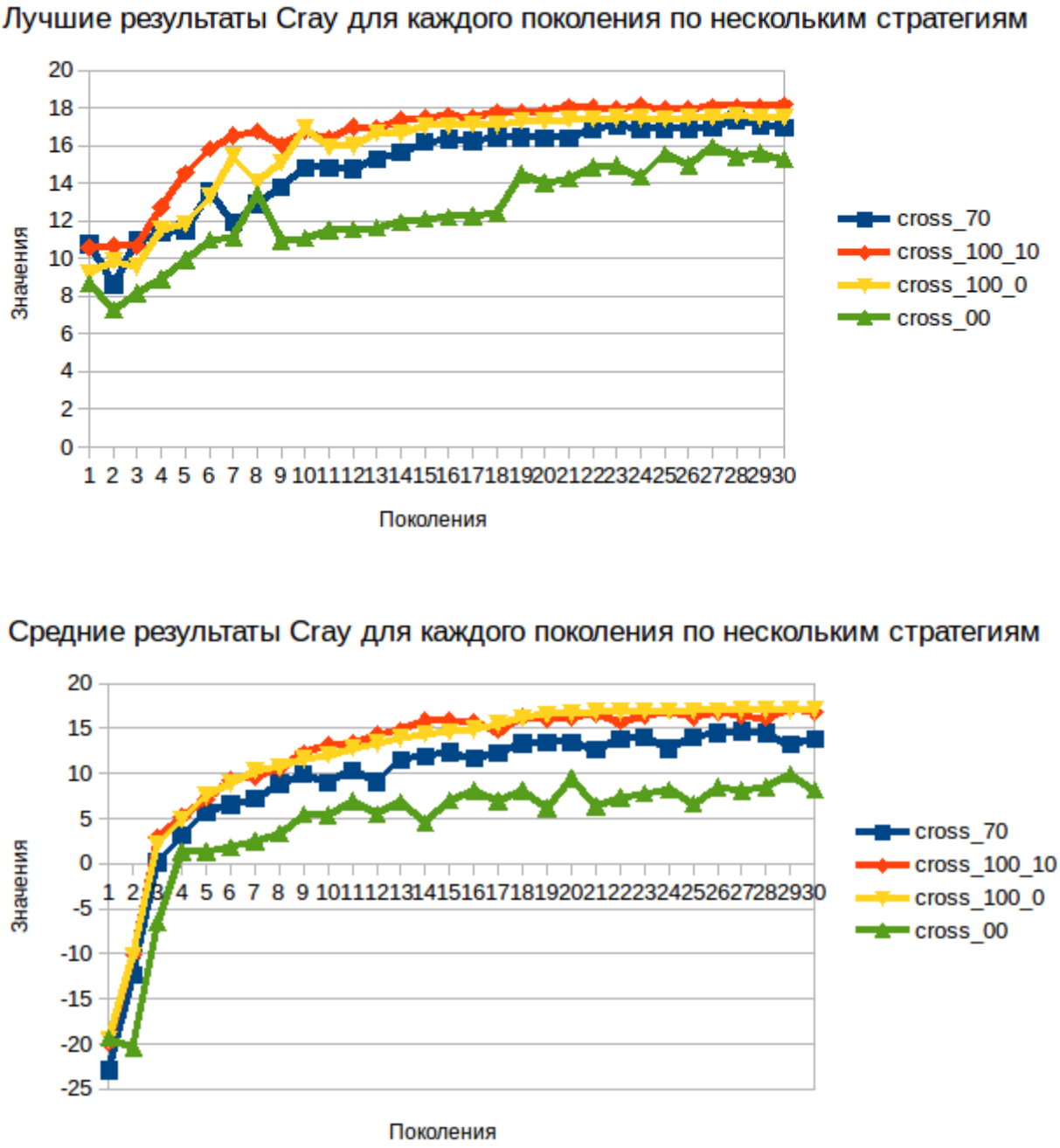


Рис. 11. Лучшие и средние результаты Cray для каждого поколения по нескольким стратегиям. 0 соответствует -O2.

На Рис. 11 продемонстрированы лучшие и средние результаты работы инструмента на приложении Cray по нескольким стратегиям (коэффициенты мутации и скрещивания). cross_70 соответствует случаю, когда скрещивание происходит в 70% случаев, мутация в 30%. cross_100 соответственно 100% и 0% а cross_00 0% и 100%. Как видно лучшие результаты получаются при 100% скрещивания.

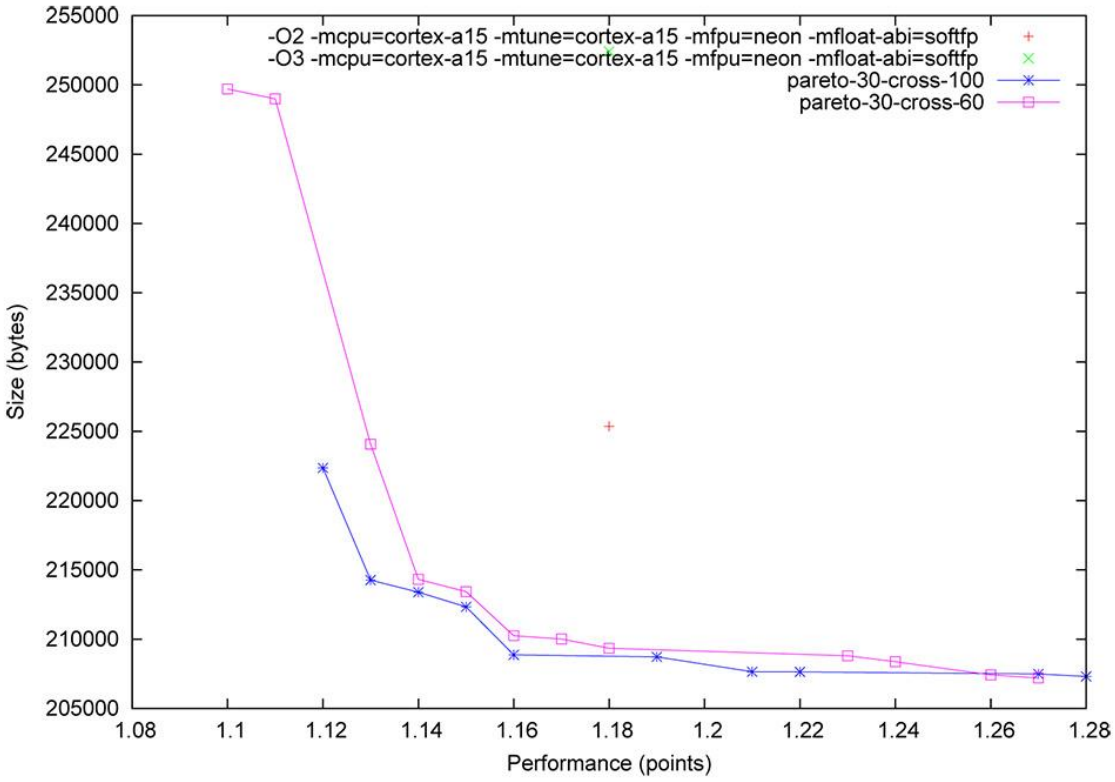


Рис. 12. Работа инструмента на приложении CxImage по двум критериям при коэффициенте мутации 100% и 60%.

Иногда можно добиться лучших результатов при более большом присутствии мутации. На рис. 12 продемонстрирован результат работы инструмента TACT на тестовое приложение CxImage, при коэффициенте мутации 100% и 60%. Хотя в среднем при стопроцентном скрещивании граница Парето лучше, лучший результат по производительности был получен при участии оператора мутации.

4.3. Сравнение с другими инструментами автоматического подбора опций компилятора

Был проведен сравнительный анализ между результатами разработанного нами инструмента TACT и некоторых развитых инструментов (рассмотренных в первой главе) автоматического подбора опций компилятора. Поскольку некоторые из этих инструментальных средств не имеют открытого исходного кода, было решено использовать результаты, взятые из статей описывающих их работу.

Для сравнения результатов OpenTuner и TACT, были взяты тестовые примеры (raytracer, matrix_multiply, tga_ga), на которых авторы инструмента проводили эксперименты. Был использован такой же компилятор gcc-4.9.0, и та же целевая архитектура x86_64. Результаты приведены на таблице 2. На всех рассмотренных тестовых примерах инструмент TACT показывает лучший результат.

В описании инструмента PEAK проводятся эксперименты на тестах из набора SPEC CPU2000 INT, используя компилятор gcc-4.3.3 и архитектуру x86. На таблице 3, приводятся сравнительные результаты инструментов TACT и PEAK на некоторых тестах из набора SPEC CPU2000 INT (с базой данных Train) для компилятора GCC и архитектуры x86.

Таблица 1. Результаты ускорения производительности с иснтрументами TACT и OpenTuner по сравнению с базовым набором –O2.

Название теста	Ускорение opentuner	Ускорение TACT %
Raytracer	~15%	~26%
matrix_multiply	~2.8 раза	~4,3 раза
tga_ga	~19%	~34%

Таблица 2. Результаты ускорения производительности с иснтрументами TACT и PEAK по сравнению с базовым набором –O2.

Название теста	Ускорение произв. PEAK	Ускорение произв. TACT
Crafty	7.5%	9.9%
bzip2	2%	1.8%
Eon	9%	21%
Gap	3%	7%
Geomean	4.2%	10.2%

4.4. Результаты TACT

Инструмент TACT был протестирован на множестве приложений с открытым исходным кодом (x264, zlib, C-Ray, libevas, SQLite, Crafty Chess и т.д.) на платформах ARM и x86. Инструмент тестировался как для компиляторной инфраструктуры GCC, так и для LLVM, причем он применялся как для поиска наборов оптимизаций по производительности, так и для производительности и размера бинарного кода (граница Парето). В итоге для компилятора GCC (gcc-4.9) и архитектуры ARM (процессор ARM Cortex-A9) инструментом TACT получено улучшение производительности от 4 до 30% (для libevas 30%, 24% для x264, 9% для SQLite и т.д.) по сравнению с результатом полученным базовым набором опций (-O2). Используя LLVM (llvm-2.4) в качестве компилятора на платформе ARM получаем соответственно от 2 до 15% (для x264 15%, 7,5% для SQLite, 6% для zlib, 2% для C-Ray и т.д.).

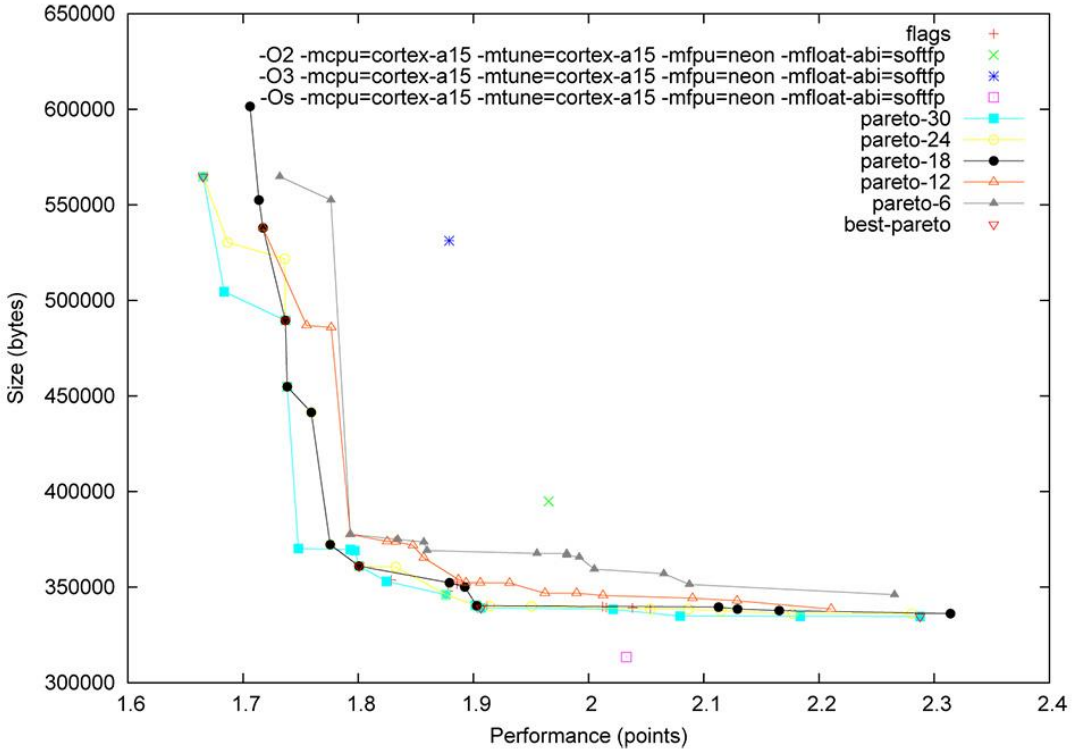


Рис. 13. Результат работы инструмента TACT на приложении CxImage.

Инструмент TACT был также протестирован для поиска эффективных оптимизаций по двум критериям (производительность и размер бинарного кода). Результаты получены в виде границ Парето, Рассмотрим результаты работы инструмента на примере приложения SQLite. На рис. 13 продемонстрировано улучшение границы Парето от поколения к поколению (6-ое, 12-ое, 18-ое, 24-ое и 30-ое поколения), также результаты при базовых уровнях оптимизаций компилятора. В работе использовались компилятор GCC (gcc-4.9.2) и архитектура ARMv7. В зависимости от требований пользователя можно выбрать результаты как оптимальные по производительности (левую верхнюю точку) так и по размеру кода (правую нижнюю точку), можно также выбрать набор оптимизаций, при

котором такой же размере бинарного кода обеспечивает ускорение производительности на 12% по сравнению с -O2.

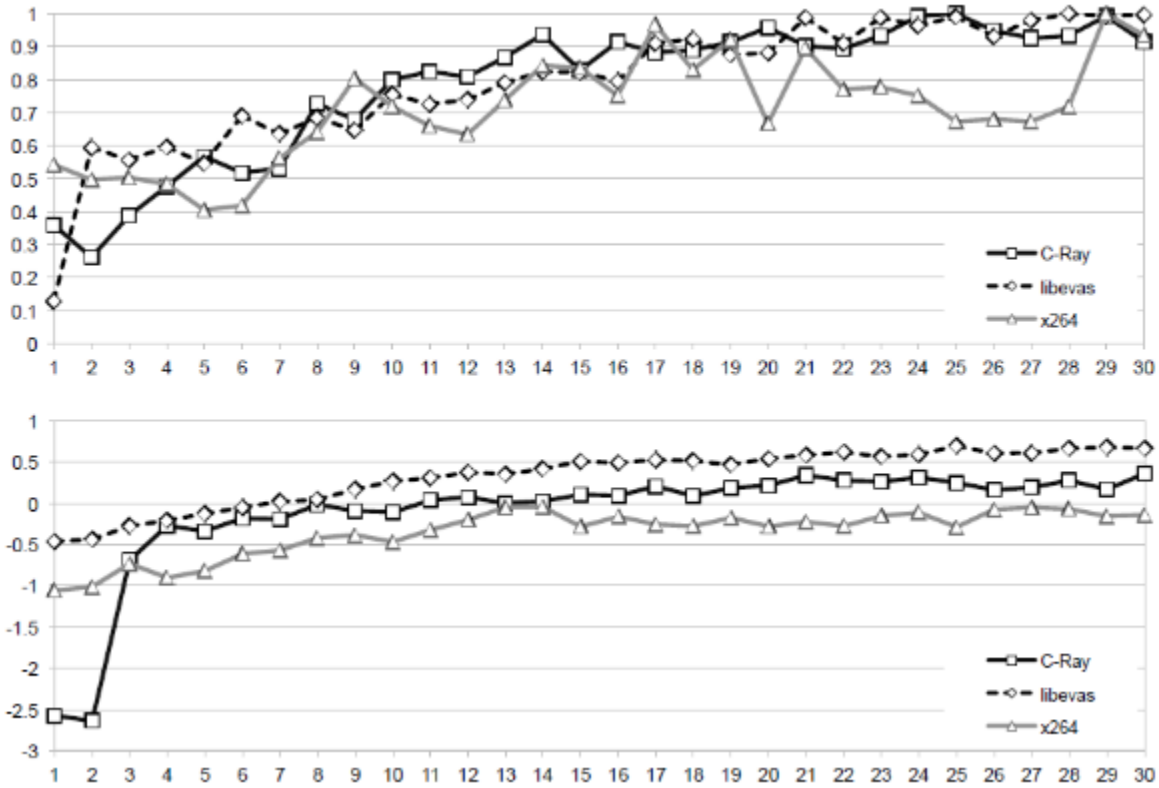


Рис. 14. Лучший (верхнее) и средний (нижнее) результаты улучшения производительности для каждого поколения, 0 соответствует -O2, 1 - максимальной производительности, достигнутой инструментом.

Количество запусков для получения существенного улучшения сгенерированного кода, зависит как от самого приложения и выбранного компилятора, так и от выбранных оптимизаций компилятора и конфигураций настройки (количество поколений и популяций, размер конфигураций в одной популяции, коэффициент мутации и т.д.).

Графики на рис. 14 показывают изменения максимального и среднего улучшения производительности для каждого поколения. На вертикальной оси продемонстрирована текущая производительность по сравнению с базовым уровнем оптимизации (-O2), учитывая, что "1" соответствует максимальной

производительности на последнем поколении. В этих графиках не учтены лучшие наборы оптимизаций, которые были найдены на предыдущих поколениях и хранились в архивах, а также те наборы, на которых произошли ошибки компиляции или запуска.

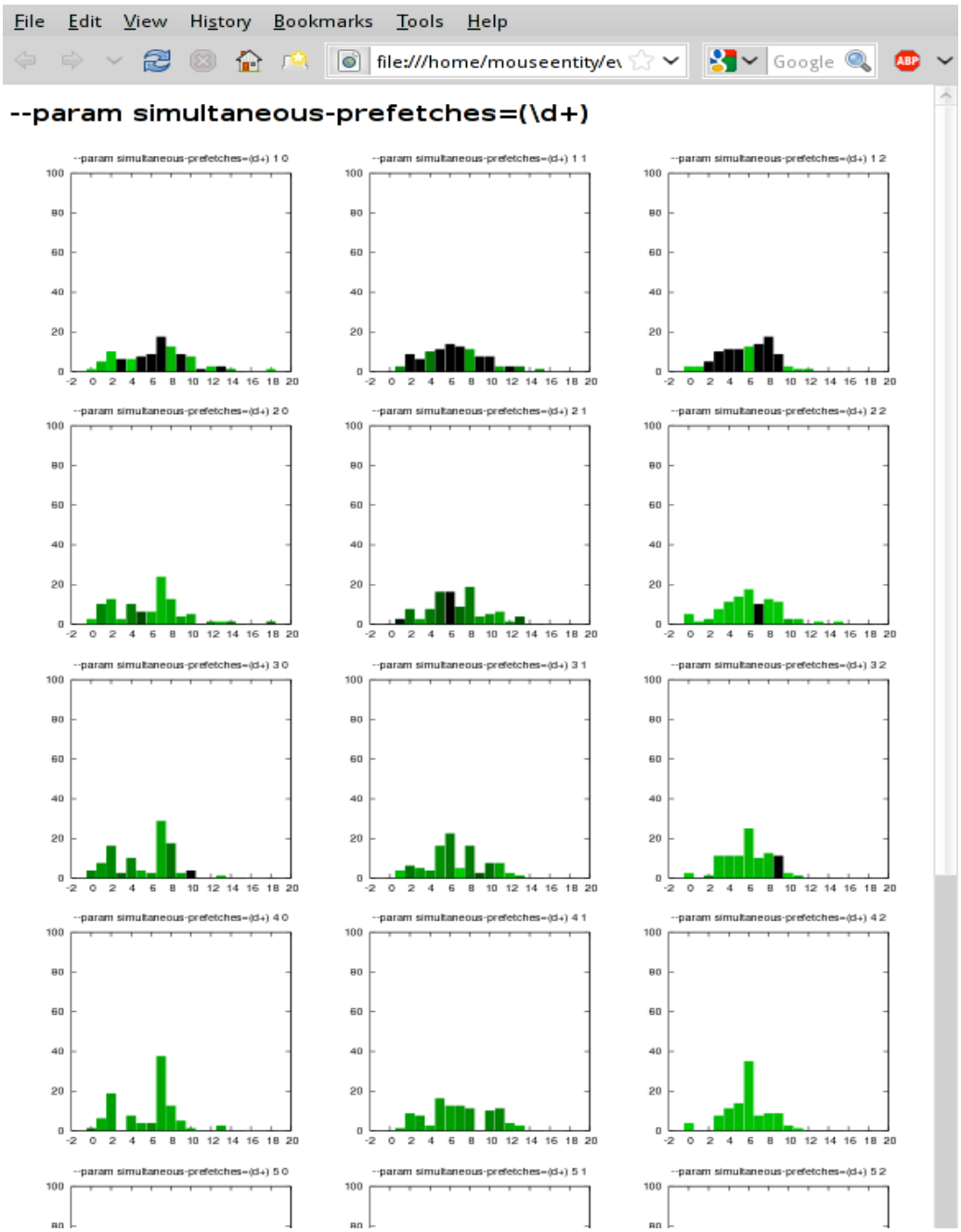


Рис. 15. Инструмент для анализа и визуализации процесса эволюции.

В рамках TACT был также реализован инструмент, позволяющий анализировать процесс эволюции. Результат работы инструмента показан на рис. 15 который демонстрирует эволюцию параметра количества одновременных предзагрузок при настройке параметров предзагрузки в циклах в GCC (--param simultaneous-prefetches). HTML страницы содержат таблицу из картинок, показывающих распределение значений параметров в каждом поколении (строки) и популяции (столбцы). По высоте откладывается процент строк популяции, в которых встречается данный параметр с определённым значением, цвет показывает средние показатели производительности для каждого параметра. Например, на рисунке 1 видно, как значение параметра 6 в третьей популяции растёт с 20% до 40% за первые 4 поколения. Наблюдение за процессом эволюции в такой визуальной форме помогает разработчику выбрать значения параметров, которые обеспечивают последовательное улучшение производительности.

В рамках набора инструментов TACT был также реализован инструмент на основе метода автоматической проверки и корректировки исходного кода библиотеки, для улучшения работы оптимизаций на этапе компоновки. Инструмент был протестирован на множестве тестовых приложений, таких как SQLite, LibEvas, CLucene, Lame, и т.д.

В таблицах 3 и 4 приводятся результаты работы инструмента на некоторых популярных библиотеках, демонстрирующие соответственно улучшение размера полученного бинарного кода и производительности. Второй, третий и четвертые столбцы демонстрируют соответственно результаты с базовым уровнем -O2, с включением –flto и итоговому результату после работы инструмента.

Таблица 3. Размер бинарного кода в байтах при применении инструмента.

Библиотека	-O2	-flto	-flto -fvisibility	улучшение в %
Clucen e	733924	615475	**567807**	**22.63%**
SQLite	343938	344010	**300676**	**12.57%**
Evas	27159	27114	**25036**	**7.81%**
Lame	197375	197325	**184604**	**6.47%**
x264	438867	438950	**427106**	**2.7%**

Таблица 4. Производительность в секундах при применении инструмента.

Библиотека	-O2	-flto	-flto -fvisibility	улучшение в %
SQLite	8.1	8.2	6.86	15.3%
Lame	10.17	10.17	10.17	0%
x264	30.14	30.14	30.08	0%

4.5. Сокращение итоговой строки опций компилятора

Набор опций, полученных инструментом, может быть избыточным: некоторые опции могут ничего не делать, но они будут по-прежнему присутствовать в результирующей строке опций. Мы написали программу, которая пытается уменьшить количество опций для лучших особей за счет удаления тех опций, присутствие которых не влияет на итоговые бинарные файлы. Она пытается достичь этого путем последовательного удаления опций по одной, однако, это может занять достаточно длительное время (около суток).

Например, для получения итогового результата по одному из проектов мы запускали настройку для выбора около 200 опций и параметров GCC, и все они присутствовали в строке результата, что значительно усложняет анализ. После использования инструмента для фильтрации мы получили 39 значимых опций и параметров для Evas, 66 для CLucene и 44 для SQLite. Также, существует потенциальная возможность ещё более сократить полученную строку опций за счёт исключения из неё опций, которые мало влияют на производительность. Такие опции, очевидно, могут попадать в результат в процессе эволюции. Однако данный метод требует множества запусков целевого приложения на тестовых платах для определения точного вклада каждой из опций в итоговую производительность, а также, в некоторых случаях, ручной оценки существенности данного вклада.

4.6. Инструменты анализа результатов

Для облегчения анализа результатов при автоматической настройке мы разработали два инструмента - анализатор текстовых логов и графический инструмент для проверки процесса эволюции. Оба инструмента используют логи запуска (в виде простого текста и XML), содержащие строки опций для компиляции, их оценки, номер текущего поколения и количество популяций (как правило, количество популяций соответствует количеству тестовых плат, на которых приложение было запущено).

Первый инструмент разбирает лог запуска и строит таблицу с опциями и параметрами GCC, упорядоченными по их полезности, которая рассчитывается как средняя производительность запусков, при компиляции которых использовалась данная опция или параметр. Кроме того, он показывает, сколько раз встречается опция и её среднее положение в логе запуска. Первое значение показывает, насколько часто встречается данная опция у особей, а второе

показывает, насколько "зрелой" является каждая опция. Большее среднее значение позиции означает, что опция имеет тенденцию чаще появляться в конце эволюционного процесса. Чем больше эти два значения для опции, тем больше вероятность того, что она принесёт выгоду при её использовании. Так как различные популяции, как правило, обрабатываются на различных тестовых платах, которые могут иметь различную производительность, то результаты производительности сопоставимы только в пределах одной популяции. Поэтому, прежде чем анализировать результаты оптимизации из подобного лога запуска, необходимо при помощи вспомогательного скрипта разделить их на отдельные логи, содержащие результаты отдельно для каждой тестовой платы. Кроме того, этот инструмент помогает диагностировать те опции или параметры, которые вызывают ошибки при компиляции, либо приводят к некорректному результату. Так как инструмент присваивает некорректно работающей опции с очень большим временем выполнения (один миллиард секунд), то такие опции могут быть легко идентифицированы по их низкому положению в таблице результатов, отсортированной по среднему времени выполнения. Тем не менее, такие опции (или их комбинации), не сильно влияют на процесс оптимизации, так как они в любом случае вымирают в процессе эволюции.

Первый инструмент, в основном направлен на изучение булевых опций (которые могут иметь только два состояния - включена либо выключена), а второй инструмент предназначен в основном для проверки эволюции числовых параметров. По логу запуска он генерирует некоторое количество HTML файлов и картинок, чтобы разработчик мог увидеть, как происходит процесс эволюции из поколения в поколение и как распределение значений параметров варьируется между поколениями и популяциями для каждого настраиваемого параметра.

4.7. Выводы

1. На основе алгоритма и метода, описанных в главах 2 и 3, были разработаны инструментальные средства, которые были интегрированы в набор инструментов TACT, разрабатываемый в ИСП РАН, для автоматического улучшения применимости оптимизаций компилятора для конкретного приложения и целевой платформы.
2. Описана общая структура набора инструментов TACT, перечислены основные особенности реализации, такие как:

· поддержка кросс-платформенности,
· единая структура для развертывания приложений,
· параллельная сборка и выполнение на целевой машине,
· быстрая перекомпиляция,
· поддержка компиляции с профилированием,
· удобный для пользователя конфигурационный файл.

3. Проанализированы и оценены влияния числовых значений коэффициентов генетического алгоритма (такие как мутация, скрещивание) на получаемые результаты.
4. Сделан сравнительный анализ с некоторыми существующими инструментами автоматического подбора оптимизаций компилятора. Полученные результаты свидетельствуют об эффективности разработанного алгоритма.
5. Инструмент также был апробирован на множестве тестовых приложений, в том числе, на SPEC как по одному критерию (производительность или размер кода), так и по двум критериям (компромиссы на границе Парето).
6. Инструмент также был протестирован для автоматического улучшения применимости оптимизаций на этапе компоновки, где было достигнуто улучшение производительности и уменьшения размера кода до 20%.

Заключение

В диссертационной работе рассмотрен актуалный в настоящее время вопрос оптимизации компиляторов. Исследоваы и анализированы имеющиеся достижения и наиболее значимые результаты. Разработаны новые подходы решения, используя и модифицируя разные методы и алгоритмы, которые нашли отражение в созданном в институте системного программирования РАН наборе инструментов TACT, который показал свою эффективность при рещении многих задач и продолжает развываться. Инструмент, разработанный на основе метода автоматического подбора эффективных оптимизаций компилятора по нескольким критериям на основе Парето доминирования является состовной частью инструментального комплекса TACT и применяется в реализации разных проектов, в том числе, в научно-исследовательском проекте корпорации Samsung.

Работа актуальна и имеет возможность дальнейшего соверштвевования, так как областть быстро развивается и остановиться на достигнутом нецелесобразно.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ ДИССЕРТАЦИОННОЙ РАБОТЫ

- Разработан метод автоматического подбора эффективных оптимизаций компилятора для конкретного приложения на заданной платформе. Данный метод поддерживает также автоматический подбор эффективных наборов оптимизаций на основе Парето-доминирования. Исследованы и оценены влияния конфигураций генетического алгоритма на процесс эволюции[1,3,5].
- Разработан метод автоматической проверки и корректировки исходного кода библиотек программ, обеспечивающий улучшение эффективности выполнения оптимизаций на этапе компоновки.

- Разработанные методы были реализованы в рамках инструментальной инфраструктуры TACT учитывая особенности компиляторов GCC и LLVM и применяемости концепций генетического алгоритма[1,3,4,5].

Список рисунков

Список таблиц

Список используемой литературы

1. Мамикон Варданян. Метод автоматического подбора наборов эффективных оптимизаций компилятора по нескольким критериям на основе Парето-доминирования. Вестник Инженерной академии Армении (ВИАА). 2015.Т.12, №2
2. Роман Жуйков, Дмитрий Плотников, Мамикон Варданян. Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM. Труды Института системного программирования РАН. 2012. Т. 22. С. 49-66
3. Dmitry Plotnikov, Dmitry Melnik, MamikonVardanyan, Ruben Buchatskiy. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. International Conference on Computational Science, ICCS 2013.
4. Дмитрий Мельник, Шамиль Курмангалеев, Арутюн Аветисян, Андрей Белеванцев, Дмитрий Плотников, Мамикон Варданян. Оптимизация приложений для заданных статических компиляторов и целевых архитектур: методы и инструменты. Труды Института системного программирования РАН. 2012,том26,с. 343-356.
5. Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov. An Automatic Tool for Tuning Compiler Optimizations. 2013 Computer Science and Information Technologies (CSIT)
6. R. M. Stallman *GNU Compiler Collection Internals*, 2003 :Free Software Foundation, Inc.

7. R. M. Stallman *Using the GNU Compiler Collection*, 2003 :Free Software Foundation, Inc.

8. GNU Compiler Collection (GCC) Internals [HTML]http://gcc.gnu.org/onlinedocs/gccint/

9. C. Yang, C. Li, F. Wang "Performance Improvements for GCC Using Architecture Features on IA-64", Proceedings of the GCC Developers Summit 2005, pp. 199-208.

10. D. Nuzman, A. Zaks. "Autovectorization in GCC - two years later", Proceedings of the GCC Developers Summit 2006, pp. 145-158

11. J. Merrill. Generic and Gimple: A New Tree Representation for Entire Functions. In the GCC Developer's summit, pages 171--180, June 2003.

12. S. Pop, R. Yazdani, Q. Neill "Improving GCC's auto-vectorization with if-conversion and loop flattening", Proceedings of the GCC Developers Summit 2010, pp. 89-96.

13. Ian Lance Taylor (2008). "A New ELF Linker". GCC Developers' Summit 2008. pp. 129–136. Retrieved 2013-03-06.

14. D. Novillo. Tree SSA - a New Optimization Infrastructure for GCC. In Proc. of the GCC Developers Summit, pages 181-194, June 2003.

15. C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See http://llvm.cs.uiuc.edu.

16. Jianzhou Zhao , Santosh Nagarakatte , Milo M.K. Martin , Steve Zdancewic, Formalizing the LLVM intermediate representation for verified program transformations, Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 25-27, 2012, Philadelphia, PA, USA

17. C. Lattner. LLVM and Clang: Next generation compiler technology. In BSDCan 2008: The BSD Conference, Ottawa, Canada, May 2008.

18. Ш.Ф. Курмангалеев. Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования. Труды ИСП РАН, том 24, стр. 127-144, 2013 г. DOI: 10.15514/ISPRAS-2013-24-7.

19. Аветисян А.И. Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения // Труды ИСП РАН – 2012. – Т. 22. – С. 11–18

20. D. Jaggar et al., "ARM Architecture and Systems," *IEEE Micro*, Vol. 17, No. 4, Jul.-Aug. 1997, pp. 9-11.

21. A. N. Sloss, D. Symes, C. Wright, “Arm System Developer's Guide: Designing and Optimizing System Software”, (2004)

22. W. Stallings, “Computer Organization and Architecture: Designing for Performance” 8th Edition, (2011)

23. About Cotex-A8 Processor [HTML] http://www.arm.com/products/processors/cortex-a/cortex-a8.php

24. Веб-сайт ARM. Технология NEON. http://www.arm.com/products/processors/technologies/neon.php

25. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques, and Tools”, 2nd Edition (2006)

26. Веб-сайт Enlightenment Foundation Libraries. http://www.enlightenment.org/p.php?p=about/ef

27. Веб-сайт SQLite.

http://www.sqlite.org/about.html

28.Веб-сайт WebKit.

http://www.webkit.org

29.Веб-сайт SunSpider

https://webkit.org/perf/sunspider/sunspider.html

30. Z. Pan, R. Eignmann, “Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning.”, Proceedings of the International Symposium on Code Generation and Optimization, 2006.

31.S. Triantafyllis, M.J. Bridges, E. Raman, G. Ottoni and D. August, “A Framework for Unrestricted WholeProgram Optimization.”, Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006.

32.Z. Pan, R. Eignmann, “Fast, Automatic, ProcedureLevel Performance Tuning.”, Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques, 2006.

33.H. Feltl, “Ein Genetischer Algorithmus fuer das Generalized Assignment Problem”, Diplomarbeit, 2003.

34.M. Haneda, P. Knijnenburg, H. Wijshoff “Automatic Selection of Compiler Options Using Non-Parametristic Inferential Statistics.”, Proceedings of the 14th International Conference on Parallel Architecture and Compilation Techniques, 2005

35.G. G. Fursin, M. F. P. O’Boyle, P. M. W. Knijnenburg, Evaluating iterative compilation, in: Proceedings of the 15th international conference on Languages and Compilers for Parallel Computing, LCPC’02, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 362–376.

36. G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, M. OBoyle, Milepost gcc: Machine learning enabled self-tuning compiler, International Journal of Parallel Programming 39 (2011) 296–327.

37. G. Fursin, O. Temam, Collective optimization: A practical collaborative approach, ACM Transactions on Architecture and Code Optimization 7 (2010) 20:1–20:29.

38. Веб-сайт ACOVEA. http://www.coyotegulch.com/products/acovea/

39. K. Hoste, L. Eeckhout, Cole: compiler optimization level exploration, in: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, CGO ’08, ACM, New York, NY, USA, 2008, pp. 165–174.

40. Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly “OpenTuner: An Extensible Framework for Program Autotuning” International Conference on Parallel Architectures and Compilation Techniques. Edmonton, Canada. August, 2014. Slides. Bibtex

41. Z. Pan, R. Eigenmann “PEAK—A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration” in Code Generation and Optimization, 2006. CGO 2006

42. D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. "PERI Auto-Tuning." *Journal of Physics: Conference Series*, 125(1):012089, 2008.

43. SPEC CPU2000 Benchmark http://www.spec.org/cpu2000/

44. Климова О.Н., Ногин В.Д. Учет взаимно зависимой информации об относительной важности критериев в процессе принятия решений// Журнал вычислительной математики и математической физики, 2006, т. 46, № 7, С. 2179-2191.
45. Меньшикова О.Р., Подиновский В.В. Построение отношения предпочтения и ядра в многокритериальных задачах с упорядоченными по важности неоднородными критериями// ЖВМиМФ, 1988, 28(5), 647-659.
46. Подиновский В.В. Введение в теорию важности критериев в многокритериальных задачах принятия решений. М.: Физматлит, 2007. 64 с. 81. Подиновский В.В., Гаврилов В.М. Оптимизация по последовательно применяемым критериям. М., «Сов. радио», 1975, 192 с. 82.
47. Подиновский В.В., Ногин В.Д. Парето-оптимальные решения многокритериальных задач. -- М.: "Наука", 1982. -- 254 с.
48. Подиновский В.В., Ногин В.Д. Парето-оптимальные решения многокритериальных задач. – М.: Наука. Главная редакция физикоматематической литературы, 1982. – 256 с.
49. Романова И.К. Программный комплекс «Многокритериальная оптимизация систем управления»: свидетельство о государственной регистрации программы для ЭВМ № 2012610400 РФ. 2012г.)
50. А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, Том 16, 2009, стр. 31-50
51. D. Melnik, A. Belevantsev, D. Plotnikov, S. Lee, A case study: optimizing gcc on arm for performance of libevas rasterization library, in: Proceedings of International Workshop on GCC Research Opportunities (GROW-2010), Pisa, Italy, 2010. URL http://ctuning.org/dissemination/grow10-03.pdf

52.F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. O'Boyle, E. Rohou, Iterative compilation in a non-linear optimisation space (1998).

53.J. Cavazos, M. F. P. O'Boyle, Automatic tuning of inlining heuristics, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 14–

54.S. V. Gheorghita, H. Corporaal, T. Basten, Iterative compilation for energy reduction, J. Embedded Comput. 1 (4) (2005) 509–520. URL http://dl.acm.org/citation.cfm?id=1233791.1233798

55. E. Park, S. Kulkarni, J. Cavazos, An evaluation of different modeling techniques for iterative compilation, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, ACM, New York, NY, USA, 2011, pp. 65–74.

56. G. Bashkansky, Y. Yaari, Black box approach for selecting optimization options using budget-limited genetic algorithms, SMART'07 (2007) pp. 1–16.

57.G. Pekhimenko, A. D. Brown, Efficient program compilation through machine learning techniques, in: Proceedings of the The Fourth International Workshop on Automatic Performance Tuning (iWAPT), Tokyo, Japan, 2009.

58.Escart Zitzler, Lothar Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach". IEEE Transactions on Evolutionary computation, vol. 3, no. 4, 1999.

59.Alba E., Troya J.M. A Survey of Parallel Distributed Genetic Algorithms// Complexity. – 1999. – Vol.4, № 4. – P.31-52.

60. Alba E., Troya J.M. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms// Future Generation Computer Systems. – 2001. – Vol.17, №4. – P.451-465.

61. Alba E., Troya J.M. Influence of the Migration Policy in Parallel Distributed GAs with Structured and Panmictic Populations// Application Intelligence. – 2000. – Vol.12, №3. – P.163-181.

62. E. Zitzler, M. Laumanns, L. Thiele, Spea2: Improving the strength pareto evolutionary algorithm

63. Bleuer S., Brack M., Thiele L., Zitzler E. Multiobjective genetic programming: Reducing bloat by using SPEA 2 // Proceedings of the 2001 Congress on Evolutionary Computation (CES-2001). Seoul, Korea, May 27-30, 2001. P. 536-543.

64. Coello Coello C.A. A comprehensive survey of evolutionary-based multiobjective optimization techniques.

65. Coello Coello Carlos A. An empirical study of evolutionary techniques for multiobjective optimization in engineering design. PhD thesis. Department of computer science, Tulane university. New Orleans, LA, apr 1996.

66. Deb K. Evolutionary Algorithm for Multi-Criterion Optimization in Engineering Design // Proceedings of Evolutionary Algorithms in Engineering and Computer Science (EUROGEN-99) -- pp. 135-161.

67. Deb K. Multi-objective Genetic Algorithms: Problem Difficulties and Construction of Test Problems. // Evolutionary Computation -- vol.7, 1999. -- pp. 205-230.

68. Gordon V.S., Whitley D., Bohn A. Dataflow parallelism in genetic algorithms // Parallel Problem Solving from Nature, Amsterdam: Elsevier Science. 1992. No.2. P. 533-542.

69. Grosso P.B. Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model// Unpublished doctoral

dissertation, The University of Michigan. (University Microfilms №8520908), 1985.

70. Gumennikova, A.V. Comparative analysis of multiobjective optimization methods by genetic algorithms / A.V. Gumennikova // Красноярский край: освоение, развитие, перспективы: Тез. Докл. Регион. Студ. Науч. Конф. Ч. 1/ Краснояр. Гос. Аграр. Ун-т; Сост. О.Н. Чепалова. – Красноярск, 2003, с. 33-34.

71. **Schaffer J. D.** Multiple Objective Optimization with Vector Evaluated Genetic Algorithms [Текст] / J. D. Schaffer // PhD thesis, Vanderbilt University, Nashville, Tennessee, 1984.

72. **Schaffer J. D.** Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. In Genetic Algorithms their Applications [Текст] / J. D. Schaffer // Proceedings of the First International Conference on Genetic Algorithms, pages 93 - 100, Hillsdale, New Jersey, 1985. Lawrence Erlbaum.

73. **Schaffer J. D.** Multiobjective Learning via Genetic Algorithms [Текст] / J. D. Schaffer J. J. Grefenstette // In Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85), pages 593 - 595, Los Angeles, California, 1985. AAAI.

74. **Norris S. R.** Pareto-Optimal Controller Gains Generated by a Genetic Algorithm [Текст] / S. R. Norris W. A. Crossley // In AIAA 36th Aerospace Sciences Meeting Exhibit, Reno, Nevada, January 1998. AIAA Paper 98 - 0010.

75. **Crossley W. A.** Using the Two-Branch Tournament Genetic Algorithm for Multiobjective Design [Текст] / W.A. Crossley, A.M. Cook, D.W. Fanjoy, V.B. Venkayya // In AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, Materials Conference, Long Beach, California, April 1998. AIAA Paper 98 - 1914.

76. **Fonseca** C.M. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion Generalization [Текст] / C. M. Fonseca P. J. Fleming // In S. Forrest, editor, Proceedings of the Fifth International Conference on Genetic Algorithms, pages 416 - 423, San Mateo, California, 1993. University of Illinois at Urbana - Champaign, Morgan Kaufmann Publishers.
77. **Fonseca C. M.** Multiobjective Optimization Multiple Constraint Handling with Evolutionary Algorithms - Part I [Текст] / C. M. Fonseca P. J. Fleming // A Unified Formulation. IEEE Transactions on Systems, Man, Cybernetics, Part A: Systems Humans, 28(1):26 - 37, 1998.
78. **Fonseca C. M.** On the Performance Assessment Comparison of Stochastic Multiobjective Optimizers [Текст] / C. M. Fonseca P. J. Fleming // In H.-M. Voigt, W. Ebeling, I. Rechenberg, H.-P. Schwefel, editors, Parallel Problem Solving from Nature - PPSN IV, pages 584 — 593. Springer-Verlag. Lecture Notes in Computer Science No. 1141, Berlin, Germany, September 1996.
79. **Horn J.** Multiobjective Optimization using the Niched Pareto Genetic Algorithm [Текст] / J. Horn N. Nafpliotis // Technical Report IIliGAl Report 93005, University of Illinois at Urbana - Champaign, Urbana, Illinois, USA, 1993.
80. **Horn J.** A Niched Pareto Genetic Algorithm for Multiobjective Optimization [Текст] / J. Horn, N. Nafpliotis, D. E. Goldberg // In Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, volume 1, pages 82 - 87, Piscataway, New Jersey, June 1994. IEEE Service Center.
81. **Erickson M.** The Niched Pareto Genetic Algorithm 2 Applied to the Design of Groundwater Remediation Systems [Текст] / M. Erickson, A. Mayer, J. Horn // In E. Zitzler, K. Deb, L. Thiele, C. A. Coello Coello, D. Corne, editors, First International Conference on Evolutionary Multi-Criterion Optimization, pages 681 - 695. Springer-Verlag. Lecture Notes in Computer Science No. 1993, 2001.

82. Zitzler E., Thiele L. Multiobjective optimization using evolutionary algorithms -- A comparative case study. // Parallel Problem Solving from Nature -- Springer, Berlin, Germany. -- pp. 292-301.

83. Bradley, P. S., Bennett, K. P., & Demiriz, A. (2000). Constrained k-means clustering (Technical Report MSR-TR-2000-65). Microsoft Research, Redmond, WA.

84. U. K. Chakraborty , K. Deb and M. Chakraborty "An analysis of selection algorithms: A Markov chain approach", *Evolutionary Computation*, vol. 4, no. 2, pp.133 -167 1996

85. S. Pop, R. Yazdani, Q. Neill "Improving GCC's auto-vectorization with if-conversion and loop flattening", Proceedings of the GCC Developers Summit 2010, pp. 89-96.

86. C. Yang, C. Li, F. Wang "Performance Improvements for GCC Using Architecture Features on IA-64", Proceedings of the GCC Developers Summit 2005, pp. 199-208.

87. Zitzler E., Thiele L. Multiobjective optimization using evolutionary algorithms -- A comparative case study. // Parallel Problem Solving from Nature -- Springer, Berlin, Germany. -- pp. 292-301.

88. K. Nordkvist, "Solving TSP with a genetic algorithm in C++," https://tinyurl.com/lq3uqlh, 2012.

89. X. Fan, "Optimize your code: Matrix multiplication," https://tinyurl.com/kuvzbp9, 2009.

90. S. Pixel, "3D Basic Lessons: Writing a simple raytracer," https://tinyurl.com/lp8ncnw, 2012.

91.J. Hubicka, “Interprocedural optimization framework in GCC”, Proceedings of the 2007 GCC Developers’ Summit.

92.Glek, T., Hubicka, J.: Optimizing real-world applications with gcc link time optimization. In: GCC Developers Summit (2010)