Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Айнур Ринадович Ибатов

Выпускная квалификационная работа

Разработка программной системы на основе LLVM для оптимизации задержки в высоконагруженных секциях кода C++-приложений

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель:
к.ф.-м.н. Д. С. Шалымов

Рецензент:
декан института прикладных компьютерных наук ИТМО
А. М. Кузнецов

Санкт-Петербург
2023 г.

Содержание

Введение

В современном мире существует огромное количество программного обеспечения, написанного на C++. Считается, что данный язык программирования является очень быстрым. Тем не менее существуют сферы, требующие от программы высокой производительности, которая недостижима лишь удачным выбором языка.

Написать код с минимальной асимптотической сложностью – не значит написать оптимальный с точки зрения задержки код: нередко решающим фактором является компилятор, а именно его оптимизации, производящиеся над программой.

В сферах, нуждающихся в высокопроизводительном коде, есть потребность в применении дополнительных оптимизаций. С этой задачей обычно достаточно хорошо справляется компилятор, предоставляющий, например, флаги компиляции [1]. При включенных оптимизационных флагах компиляции часто достигается удовлетворяющий пользователя результат, однако далеко не всегда этого достаточно.

Одной из самых важных оптимизаций компилятора является встраивание функций [2] – замена вызова функции непосредственно телом функции. Данная работа направлена на создание программной системы, которая предоставляет пользователям возможность оптимизировать задержки в C++-приложениях путем регулируемого агрессивного встраивания функций.

Постановка задачи

В качестве задержки в рамках работы рассматривается среднее время работы программы (throughput).

Цель данной работы заключается в создании программной системы, основанной на LLVM [3], которая позволит сократить задержку в высоконагруженных секциях кода C++ приложений. В качестве метода оптимизации было выбрано агрессивное встраивание функций. Разработанная программа должна предоставлять пользователям возможность оптимизировать задержку в проектах на языке C++.

Таким образом, от конечного результата ожидается:

- Предоставить пользователям программную систему, позволяющую применять регулируемую оптимизацию. Одним из важных параметров может служить ограничение на размер файла после исполнения оптимизации.

- Получение значительного улучшения задержки хотя бы одного проекта в сравнении с компиляцией без использования разработанной программной системы на одинаковых версиях компилятора и флагах компиляции.

1. Обзорный раздел по предметной области

1.1. Встраивание функции

Встраивание функции – оптимизация компилятора, направленная на замену вызова функции непосредственно телом функции. Данное действие может улучшить скорость работы программы, так как после встраивания отсутствуют накладные расходы на вызов функции. Важно заметить, что порядок применения оптимизаций компилятором тоже имеет значение, поэтому оптимизации, примененные после встраивания функций могут нести лучший эффект.

Однако стоит учитывать, что встраивание функций может также привести к увеличению размера кода, что может повысить нагрузку на кэш и память, что в свою очередь может привести к ухудшению производительности. Также замедляется компиляция программы. Поэтому встраивание функций следует применять осторожно и с учетом специфики конкретной программы.

Трудность заключается в том, что даже наличие знания о специфике не позволяет с легкостью влиять на встраивание функций. Например, существует спецификатор inline [4] в реализации LLVM добавляющий некоторое численное значение к стоимости встраивания, которое используется для проверки целесообразности встраивания. То есть использование данного спецификатора не гарантирует встраивание, а лишь является некоторой подсказкой компилятору. Также существует атрибут noinline [5], запрещающий встраивание функции (исключением является встраивание вследствие свёртки констант [6]).

В целях оптимизации широко применяется Profile Guided Optimization (PGO) [7], суть которого заключается в инструментации во время работы с данными. Благодаря этому возможно сделать более точные выводы о тех или иных оптимизациях. Если вернуться к встраиванию функций, PGO может помочь компилятору, например, в ситуации, когда одна функция вызывается сильно чаще другой. В таком случае компилятор может с большей уверенностью сделать вывод о встраивании первой функции.

1.2. LLVM

LLVM – это программная система, представляющая собой набор инструментов для разработки компиляторов, ассемблеров и других систем, связанных с обработкой программного кода. LLVM предоставляет универсальный, низкоуровневый и модульный интерфейс для компиляции программ, который может быть использован для разработки компиляторов для различных языков программирования.

Процесс компиляции в LLVM включает в себя несколько этапов. Во время первого этапа, исходный код программы переводится в промежуточное представление (intermediate representation) LLVM IR с помощью компилятора Clang [8] при помощи вызова программы clang++ [9] или другого совместимого компилятора. Затем IR передается в оптимизатор LLVM, который применяет различные оптимизации, такие как встраивание функций, удаление недостижимого кода и другие, для улучшения производительности и качества сгенерированного кода. Точкой входа является команда opt [10].

Далее, оптимизированное IR преобразуется в машинный код целевой архитектуры, используя генератор кода LLVM. При генерации машинного кода LLVM использует множество оптимизаций и техник генерации кода для создания быстрого и эффективного исполняемого файла. Это реализовано в рамках программы llc [11].

В итоге LLVM обеспечивает высокую эффективность и гибкость в компиляции программного кода, что делает его популярным инструментом для разработки компиляторов, ассемблеров и других систем, связанных с обработкой программного кода.

В рамках данной работы необходимо модифицировать процесс компиляции, что делает выбор LLVM оправданным.

1.3. Аналоги

Вместе с научным руководителем производились попытки найти аналоги. Мною для поиска существующих решений были сделаны следующие запросы в поисковую систему Google:

- «agressive function inlining c++ optimizator»

- «function inlining c++ optimizator»
- «function inlining c++ optimization program»
- «function inlining c++ tuner»

Готовых решений не нашлось. Удалось обнаружить научные статьи на смежные темы, о которых речь пойдет в следующем разделе. Сравниться с результатами статей не представляется возможным, так как нет в свободном доступе ни проектов, на которых производились измерения, ни самого кода, с помощью которого данные измерения проводились.

1.4. Обзор литературы

Далее описаны научные статьи, в рамках которых изучается встраивание функций с довольно разных точек зрения.

1.4.1 Aggressive Function Splitting for Partial Inlining

В рамках работы [12] разработан способ частичного встраивания функций: найден новый подход к описанию функций, который позволяет разбивать их на основе подграфа вызовов. Также особое внимание было уделено минимизации дублирования кода. Предложен новый анализ затрат и выгод при оценке стоимости встраивания, который позволил гарантировать контроль за накладными расходами на вызов выделенной функции.

Авторы реализовали частичное встраивание поверх LLVM. В результате данная работа действительно решила задачу реализации частичного встраивания функций с минимальным дублированием кода, но время работы в половине случаев стало больше, а в другой половине – незаметно увеличилось (до 3%).

1.4.2 Automatic Tuning of Inlining Heuristics

В статье [13] описывается проблема выбора эвристик встраивания функций в процессе компиляции, которая может сильно влиять на производи-

тельность приложения. Авторы предлагают метод автоматической настройки эвристик встраивания функций на основе генетических алгоритмов.

Были описаны эксперименты, в которых метод автоматической настройки эвристик был применен для встраивания функций в наборе тестовых программ. Результаты показали, что автоматическая настройка эвристик встраивания функций может значительно улучшить производительность приложений по сравнению с настройками, установленными по умолчанию.

Исследование, проведенное в этой статье, имеет большое значение для оптимизации производительности приложений. Метод автоматической настройки эвристик встраивания функций может быть применен в различных областях, в которых требуется оптимизация производительности кода.

1.4.3 Function Inlining with Code Size Limitation in Embedded Systems

Статья [14] описывает алгоритм для встраивания функций в программы, которые выполняются на микроконтроллерах с ограниченным объемом памяти.

Авторы статьи предлагают метод, который позволяет определить, какие функции следует встраивать, чтобы минимизировать размер кода, при этом сохраняя производительность программы. Для этого используется алгоритм, который оценивает влияние встраивания каждой функции на размер кода и на время выполнения программы. Авторы также учитывают зависимости между функциями, чтобы избежать ошибок во время выполнения.

В результате экспериментов авторы статьи показали, что их метод позволяет достичь существенного уменьшения размера кода и улучшения производительности программы на микроконтроллерах с ограниченным объемом памяти. Таким образом, этот метод может быть полезен для разработчиков, работающих с встроенными системами, где ограничен объем доступной памяти, и требуется максимально эффективное использование ресурсов.

1.5. Вывод

Найденные научные статьи на тему оптимизаций путем встраивания функций не ставили перед собой задачу, совпадающую с поставленной в рам-

ках данной работы, поэтому использовать выводы данных статей не представляется возможным: в некоторых статьях оптимизировалось не время работы программы, а размер кода после встраивания функций.

Востребованность смежных по теме статей демонстрирует, что в целях оптимизации (в случае данной работы – оптимизации задержки) полезной может быть любая программная система, которая добавляет возможность провести эксперимент для более удачной компиляции программы.

2. Программная реализация

Программная реализация использует C++17, LLVM версии 12 [15], систему сборки CMake [16] версии 3.26.3 и Python версии 3.8 [17].

В рамках данной работы необходимо реализовать агрессивное регулируемое встраивание функций. То есть с одной стороны встраивание функций должно приводить к увеличению размера исполняемого кода, с другой стороны хочется предоставить пользователю возможность регулировать данный процесс.

Хочется выделить два параметра, которые должны регулировать процесс встраивания. Первый параметр – агрессивное встраивание используется только для функций, которые пользователь пометил специальным новым атрибутом aggressiveInline. Это позволит пользователю проводить огромное множество экспериментов в поисках лучшей для специфичного случая оптимизации. Второй – ограничение на размер файла, который получается после применения оптимизации. Это позволит пользователю не позволять оптимизации встраивать слишком много кода, ведь это может быть непозволительно в некоторых случаях.

2.1. Основные компоненты

Назовем **локальным оптимизатором** программную систему, позволяющую запускать оптимизацию LLVM (команду opt) так, чтобы на помеченных специальным атрибутом aggressiveInline функциях срабатывало встраивание функций. Для контроля агрессивности встраивания используются фиксированные параметры.

Назовем **оптимизатором** программу, позволяющую находить в определенном пространстве поиска параметры **локального оптимизатора**, максимизирующие размер файла (в байтах), полученного после применения **локального оптимизатора**. У **оптимизатора** также нужно предусмотреть ограничение сверху на размер выходного файла.

Назовем **модификатором** программу, позволяющую модифицировать команды компиляции проекта с системой сборки CMake в цепочку команд, среди которых появляется обращение к **оптимизатору**.

Назовем **исполнителем** программу, позволяющую запускать команды компиляции.

Реализация каждой компоненты предоставляет решение первой задачи, сформулированной в постановке задачи: для запуска оптимизации необходимо воспользоваться **модификатором** и **исполнителем**.

2.2. Локальный оптимизатор

В данном разделе будет приведено описание реализации **локального оптимизатора**.

Локальный оптимизатор должен предоставить программную систему, которая позволяет запускать встраивание функций на функциях, помеченных атрибутом aggressiveInline. Агрессивность встраивания должна настраиваться внешне при помощи аргументов командной строки, так как далее нужно производить оптимизацию относительно агрессивности встраивания.

Предлагается декомпозировать решение на следующие задачи:

- Добавление атрибута aggressiveInline
- Добавление компоненты, отвечающей за встраивание функций
- Добавление параметров команды opt, отвечающих за агрессивность встраивания функций

2.2.1 Добавление атрибута

Поймем для начала какие части компилятора могут использовать атрибут. Ясно, что использование пользователями атрибута влечет за собой необходимость обрабатывать исходный код программы, таким образом необходимо модифицировать первый этап компиляции. Атрибуты функций могут влиять и на оптимизации (второй этап компиляции), и на выбор инструкций при преобразовании в машинный код (третий этап компиляции). То есть для добавления атрибута необходимо модифицировать все три части компиляции.

Была проведена модификация нескольких файлов компилятора Clang и программной системы LLVM:

- clang/include/clang/Basic/Attr.td и clang/include/clang/Basic/AttrDocs.td, отвечающие за определение, описание и документирование всех атрибутов Clang. В них необходимо было добавить стиль написания атрибута: в C++ разрешается использовать различные стили, например, __attribute__((noinline)) либо [[noinline]]. В данном случае используется первый вариант. Также необходимо уточнить сущность, которую можно связать с атрибутом. Здесь это функции.

- clang/lib/CodeGen/CodeGenModule.cpp и

  clang/lib/Sema/SemaDeclAttr.cpp. Первый файл отвечает за генерацию кода из AST, второй – за обработку атрибутов и проверку применимости атрибутов к соответствующим декларациям.

- llvm/include/llvm/Bitcode/LLVMBitCodes.h,

  llvm/lib/Bitcode/Reader/BitcodeReader.cpp и

  llvm/lib/Bitcode/Writer/BitcodeWriter.cpp были изменены для добавления нового атрибута aggressiveInline в обработку биткода. Биткод используется при работе с бинарным представлением файлов LLVM.

- llvm/include/llvm/IR/Attributes.td отвечает за определение атрибутов внутри LLVM IR.

- llvm/lib/AsmParser/LLLexer.cpp,

  llvm/lib/AsmParser/LLParser.cpp и

  llvm/lib/AsmParser/LLToken.h были изменены для поддержки нового атрибута aggressiveInline в парсере LLVM IR.

- llvm/lib/IR/Attributes.cpp отвечает за работу с атрибутами внутри системы LLVM.

- llvm/lib/IR/Verifier.cpp занимается проверкой корректности и согласованности LLVM IR.

Таким образом, была проведена комплексная модификация компилятора Clang и системы LLVM, что делает использование атрибута aggressiveInline возможным.

В целях тестирования работоспособности предложенного подхода был реализован LLVM Pass [18], который печатает в консоль LLVM IR всякой функции, помеченной новым атрибутом при использовании команды llc (третий этап компиляции) в режиме отладки.

Например, для функции

```
int fib(int n) __attribute__((aggressiveInline)) {
  if (n == 0) {
    return 1;
  }
  if (n == 1) {
    return 2;
  }
  return fib(n - 1) +  fib(n - 2);
}
```

частью вывода на экран является

```
Contents of BasicBlock corresponding to MachineBasicBlock:
0x5613102bb850
Contents of MachineBasicBlock:
bb.1.cond.true:
; predecessors: %bb.0
  successors: %bb.3
  JMP_1 %bb.3
```

2.2.2 Встраивание функций

Встраивание функций в терминах LLVM является межпроцедурной оптимизацией (interprocedural optimization, IPO). llvm/lib/Transforms/IPO – директория, содержащая реализации нескольких LLVM Pass, производящих модификации оптимизируемого LLVM IR. Например, ArgumentPromotion.cpp

оптимизирует вызовы функций, заменяя аргументы на константы, если это возможно. Данные оптимизации запускаются при использовании команды opt (по умолчанию запускается лишь некоторая часть, для использования определенных IPO существуют определенные аргументы запуска).

Для реализации требуемой компоненты необходимо создать трансформирующий LLVM Pass, реализующий класс LegacyInlinerBase, содержащий логику определения целесообразности встраивания функций. Целесообразность встраивания функции определяется путем оценивания стоимости встраивания функции (класс InlineCost). Также существует верхняя граница стоимости: если встраивание функции имеет стоимость больше верхней границы, то считается, что данное встраивание убыточно. Полный спектр всех правил определения целесообразности встраивания функций содержится в директории llvm/lib/Analysis.

Таким образом, для того, чтобы реализовать контролируемое агрессивное встраивание функций, необходимо реализовать класс, наследующийся от LegacyInlinerBase и переопределяющий метод, ответственный за вычисление стоимости встраивания.

В рамках данной работы достаточно определить вычисление стоимости и верхней границы стоимости константными для функций, помеченных атрибутом aggressiveInline.

В коде это выглядит так:

```
InlineCost getInlineCost(CallBase &CB) override {
    Function *Callee = CB.getCalledFunction();
    if (!Callee || Callee->isDeclaration() ||
    !Callee->hasFnAttribute(
    Attribute::AttrKind::AggressiveInline)) {
      return InlineCost::getNever("Not suitable");
    }
    return InlineCost::get(Cost, Threshold);
}
```

Переменные Cost и Threshold – численные константы, определенные при создании класса. Именно они и являются параметрами, которые регулируют

агрессивность встраивания функции.

2.2.3 Агрессивность встраивания функций

Для того чтобы реализованный выше компонент можно было контролировать при запусках opt, необходимо предоставить возможность фиксировать стоимость и верхнюю границу стоимости в момент запуска opt. Было принято решение передавать вышеупомянутые параметры, используя параметры командной строки при вызове opt.

LLVM предоставляет удобный интерфейс для этого. Достаточно лишь создать статическую переменную с типом, для которого уже реализовано взаимодействие с параметрами вызова:

```
static cl::opt<int>
MyInlinerCost("my-inliner-cost", cl::init(0),
        cl::desc("My inliner cost for optimization"));
```

Теперь вышеупомянутые переменные Cost и Threshold определяются при создании класса значениями статических переменных MyInlinerCost и MyInlinerThreshold, что позволяет настраивать агрессивность встраивания вызовом команды opt с определенными аргументами командной строки.

2.2.4 Промежуточный результат

Локальный оптимизатор – модифицированная версия команды opt. Ею можно пользоваться так:

```
opt -O2 -my-inliner -my-inliner-cost=30
-my-inliner-threshold=400 main.bc -o main.opt.bc
```

Здесь -my-inliner – название реализованного LLVM Pass, которое необходимо запустить при оптимизации LLVM IR кода. -my-inliner-cost и -my-inliner-threshold – константы, отвечающие за агрессивность встраивания помеченных атрибутом aggressiveInline функций.

2.3. Оптимизатор

Оптимизатор [19] должен находить оптимальные параметры агрессивности **локального оптимизатора** в рамках задачи максимизации выходящего размера файла. Также для дополнительной регуляризации процесса необходимо предоставить возможность задавать пространство поиска вышеописанных параметров.

Можно было бы реализовать поиск оптимальных параметров непосредственно внутри opt, но в C++ библиотеки для решения задач оптимизации обычно имеют намного более сложный интерфейс, чем, например, в Python. В связи с этим для реализации **оптимизатора** было принято решение использовать Python, внутри которого необходимо расположить вызовы opt.

Заметим, что под максимально агрессивным встраиванием функций мы имеем в виду встраивание как можно большего количества кода, что равносильно максимизации размера выходного файла. Таким образом, необходимо произвести оптимизацию многомерной функции, аргументы которой – параметры, передаваемые **локальному оптимизатору**, а возвращаемое значение – размер файла после вызова команды opt с данными аргументами.

Для оптимизации данной функции используется scipy [20]. В рамках данной библиотеки есть множество решений [21], позволяющих оптимизировать многомерную функцию с ограничением в пространстве поиска.

В рамках поставленной задачи нет необходимости в высокой точности, поэтому при выборе решения обращалось внимание лишь на возможность использования и на скорость работы. В результате выбор пал на дифференциальную эволюцию [22].

Дифференциальная эволюция – метод многомерной математической оптимизации, относящийся к классу стохастических алгоритмов оптимизации (то есть работает с использованием случайных чисел) и использующий некоторые идеи генетических алгоритмов, но, в отличие от них, не требует работы с переменными в бинарном коде. Это прямой метод оптимизации, то есть он требует только возможности вычислять значения целевой функции, но не её производных. Метод дифференциальной эволюции предназначен для нахождения глобального минимума (или максимума) недифференциру-

емых, нелинейных, мультимодальных (имеющих, возможно, большое число локальных экстремумов) функций от многих переменных.

Приложение принимает следующие обязательные параметры:

- inliner_input_file – путь файла бинарного формата LLVM IR (биткод), который необходимо оптимизировать
- inliner_output_file – путь для сохранения вывода алгоритма оптимизации, тоже является бинарным форматом LLVM IR
- inliner_inline_lines_upper_bound – ограничение сверху на размер встроенного кода (в байтах)

Также предусмотрен ряд необязательных параметров:

- inliner_opt_path – путь к инструменту opt оптимизаций LLVM. Необходимо, чтобы эта команда была собрана на проекте LLVM, в котором имеется реализация **локального оптимизатора**.
- inliner_arguments – дополнительные аргументы для внутреннего запуска opt. Может быть полезно пользоваться, например, аргументом -O2.
- inliner_cost_left_bound, inliner_cost_right_bound,

  inliner_threshold_left_bound, inliner_threshold_right_bound – параметры, задающие границы пространства поиска параметров my-inliner-cost и my-inliner-threshold для вызова **локального оптимизатора**
- inliner_cores_to_use – число, которе необходимо подставить в аргумент workers при вызове дифференциальной эволюции. Данный аргумент отвечает за распараллеливание оптимизации.
- inliner_tmp_folder_name – название подпапки в директории вызова **оптимизатора**, в которое будут помещаться временные файлы, необходимые для работы.

Реализация **оптимизатора** заключается в том, чтобы оптимизировать функцию, возвращающую размер файла, получающегося при использовании **локального оптимизатора** с параметрами my-inliner-cost и my-inliner-threshold, оптимизируемыми при помощи дифференциальной эволюции.

В результате **оптимизатором** является программа optimizer.py. Его использование может выглядеть следующим образом:

```
python3.8 optimizer.py --inliner_input_file=tinyxml2.cpp.bc
--inliner_output_file=tinyxml2.cpp.bc.out
--inliner_inline_lines_upper_bound=1600000
--inliner_cost_right_bound=100
--inliner_threshold_right_bound=500
--inliner_cores_to_use=16 --inliner_arguments=-O2
```

2.4. Модификатор

Реализованное выше довольно трудно применять на практике, так как работа производится на уровне LLVM IR, поэтому в данном разделе приводится попытка модифицировать процесс сборки проекта таким образом, чтобы автоматически вызывать **оптимизатор**. Это необходимо хотя бы для того, чтобы провести измерения задержки оптимизированных проектов.

Так как сборка проектов сама по себе является очень сложной и большой темой, в рамках данной работы предлагается сделать ряд допущений:

- Система сборки – CMake
- Использование CMAKE_EXPORT_COMPILE_COMMANDS [23] генерирует корректный compile_commands.json.

Рассмотрим типичный пример compile_commands.json (пути до файлов для краткости сокращены):

```
[
{
  "directory": "Diploma/RedBlackTree",
  "command": "Diploma/llvm-project/build/bin/clang++
  -I/Diploma/RedBlackTree/include -std=gnu++17 -o
  CMakeFiles/example_RBT.dir/examples/RBT.cpp.o -c
  Diploma/RedBlackTree/examples/RBT.cpp",
  "file": "Diploma/RedBlackTree/examples/RBT.cpp",
```

```
  "output": "CMakeFiles/example_RBT.dir/examples/RBT.cpp.o"
}
]
```

Модификатор разбивает одну команду компиляции на 4:

- Для начала компилируемый исходный файл преобразуется не в объектный файл, а в биткод в формате LLVM IR. Это достигается использованием флага emit-llvm [24]. Необходимо использовать модифицированную сборку LLVM, в которой реализован **локальный оптимизатор**.
- Далее над полученным файлом в бинарном формате LLVM IR вызывается **оптимизатор**.
- Полученный оптимизированный файл преобразуется в машинный код для целевой архитектуры. Для этого необходимо использовать команду llc.
- Наконец, используется команда as для получения объектного файла из машинного кода.

Для примера выше получается следующее:

```
[
{
   "command": "Diploma/llvm-project/build/bin/clang++ -
   I/Diploma/RedBlackTree/include -std=gnu++17  -emit-llvm -o
   CMakeFiles/example_RBT.dir/examples/RBT.cpp.ll -c
   Diploma/RedBlackTree/examples/RBT.cpp",
   "directory": "Diploma/RedBlackTree",
   "file": "Diploma/RedBlackTree/examples/RBT.cpp"
},
{
   "command": "python3.8 optimizer.py --inliner_input_file=
   Diploma/RedBlackTree/CMakeFiles/example_RBT.dir/examples/
   RBT.cpp.ll --inliner_output_file=Diploma/RedBlackTree/
```

```
    CMakeFiles/example_RBT.dir/examples/RBT.cpp.ll.out
    --inliner_inline_lines_upper_bound=50000
    --inliner_cores_to_use=16",
    "directory": "Diploma/llvm-project/scripts",
    "file": "Diploma/RedBlackTree/examples/RBT.cpp"
},
{
    "command": "Diploma/llvm-project/build/bin/llc
    Diploma/RedBlackTree/CMakeFiles/example_RBT.dir/
    examples/RBT.cpp.ll.out
    -o Diploma/RedBlackTree/CMakeFiles/example_RBT.dir/
    examples/RBT.cpp.ll.s",
    "directory": "Diploma/llvm-project/scripts",
    "file": "Diploma/RedBlackTree/examples/RBT.cpp"
},
{
    "command": "as Diploma/RedBlackTree/CMakeFiles/
    example_RBT.dir/examples/RBT.cpp.ll.s -o
    CMakeFiles/example_RBT.dir/examples/RBT.cpp.o",
    "directory": "Diploma/RedBlackTree",
    "file": "Diploma/RedBlackTree/examples/RBT.cpp"
}
]
```


В итоге **модификатор** [25] создает JSON-файл [26], который довольно просто модифицировать для экспериментов в оптимизации. В частности, если необходимо, можно передать дополнительные аргументы оптимизации **оптимизатору** либо команде llc.

Таким образом реализована программа modify_compile_commands.py, принимающая путь к файлу, содержащему команды компиляции. Также она принимает путь, по которому должен быть расположен модифицированный файл и флаги оптимизации, используемые в вызовах команд компиляции.

2.5. Исполнитель

Исполнитель [27] – программа run_compile_commands.py, которая запускает команды компиляции в порядке, заданном в файле с командами компиляции, параллельно выводя на экран текущую исполняемую команду. Принимает единственный аргумент – путь к файлу.

Реализация данной компоненты сводится к разбору JSON-файла и последовательному применению команд, расположенных в файле.

2.6. Пример использования

Реализована программная система, пользоваться которой можно, следуя следующим шагам:

- Собрать модифицированную версию LLVM (это необходимо сделать единожды)
- Форсировать генерацию compile_commands.json
- Вызвать команду cmake
- Модифицировать compile_commands.json
- Запустить команды из модифицированного compile_commands.json
- Вызвать команду make [28]

Рассмотрим использование данной работы на примере тестового проекта [29], посвященному парсингу XML [30] файлов. Модифицируем файл tinyxml2/tinyxml2.cpp, расставив __attribute__((aggressiveInline)) на все функции и методы. Указываем в CMakeLists.txt путь до собранной модифицированной версии Clang++ и форсируем генерацию списка команд компиляции compile_commands.json:

```
set(CMAKE_CXX_COMPILER Diploma/llvm-project/build/bin/clang++)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

Далее вызываем команду cmake CMakeLists.txt, затем необходимо модифицировать список команд компиляции при помощи **модификатора**:

```
python3.8 modify_compile_commands.py --build_input=
Diploma/tinyxml2/compile_commands.json --build_output=
Diploma/tinyxml2/compile_commands_cmake.opt.json
--build_opt_flag=-O2
```

Осталось запустить **исполнитель** и собрать проект вызовом команды make [28]:

```
python3.8 run_compile_commands.py  --compile_command_path=
Diploma/tinyxml2/compile_commands_cmake.opt.json
```

В результате собранные исполняемые файлы собраны с использованием агрессивного встраивания функций, разработанного в данной работе.

3. Тестирование и результаты

3.1. Тестирование

Для измерения среднего времени работы программы были написаны дополнительные программы, симулирующие нагрузку на тестируемые проекты. Также в случае декартового дерева необходимо было зафиксировать инициализацию генератора случайных чисел, так как в противном случае измерения проводились бы на деревьях разной структуры.

Важным аспектом тестирования является обеспечение одинаковых условий выполнения тестовых измерений. Среда исполнения может влиять на результаты измерений и их точность.

Виртуальная машина является одним из способов обеспечения одинаковых условий тестирования. Виртуальный сервер позволяет создать изолированную среду, которая может быть настроена для максимальной производительности и устойчивости к внешним воздействиям. Для тестирования был выбран хостинг Yandex Cloud [31], в котором была создана виртуальная машина со следующими характеристиками:

- HDD [32] диск
- Процессор Intel Xeon Gold 6338 с 4 vCPU и тактовой частотой процессора 2 ГГц
- Оперативная память 8 Гб

Изолирование ядра также является важным аспектом тестирования. Это позволяет избежать влияния других процессов на результаты измерений. В частности, изолирование ядра позволяет уменьшить влияние операционной системы на производительность приложения.

3.2. Проекты для тестирования

Для тестирования и измерения задержек было выбрано три проекта, расположенные в свободном доступе в github: tinyxml2 [29], RedBlackTree [33] и treap [34]:

- Проект tinyxml2 предназначен для эффективного парсинга XML файлов.
- RedBlackTree содержит в себе реализацию красно-черного дерева [35].
- treap содержит в себе реализацию декартового дерева [36].

3.3. Измерение задержки

Хоть задача поставлена для задержки в смысле среднего времени работы (throughput), в данном разделе приводятся также и персентили. Таким образом, можно использовать, например, 99-ый персентиль для измерения задержки (в смысле latency).

Все тестовые измерения задержек были произведены в изолированном ядре. Для минимизации шума при измерениях программы, симулирующие нагрузку, были запущены 2000 раз.

Измерения задержек производились для разных оптимизационных флагов компиляции: O0, O2 и O3. Данные флаги передавались clang++, opt и llc.

Вариантов расстановки атрибута aggressiveInline в данных проектах очень много. Весь процесс сборки, измерения времени работы и подведения результатов занимает существенное количество времени, поэтому было принято решение проводить измерения при расстановке атрибута на всех функциях и методах.

Измерения производятся без ограничения на размер файла, выводящийся **оптимизатором**.

Все проценты сокращаются до двух знаков после запятой стандартным математическим правилом сокращения.

Отчеты находятся в github репозитории в папке results2 [37], скрипты, производящие измерения в папке выше [38]. Графики формировались блокнотом [39].

3.3.1 RedBlackTree

На рисунке 1 изображены значения персентилей времени работы реализации RedBlackTree с разными способами компиляции. Модификация –

разработанная в рамках данной дипломной работы программная система, базовый компилятор – компилятор на базе LLVM версии 12. В названии рисунка указывается тестируемый проект и оптимизационный флаг, использующийся при запуске. Время по вертикальной оси указано в секундах.

В случае флага O0 среднее время работы базового компилятора составило примерно 0.156 секунд, модификация показала примерно 0.148 секунд. Таким образом, получено ускорение на 4.95%.

На рисунке 2 изображены те же измерения, но при использовании флага O2. Заметим, что среднее время работы базового компилятора значительно ускорилось, таким образом, при наличии флагов оптимизации получать ускорение становится сложнее. В частности, модификация показала ускорение среднего времени работы на 1.21%.

Перейдем к флагу оптимизации O3. На рисунке 3 изображен график, показывающий, что агрессивное встраивание на некоторых персентилях ухудшило время работы. Среднее время работы увеличилось на 0.38%.

3.3.2 tinyxml2

На рисунках 4, 5, 6 изображены измерения времени работы tinyxml2 с флагами компиляции O0, O2, O3 соответственно.

Агрессивное встраивание функций хорошо показывает себя на флаге компиляции O0, ускорив среднюю скорость работы на 11.3%.

При использовании O2 тенденция уменьшения времени работы сохраняется на всех персентилях. Среднее время работы ускоряется на 1.79%.

Флаг O3 также сохраняет тенденцию уменьшения времени работы при использовании разработанной программной системы. Ускорение среднего времени работы получилось равно 1.66%.

3.3.3 treap

Измерения на рисунках 7, 8, 9 демонстрируют значительное ускорение: например, при использовании флага оптимизации O2 сформирован следующий отчет для базового компилятора:

Total time taken: 319.025875 seconds
Average time taken: 0.159513 seconds
Variance: 0.0000529622295807
Standard deviation: 0.0072775153439022
10th percentile: 0.151603 seconds
25th percentile: 0.154321 seconds
50th percentile: 0.158299 seconds
75th percentile: 0.163166 seconds
90th percentile: 0.169203 seconds
95th percentile: 0.173528 seconds
99th percentile: 0.181843 seconds

И такой для модификации:

Total time taken: 269.129571 seconds
Average time taken: 0.134565 seconds
Variance: 0.0000399415451146
Standard deviation: 0.0063199323662995
10th percentile: 0.128123 seconds
25th percentile: 0.130002 seconds
50th percentile: 0.133393 seconds
75th percentile: 0.137574 seconds
90th percentile: 0.142931 seconds
95th percentile: 0.146749 seconds
99th percentile: 0.155217 seconds

Получены следующие ускорения среднего времени работы:

- при O0 – на 12.5%
- при O2 – на 15.64%
- при O3 – на 14.98%

Данные демонстрируют статистически значимое ускорение среднего времени работы на всех флагах компиляции, где статистическая значимость определяется t-критерием Уэлча [40] с уровнем значимости 0.001.

Также видно, что ускорение характерно для всех указанных персентилей.

3.4. Размеры исполняемых файлов

Как уже упоминалось, все замеры производились без ограничения на размер вывода **оптимизатора**. Рассмотрим, насколько увеличились размеры исполняемых файлов после использования реализованной программной системы.

Из таблицы 1 видно, что при флаге O0 размер файла может увеличиться в сотни раз. Для двух из трех проектов произошло увеличение в 2-3 раза.

А при флаге O2 из таблицы 2, что раз исполняемых файлов увеличивается в 1.7-3 раза.

Включение флага оптимизации O3 не сильно отличается от O2 в плане размера исполняемых файлов. Таблица 3 практически идентична таблице 2, изменения размеров исполняемых файлов проектов treap и RedBlackTree составляют сотни байт.

Таким образом, разработанная программная система увеличивает размеры файлов при отсутствии ограничений. Но на тестовых проектах значительное увеличение (на два порядка) получилось лишь в случае отсутствия оптимизационных флагах компиляции.

Выводы

Удалось реализовать программную систему, позволяющую оптимизировать задержку приложений на C++, используя регулируемое агрессивное встраивание функций. Регулируемость реализуется многими параметрами:

- Пользователь сам выбирает, какие функции можно пытаться агрессивно встраивать, а какие – нет. Для этого необходимо помечать функции специальным атрибутом aggressiveInline.

- Пользователь может использовать различные флаги оптимизации, реализованные в clang++, opt и llc. Таким образом реализация данной системы позиционируется не как независимая оптимизация, блокирующая применение другие, а как дополнительная оптимизация, которая может производиться наряду с другими.

- Доступен выбор пространства поиска параметров, которые определяют структуру класса InlineCost для определения целесообразности встраивания функций, помеченных атрибутом aggressiveInline.

- Также можно ограничивать размер выходного из **оптимизатора** файла. Это может оказаться полезным, если специфика такова, что большие файлы и/или более длительная компиляция представляют существенные неудобства.

Таким образом получена программная система, предоставляющая возможность использовать регулируемую оптимизацию задержки скорости работы. Тем самым первая задача из постановки задачи в полной мере решена.

Также был произведен сравнительный анализ скорости работы на различных тестовых проектах между разработанной программной системой и базовым компилятором, в ходе которого обнаружено, что даже при использовании оптимизационных флагов компиляции удается достичь ускорения. В частности, на проекте treap удалось достичь значительного уменьшения задержки, причем увеличение размера исполняемых файлов при использовании флагов O2 и O3 не превышает троекратного. Из вышеизложенного следует, что и вторая поставленная задача решена в полной мере.

Обозначим улучшения, которые возможны:

- Сборка проекта. В данной работе был ряд допущений, вероятно, устранимые при помощи использования инструмента Bear [41], который позволяет генерировать команды компиляции для большого числа систем сборки.

- Скорость сборки проекта. Вызовы clang++, opt, llc и as достаточно сильно замедляют процесс. Вероятно, можно исключить какие-то шаги компиляции для достижения той же цели.

- Сложность реализации выдачи стоимости встраивания. В данной реализации InlineCost строится по константам, общим для всех функций оптимизируемого LLVM IR файла. Вероятно, можно исследовать структуру функций, чтобы более утонченно выдавать разные стоимости встраивания разным функциям.

- Использование более новой версии LLVM. Из-за существования некоторых багов поведения LLVM Pass в новых версиях было принято решение зафиксировать довольно старую версию LLVM 12. Вероятно, в новых версиях предоставлен более богатый и мощный набор инструментов для оптимизаций.

- Заметим, что **оптимизатор** работает на уровне LLVM IR. Это позволяет добавлять новые языки программирования для осуществления предложенной в работе оптимизации.

Заключение

Проведено исследование статей на смежные темы и выявлены различные подходы к встраиванию функций.

Основным результатом данной работы является реализованная программная система, позволяющая оптимизировать среднее время выполнения программ на C++ при помощи агрессивного встраивания функций. Широкий набор параметров для регулирования агрессивности является существенным преимуществом разработанного решения.

На базе созданной системы произведены измерения среднего времени работы нескольких тестовых проектов. На одном из тестовых проектов выявлено значительное ускорение до 15%. Произведенные измерения демонстрируют возможность ускорения программ при помощи использования разработанной системы.

Благодарность

Автор выражает особую благодарность Александру Сергеевичу Туинову за регулярное внимание и наставления к данной работе.

Список литературы

[1] *Clang command line arguments*. URL: https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options. (дата обращения 20.05.2023).

[2] *Function inlining*. URL: https://en.wikipedia.org/wiki/Inline_function. (дата обращения 20.05.2023).

[3] *LLVM*. URL: https://llvm.org/. (дата обращения 20.05.2023).

[4] *Inline specifier*. URL: https://en.cppreference.com/w/cpp/language/inline. (дата обращения 20.05.2023).

[5] *noinline attribute*. URL: https://clang.llvm.org/docs/AttributeReference.html#noinline. (дата обращения 20.05.2023).

[6] *Constant folding*. URL: https://en.wikipedia.org/wiki/Constant_folding. (дата обращения 20.05.2023).

[7] *Profile guided optimizetion*. URL: https://llvm.org/docs/HowToBuildWithPGO.html#introduction. (дата обращения 20.05.2023).

[8] *Clang*. URL: https://clang.llvm.org/. (дата обращения 20.05.2023).

[9] *clang command*. URL: https://clang.llvm.org/docs/CommandGuide/clang.html. (дата обращения 20.05.2023).

[10] *opt command*. URL: https://llvm.org/docs/CommandGuide/opt.html. (дата обращения 20.05.2023).

[11] *llc command*. URL: https://llvm.org/docs/CommandGuide/llc.html. (дата обращения 20.05.2023).

[12] Jun-Pyo Lee и др. «Aggressive Function Splitting for Partial Inlining». В: *Interaction between Compilers and Computer Architecture, Annual Workshop on* 0 (февр. 2011), с. 80—86. DOI: 10.1109/INTERACT.2011.14.

[13] John Cavazos и Michael FP O'Boyle. «Automatic tuning of inlining heuristics». В: *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE. 2005, с. 14—14.

[14] Xinrong Zhou, Johan Lilius и Lu Yan. «Function Inlining with Code Size Limitation in Embedded Systems.» В: *Int. Arab J. Inf. Technol.* 2.3 (2005), с. 214—218.

[15] *LLVM 12 github project repository*. URL: https://github.com/llvm/llvm-project/tree/release/12.x. (дата обращения 20.05.2023).

[16] *CMake*. URL: https://cmake.org/. (дата обращения 20.05.2023).

[17] *Python 3.8*. URL: https://www.python.org/downloads/release/python-380/. (дата обращения 20.05.2023).

[18] *LLVM Pass*. URL: https://llvm.org/docs/WritingAnLLVMPass.html#introduction-what-is-a-pass. (дата обращения 20.05.2023).

[19] *Optimizer source code*. URL: https://github.com/ainurbl/llvm-project/blob/release/12.x/scripts/optimizer.py. (дата обращения 20.05.2023).

[20] *SciPy*. URL: https://scipy.org/. (дата обращения 20.05.2023).

[21] *SciPy Global optimization*. URL: https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization. (дата обращения 20.05.2023).

[22] *SciPy Differential evolution*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution. (дата обращения 20.05.2023).

[23] *CMake CMAKE_EXPORT_COMPILE_COMMANDS variable*. URL: https://cmake.org/cmake/help/latest/variable/CMAKE_EXPORT_COMPILE_COMMANDS.html. (дата обращения 20.05.2023).

[24] *LLVM emit-llvm command line argument*. URL: https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-emit-llvm. (дата обращения 20.05.2023).

[25] *Modifier source code*. URL: https://github.com/ainurbl/llvm-project/blob/release/12.x/scripts/modify_compile_commands.py. (дата обращения 20.05.2023).

[26] *JSON*. URL: https://www.json.org/json-en.html. (дата обращения 20.05.2023).

[27] *Runner source code*. URL: https://github.com/ainurbl/llvm-project/blob/release/12.x/scripts/run_compile_commands.py. (дата обращения 20.05.2023).

[28] *GNU make*. URL: https://www.gnu.org/software/make/manual/make.html. (дата обращения 20.05.2023).

[29] *C++ XML parser github repository*. URL: https://github.com/leethomason/tinyxml2. (дата обращения 20.05.2023).

[30] *XML format*. URL: https://en.wikipedia.org/wiki/XML. (дата обращения 20.05.2023).

[31] *Yandex Cloud*. URL: https://cloud.yandex.ru/. (дата обращения 20.05.2023).

[32] *Hard disk drive*. URL: https://en.wikipedia.org/wiki/Hard_disk_drive. (дата обращения 20.05.2023).

[33] *Red-black-tree data structure implementation github repository*. URL: https://github.com/MaximSurovtsev/RedBlackTree/tree/master. (дата обращения 20.05.2023).

[34] *Treap data structure implementation github repository*. URL: https://github.com/sobkulir/treap. (дата обращения 20.05.2023).

[35] *Red-black tree*. URL: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree. (дата обращения 20.05.2023).

[36] *Cartesian tree*. URL: https://en.wikipedia.org/wiki/Cartesian_tree. (дата обращения 20.05.2023).

[37] *Отчеты измерений*. URL: https://github.com/ainurbl/llvm-project/tree/release/12.x/scripts/results2. (дата обращения 20.05.2023).

[38] *Скрипт измерений*. URL: https://github.com/ainurbl/llvm-project/blob/release/12.x/scripts/pbench.py. (дата обращения 20.05.2023).

[39] *Блокнот, создававший графики*. URL: https://github.com/ainurbl/llvm-project/blob/release/12.x/scripts/ChartMaker.ipynb. (дата обращения 20.05.2023).

[40] *Welch's t-test*. URL: https://en.wikipedia.org/wiki/Welch%27s_t-test. (дата обращения 20.05.2023).

[41] *Bear*. URL: https://github.com/rizsotto/Bear. (дата обращения 20.05.2023).

Приложения

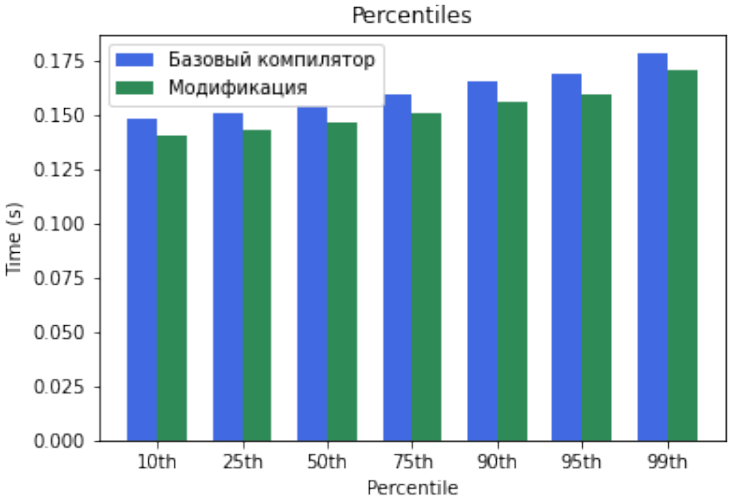

Рис. 1: RedBlackTree. Флаг O0

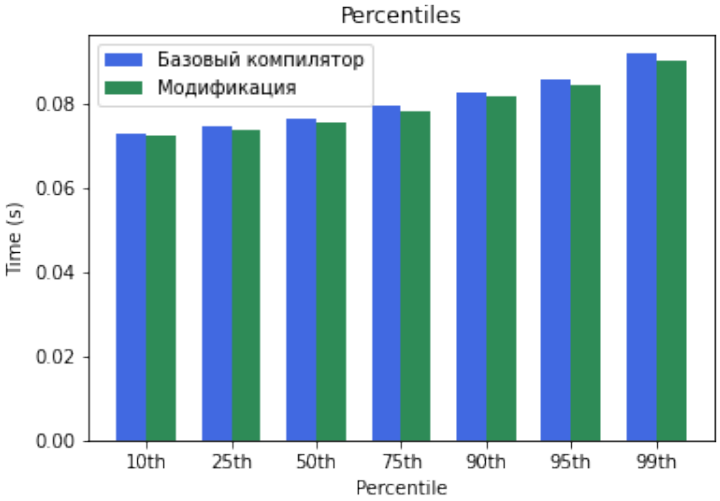

Рис. 2: RedBlackTree. Флаг O2

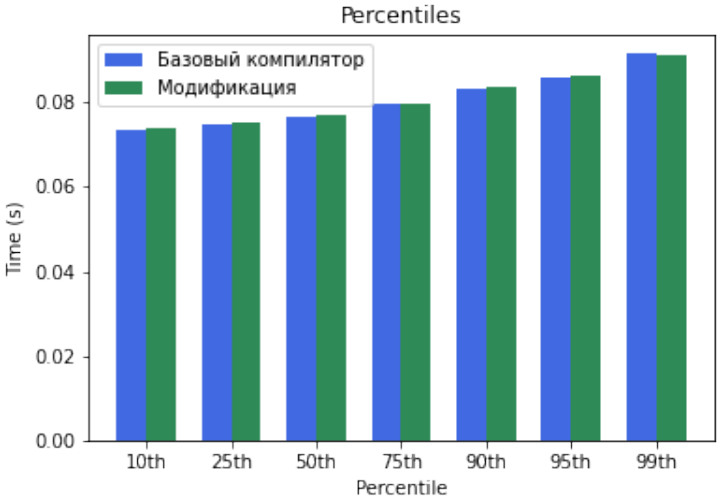

Рис. 3: RedBlackTree. Флаг O3

Тестируемый проект	Базовый компилятор	Модификация
tinyxml2	171K	423K
RedBlackTree	23K	55K
treap	28K	1.1M

Таблица 1: Сравнение размера исполняемых файлов. Флаг O0. K – размер файла в килобайтах, M – в мегабайтах.

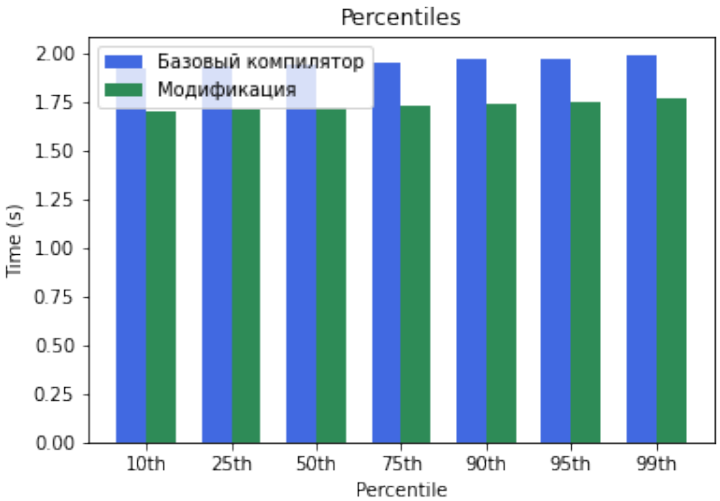

Рис. 4: tinyxml2. Флаг O0

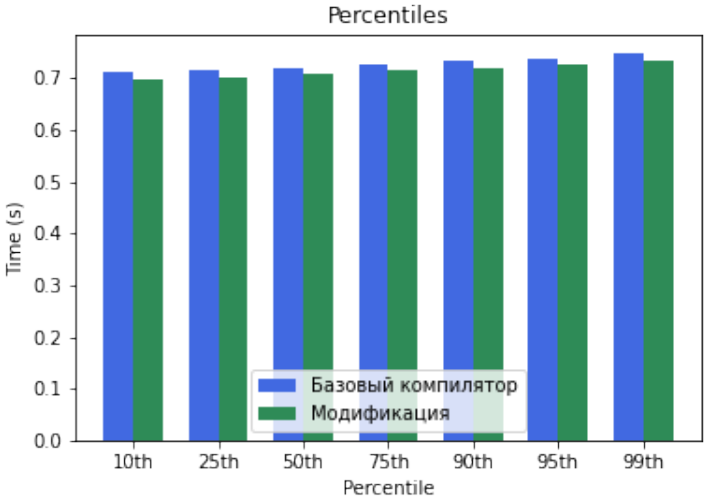

Рис. 5: tinyxml2. Флаг O2

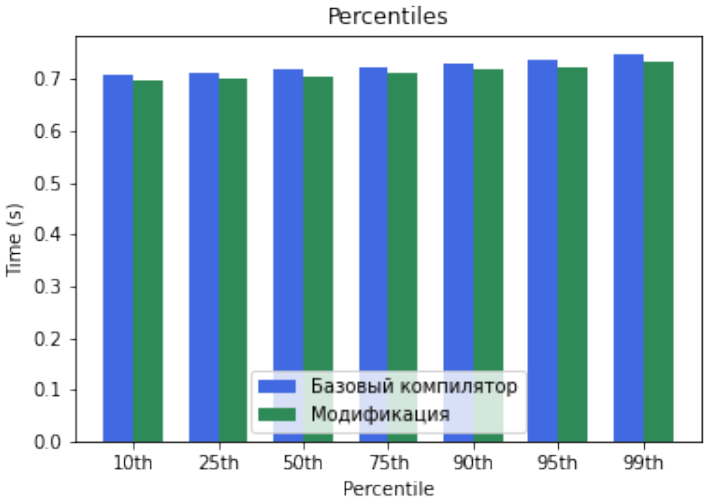

Рис. 6: tinyxml2. Флаг O3

Тестируемый проект	Базовый компилятор	Модификация
tinyxml2	177K	345K
RedBlackTree	21K	37K
treap	22K	66K

Таблица 2: Сравнение размера исполняемых файлов. Флаг O2. K – размер файла в килобайтах, M – в мегабайтах.

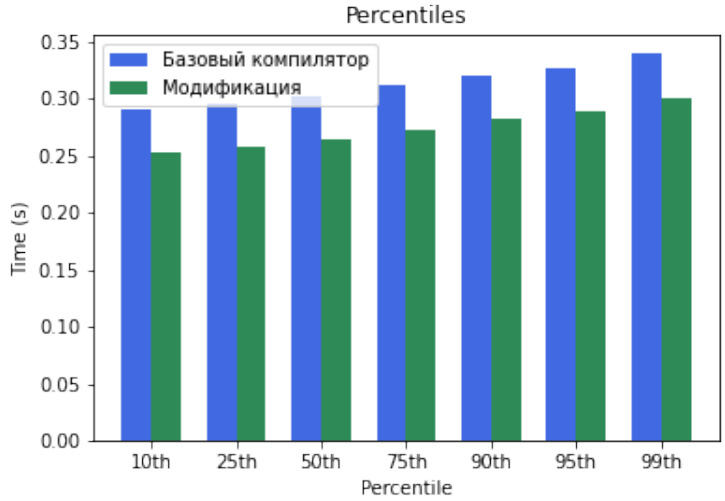

Рис. 7: treap. Флаг O0

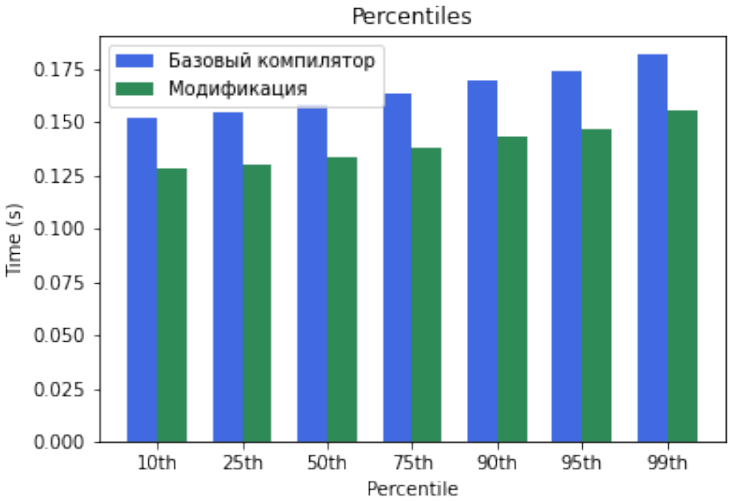

Рис. 8: treap. Флаг O2

Тестируемый проект	Базовый компилятор	Модификация
tinyxml2	185K	365K
RedBlackTree	21K	37K
treap	22K	66K

Таблица 3: Сравнение размера исполняемых файлов. Флаг O3. K – размер файла в килобайтах, M – в мегабайтах.

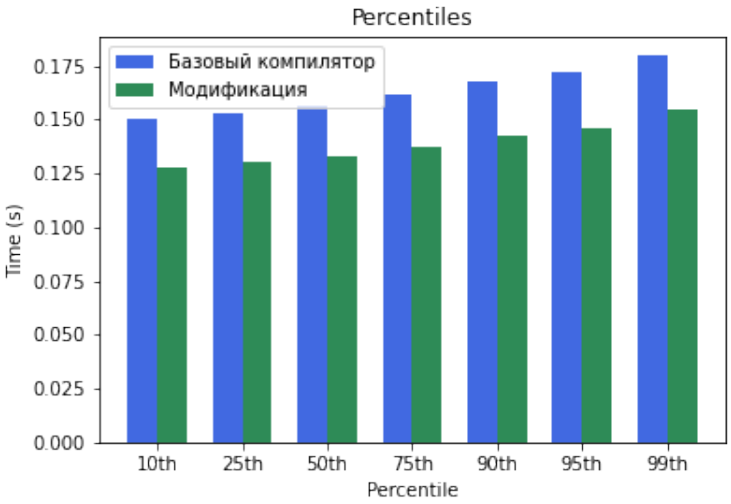

Рис. 9: treap. Флаг O3